

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 | 5/20/2023 | Dalton Walls | This is our plan to bring GameRoom’s Android game, Draw it or lose it, to browsers |

**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)

We will port the Android game “Draw it or lose it” to web browsers. In doing so we will preserve the functionality of having multiple teams (with unique player-chosen team names) and players assigned to those teams. We will preserve the functionality present in the Android version and make a web-based version that works very closely to the same.

## Requirements

*1.) The game must work on most of the commonly used browsers (IE, Firefox, Chrome, Brave, etc.)*

*2.) There must be a choice to play with one or more teams.*

*3.) Game and team names must be unique and allow the player to check if their chosen name exists.*

*4.) Only one instance of the game can be running at one time.*

## [Design Constraints](#_2et92p0)

1.) We are targeting a browser-based architecture.

2.) The game and its rules already exist, so we will be using that as our design for the port.

3.) There must be a delayed rendering system that can fill in a picture slowly (as per the game rules when the timer is running).

4.) This will be a distributed system in the sense that it will involve a connection to Gaming Room’s web server from one or more computers.

## [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)

Looking at our UML Diagram (which is serving as our Domain Model) we can see that the outer layer shows a package called “com.gamingroom” which follows proper naming conventions for packages and tells us that all of our scripts for the game will exist in that folder. Next, we can begin looking at the various classes and properties in our program. First, we can see the “ProgramDriver” class which has one public static function main(). Main() will be where our code execution begins and we can see that this is the purpose of the ProgramDriver class, to drive the program by beginning execution of code.

We can see a black arrow pointing from our ProgramDriver class to a SingletonTester class. We can see that SingletonTester has one public function called testSingleton(), and as we can tell from the name, its purpose is to test our singleton and make sure that there can only be one instance of it. The black arrow going one way signifies an association, and that association is that our ProgramDriver class can call testSingleton(), this is further signified with the “uses” inside of the chevrons on the line – which is a “properties” marker, here we can see that the word uses shows us that ProgramDriver “uses” the class SingletonTester.

Next, we can see that we have an “Entity” class that is a parent to three other classes “Game”, “Team”, and “Player”. The arrow with the unfilled head shows us this parent-child relationship. Entity serves as sort of a generic “object” class for the Game that holds common fields and methods relevant to all objects. This architecture displays the use of the Object Oriented Programming (OOP) principle: Inheritance. Inheritance helps to extend Classes and their functionality as well as prevent duplicate code. Game, Team, and Player will inherit all of the methods and fields included in the Entity class. We can also observe another OOP principle there: Polymorphism. Polymorphism is a principle allowing objects to have more than one form in a program. Game objects are Games, but they are also Entities. Likewise with the Team and Player objects. They are both, and can be treated as either form by a calling function. If a function is looking for an Entity, a Game object can fill that role because it is both a Game and an Entity. If it is called as an Entity, it will be treated as an Entity, but it can also be called as a Game and treated as such through Polymorphism. Inheritance and Polymorphism are two principles that go hand-in-hand.

To briefly describe the classes that we just talked about, we can look over their fields and methods. Entity has two Private fields: id, and name, which are of types long and String respectively. These are set to private for protection purposes and we can see from Entity’s methods that it allows access to those fields in a controlled manner through the public accessor methods: getId(), and getName(). Here we can see another Object Oriented Principle being displayed: Encapsulation. Encapsulation is the idea that we should only expose necessary data, and otherwise keep it inaccessible. We did not expose Entity’s fields directly, we hid them with the private keyword, and only allow access to them, again, in a very controlled manner. This principle has a counterpart much like how Polymorphism and Inheritance are counterparts. Encapsulation is very closely related to the principle of Abstraction. Abstraction is a principle that says that complex details should be hidden and only a simplified interface to that complexity should be visible. An example of this would be our tv remotes: we press buttons, we know there’s some kind of signal being sent, but beyond that, most people don’t know the complex process that takes place once the button is pressed. They just know what each clearly labeled button does, that is their abstracted interface to the complexity underneath: the underlying hardware, firmware, and software inside a tv remote. This is good, because they don’t *need* to know. Knowing might only complicate the process. In much the same way Encapsulation can protect a program from unintended behavior by hiding information, Abstraction can protect a user from a frustrating or fruitless experience by hiding information, or the product from the user. We can observe the property of Abstraction with almost any public function. After all, the caller does not need to know the exact details of that function to call it, it just needs to be able to do so through a public interface. So, in our UML diagram, we could see this property in action through the public function toString(). We can see that all of our “Entity” classes implement their own overridden version of that method, and we can also notice that if we were to call any of these we would ostensibly receive some sort of String representation of that object. However, we do not know exactly how it is formatted, or need to know, to call toString on any of those objects therefore demonstrating Abstraction.

To finish off describing the architecture that we see in the UML diagram, we can see that Entity has a private constructor, ostensibly to ensure that an Entity object without arguments cannot be created. We can see that there is a public Entity constructor with two parameters: id and name, a long and a string, which is the constructor that would need to be used to instantiate an Entity. Similarly, has a public constructor with the same two parameters, but it does not follow the pattern of using a private default constructor. This is also the case with Game and Team. Additionally, Team has a private field called players, which is a list of Player objects and Game has a private field called teams, which is a list of Team objects. There is a public method in Team to add a player, and a public method in Game to add a team. We can see a line between Game, Team, Player, and even the GameService class that we have yet to describe. This line without an arrow signifies an Association, which means that these Objects or Classes are linked in some way, though it is not always specified how.

Before we move on to describe the fields and methods present in the GameService class, we can quickly cover the Multiplicity information present in the UML diagram. We can see at the bottom row of the diagram a series of “0...\*” texts in the middle of these Classes. These numbers signify Multiplicity, or how many/how few objects can be associated with another object or Class. The Multiplicity of “0...\*” signifies the quantity of 0-infinity, meaning that either 0 objects may be associated or infinite objects may be associated with the Class. So for our Gameservice class, we can see that either 0 or many Games can be associated with a GameService object. We can also see that either 0 or many Teams may be associated with a game. Finally, we can see that either 0 or many players can be associated with a team.

Finally, we can review the GameService class. This class has many private static fields, static being signified by the underlining in the diagram. Static means that only one instance of a related object or field should exist during the lifetime of a program. GameService has a private static list of Game objects called games, a private static long nextGameId, another private static long called nextPlayerId, and another private static long called nextTeamId. It also has a private static instance of GameService called service. Moving down to the methods section of the Class we can see there is a private default constructor for GameService, again, to prevent instantiations (only in this case ALL instantiations due to GameService being our Singleton). There is also a public static getter function for our Singleton called getInstance(). GameService also provides the following public functions: addGame(name), getGame(id), getGame(name), getGameCount(), getNextPlayerId(), getNextTeamId().

"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** | Similar to Windows, there is a high licensing fee involved in running a Mac server. Mac servers are generally more stable than Windows. That being said, Mac servers will not be able to run certain Microsoft apps. Hosting support exists for Mac at an average level and Mac offers a GUI for ease of use. Mac servers may even be the easiest to use out of all, with a fluid and intuitive layout provided for the server admin, and pre-installed server apps. Someone may not need as high of technical knowledge to run this server as others. Data losses are very rare.  ***There is an important caveat to consider with Mac servers, and that’s the fact that macOS server was discontinued in 2022. (I will continue analyzing it as if it wasn’t for the assignment)*** | Hosting a server on Linux is completely free. Linux may not be able to run certain applications, such as those made with .net frameworks or Microsoft products. Linux has a high level of security and stability out of the box due to its open-source nature which has led many Linux users to work on its vulnerabilities in the past. Additionally, Linux employs a segmented design which can protect the overall system from threats. Linux servers are very widely and commonly used, so the support for hosting is above average, but the system itself is more difficult to use than Mac or Windows due to running on a terminal. | Hosting a server on Windows can be costly since this OS has the highest licensing cost, but the server will be able to run a wide variety of applications. This ability to run applications is even more relevant if the software was developed using the .net framework. Windows servers will need additional measures taken for security because they are vulnerable to attacks. Hosting support exists for Windows servers at an average level. Windows systems are often less stable than their counterparts, but they are also generally easier to use and offer a GUI. | There may be some cost involved with implemented a mobile web server, but it should be cheaper than a Windows server. There are various Operating Systems involved with mobile devices and open-sourced software exists (such as i-jetty) that can turn a mobile device into a web server host. These solutions can scale but may not be able to offer the same performance or wide array of functionality that other server solutions can. Mobiles devices also have an inherent risk towards security breaches, and it may be hard to find support if needed. |
| **Client Side** | Developing software for a client-server connection on Mac, as well as the client-side app, should be fairly easy and inexpensive. There are far less options involved here than with Linux, Windows, or mobile devices. | Maintaining client-server connections with a Linux server and the development of said software is the cheapest using Linux. However, a higher level of expertise will be needed when developing this software and possible connecting the various free open-source Linux modules. If the administrator does not have the expertise needed for this system, they may incur additional costs by utilizing Linux support services. | The cost for a Windows server will not increase much with an increase in clients. There are many common frameworks for implementing client-server connections with Windows, so developing the software needed should not be hard. Only a minimal level of time and expertise should be needed. | A mobile web server with using i-jetty or similar software can support tens of thousands of connections. Cost will be relatively cheap but will increase with the amount of users. More expertise may be needed to facilitate these types of client connections than with other Operating Systems. More time may be required here than with a Windows or Mac solution. |
| **Development Tools** | Mac OS may be better for programming than Windows. It offers terminal with much more functionality. It also offers gestures which can speed up a workflow. Finally, it is much easier to build apps for multiple systems on Mac, as it provides a virtual Windows environment and related dev tools. Mac users can enjoy IDEs such as Visual Studio or PyCharm (and more). | Linux perhaps offers the best environment for developers. With a robust community and endless selection of tools and customizations, its easy to see why. Like Mac, Linux offers a more advanced terminal than Windows. Additionally, Linux can compile code faster. Linux supports all programming languages and IDEs. | Windows offers many different IDEs and dev tools. The variety is endless. However, it may be harder to build for certain platforms like iOS, due to the fact that there are no built-in tools targeting that like Mac has. Windows users can use IDEs like Eclipse or Visual Studio (and more). | Development will be much different here than other options because we are targeting a different kind of device. Extra attention will need to be paid to screen size and user experience. A touch-based interface, for instance, is much difference and less accurate than one with a keyboard/mouse. Mobile IDEs include programs like Android Studio or Visual studio (and more). |

## Recommendations

1. **Operating Platform**: I am recommending that we use the Linux Operating System for our server. This will save money as Linux is open-source and there are no licensing fees. There are optional services we can purchase if needed, such as technical support, which are still only a fraction of the cost of a Windows or Mac server. As Draw it or Lose it is a relatively small game, we won’t need large enterprise-level solutions. It would be better to have a customizable/flexible server that we can upgrade as needed rather than one that overshoots our needs. Additionally, there are many open source components and modules freely available for Linux that can be easily adapted for use in Draw it or Lose it. Finally, Linux servers are highly compatible, which makes it perfect for our system because we will be servicing many different browsers and client-side Operating Systems from our player base.
2. **Operating Systems Architectures**: Linux systems are highly customizable, and I suggest that we customize a solution specific for the game. Most Linux systems consist of the following components: a Kernel, a System Library, Hardware, System Utilities, and a Shell. These components are distributed amongst the following layers (from lowest to highest): Hardware, Kernel, Shell, and Applications. There are different types of kernels, and I suggest that we choose a Monolithic Kernel. This type should serve the game well due to the fact that it allows the Kernel and User to share the same memory space which promotes faster process execution. The main drawback to a Monolithic Kernel is its large size, which is completely mitigated by the fact that we don’t have to store much data on the server (~2gb). The Kernel will manage resources (our image set) and I/O (requests and services from our host). It will also abstract hardware details and manage memory. We should take advantage of Linux System Libraries (functions for apps, OS functionality, file manipulation, networking, and memory management) and the Linux System Utilities (a set of programs that perform system-level tasks). I suggest that we skip a GUI for the server and instead access it through a separate workstation to save space.
3. **Storage Management**: On our bottom-most layer is the hardware for the server, and I recommend we include one 500gb SSD (solid-state drive) for storage. This should cover all of our needs for the game and because they are so cheap, it shouldn’t be a problem to add another drive later if needed. A SSD will be better in almost every way compared to HDDs (which can be cheaper and longer-lasting) since we are serving many users with a game and we don’t want any lag or slow access times.
4. **Memory Management**: To manage memory we will keep a list of all 200 images in cache memory on the Server. This would be reset if the servers need to go down for maintenance or updates. Images will be accessed from this list by users in random order to facilitate the game features and function. A client will have one pointer reference at a time which will always point to the address of one image at a time (for each round of the game) and this pointer will be updated if a new game is started (and checked to make sure it’s not the same address as before.) We will use a demand paging system, which is compatible with Linux, to store each image with a standard page size of 2mb.
5. **Distributed Systems and Networks**:   
   Draw it or Lose it will be compatible with all systems due to the fact that it is written primarily in Java with HTTP for web-based functionality. The distributed system will consist of one host server and multiple clients. A client will first try to establish a connection to the Server by connecting to one if its open ports. When successful clients will make requests to the server to play the game ranging from initializing the application, logging in, starting games, and the other various functions involved like adding players and teams. The server will respond to these requests and if they are valid will service the clients allowing them to play the game. The game will work in any browser due to the programming languages involved in the code and any Operating System should be able to access it. The server will be hosted on a WWW domain that Users can connect to by typing in the internet address for the game. We should have a backup generator on site and a mechanism for the server to switch to backup power in case of a power outage.
6. **Security**: I recommend using a username/password system for logging in and require a strong password to prevent brute force attacks. We will then protect user information through the use of Token-based authentication. We don’t want a User’s username and password to be transmitted often, this would be a security risk as a malicious actor could intercept these transmissions, possibly leading to a breach of the User’s more sensitive accounts. Instead we will transmit valid-tokens. We will use local storage for the tokens with an AES encryption scheme to mitigate XSS attacks. Finally, we will set expiration dates for the tokens to further reduce our security risks.