

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template**](#_l6ti7uoag22u)1

[**Table of Contents**](#_30j0zll)2

[**Document Revision History**](#_grjogdjh5fi8)2

[**Executive Summary**](#_sbfa50wo7nsh)3

[**Design Constraints**](#_2et92p0)3

[**System Architecture View**](#_ilbxbyevv6b6)3

[**Domain Model**](#_8h2ehzxfam4o)3

[**Evaluation**](#_2o15spng8stw)3

[**Recommendations**](#_m8aleynsvzvc)5

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/16/2021 | Jared Hodgkins | Outline the changes that need to be made to the current implementation of the game to better encapsulate data, and provide a managed, single source of data for handling object uniqueness.  Express the design concerns for both the client and server applications across multiple operating systems and platforms. |
| 1.1 | 10/17/2021 | Jared Hodgkins | Updated recommendations section to more accurately address the characteristics and techniques in regards to various systems architectures. |

## [Executive Summary](#_sbfa50wo7nsh)

In its current state, the architecture for Draw It or Lose It has repeated code in multiple classes that could benefit from a class-inheritance refactor with a base class that contains common attributes and functionality. I am proposing a base class “Entity” that stores an id and a name that Game, Team, and Player objects will inherit from as they all need an id and a name. The project is currently built so that a single GameService controls all the games, teams and players, but it needs to provide an interface for handling unique ids. To address this issue, we will add fields to store the next available id for each of these classes, and public interfaces for these classes to retrieve the next class-appropriate available id when creating new objects while updating the next id at the same time. It’s important that games, teams, and players have unique ids to keep track of the individual entities, or it could lead to serious bugs and unexpected behavior for the players of the game. Our central singleton GameService object is the perfect place to provide this functionality as we can be sure that the data is consistent and coming from a single source.

## [Design Constraints](#_2et92p0)

Developing the game for a web experience poses its own unique challenges. The largest and most obvious challenge is that web-browsers aren’t as performant with handling several fast animations and graphics rendering compared to native desktop or mobile applications. Drawing to the screen and optimizing this process will require a different system than currently implemented in the Android application. Another major factor is that there are multiple browsers from different vendors that don’t always behave the same when rendering content. In order to provide the same experience for most or all users, we will need to add additional time for testing and software iteration on various browsers throughout the software development lifecycle.

## [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)

Below is our object model for Draw It or Lose It. In the top left corner, we have our main application entry point that will be responsible for running the application, creating our GameService and bootstrapping the service that will provide functionality for individual games to take place. The SingletonTester is a class with a sole purpose of making sure that our GameService follows the singleton pattern, and more specifically that only one instance of GameService is allowed to exist.

The Entity class is a base class that provides common functionality to classes that inherit or extend from it. Namely this functionality here is encapsulating and storing values for an entity’s id and name. This class provides public interfaces for getting the id, name, and additionally a toString() method that returns a human-readable formatted string containing these data values.

From the Entity class, we have Game, Team, and Player that all inherit, or extend from Entity. These classes can all be considered entities but have unique data that distinguishes them from one another even though they are still similar. In this structure, the Game class is associated with the Team class, and the Team class is associated with the Player class. A game can have 0 or more teams, and a team can have 0 or more players, which are managed in List objects for each class respectively. The Game class provides a public interface for adding a team, while the Team class provides a public interface for adding a player. Each of these three classes override the toString() method provided from their base Entity class to present their data in specialized human-readable formatted strings.

Lastly, we have the GameService class which is providing most of the functionality in our object model. This class is associated with the Game class and can have 0 or more games, which will be managed in the private static games List object. It also keeps track of the next available id for games, teams, and players in respective private static variables. The constructor for this class is marked as private so no other classes can create instances of it, and instead provides a public interface getInstance() that will create an instance of GameService if one does not yet exist and then store this instance in the private static variable named service before returning it to the caller. If an instance however already resides in the service variable, then it will instead simply return the instance stored there. GameService also provides public interfaces for adding a game by using a game’s name, getting a game by either the game’s id or game’s name, and another for getting the total number of current games. Finally, this class provides public interfaces getNextPlayerId and getNextTeamID that will return the current available id for players and teams and then increment the values within their respective private static variables as well in preparation for their next call.

**"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** | The main disadvantage to using a mac for this server is the cost associated with it compared to a Linux or Windows machine. The hardware itself is much more expensive than running a server on a Windows or Linux machine. Macs have excellent developer support and tooling for web development and running performant server applications. | Linux is an excellent choice for running a server application, as the cost of the operating system itself is typically none. Linux is used to power most server-side applications as it is inexpensive, reliable, and boasts great uptime. The downside to using Linux would be making sure that the server runs on multiple distributions and avoiding inconsistencies in libraries between versions. | Windows is another good choice for running server-applications depending on the version of Windows. Typically, the cost of the hardware and software is less than mac, but more than Linux. Windows applications don’t typically experience the library inconsistencies that Linux does. Windows is the target of more viruses and malware however which should be considered during development and throughout a product’s life. | While theoretically possible to host a server from a mobile device, these devices typically suffer from lack of resources required to provide a persistent service. CPU and memory is more limited and thermal output from running a service 24/7 will cause the device to rapidly deteriorate. The cost associated would be higher as well for specs that would rival desktop counterparts. |
| **Client Side** | Since we are porting from Android, we are probably less likely to have mac developers specifically that are familiar with Android’s Java or Kotlin source code. That isn’t to say there aren’t Android developers on mac, because Android Studio runs fine on this platform, but most mac developers are likely using xcode with Swift or objective-C. If there are no macs on premise to test with, a solution would be to utilize a “mac in the cloud” service to gain access to a mac remotely for the purposes of testing. | Android development on Linux is common, so many Android developers will likely also be familiar enough with Linux. It is usually enough to have one engineer or QA experienced with Linux to assess loading the application in each of the most popular browsers even without Linux experienced developers. | Android development is common on Windows as well, but developers may be more familiar with Microsoft’s line of products such as .NET and ASP. | Web browsers on mobile devices typically lack features and certain standards compliance that desktop browsers have an advantage with, making developing a client for mobile browsers a more time-consuming process. Mobile browsers also tend to run less performant than desktop counterparts due to the hardware specifications in these devices. Multiple browsers and devices will need to be tested to guarantee consistent functionality and operation. |
| **Development Tools** | Android development is done using Android Studio, web development isn’t as specific, but Visual Studio Code, Sublime Text, and Atom are all highly popular choices. Vi and Vim might also be chosen by a niche group of developers. The server and client would likely be developed using JavaScript, leveraging some framework or library such as Angular, React, or Vue to name a few. The server could be built with nodejs, as a python application, or potentially as a Java or Swift application. A native mac server application would be built in xcode, while a Java application could be built using Eclipse for instance. | Android source development is typically done using Android Studio. As with the mac, common choices today for web development are Visual Studio Code, and Atom. It would also be more common for Linux developers to choose vi or vim. As with mac, JavaScript could be utilized for the game client using Angular, React, or Vue. The server could potentially be either a nodejs, python, or Java application, but might also be additionally likely to be developed in C/C++ on this platform specifically. | Android Studio once again for Android development. Eclipse is a very common IDE bundled with Android Studio for Java/Kotlin development. In the web world we have Visual Studio, Visual Studio Code, Atom, Notepad++ to name a few. Visual Studio Code is very popular for web development across all platforms. The server application would mimic the Linux choices but might also include C# or .NET inside Visual Studio. Eclipse could also be used to write the application in Java. | Development wouldn’t be typically taking place on a mobile device, but since the client is a web application, the deployment for the client would be the same as all other platforms. The browser would need to support JavaScript and HTML5. A server application written for Android would need to be developed using Java or Kotlin. Development is possible from an Android device, but not as streamlined as from a desktop. If developing a server application for iOS, a mac would be needed, and it would have to be written in either Swift or Objective-C using XCode. |

## 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 that the server for Draw It or Lose it game is hosted with a Linux distribution. The main reasoning for this is because Linux is already being used to power over two-thirds of all web servers. It boasts great performance, high security, amazing uptime, and best of all, it’s free! We will be able to leverage all the great features of Linux and meet our client’s budget at the same time by eliminating additional licensing costs. The Java server we will develop for the game will compile and run without issues on Linux, as it would on Windows or Mac, but the main advantage here is cost (or lack of cost) associated with it. The server will provide a sockets-based connection and REST API that clients on multiple platforms can interface with in a uniform manner.
2. **Operating Systems Architectures**: Keeping in mind that we have deliberately chosen to have players utilize a REST API to load images on demand in order to eliminate the need for them to store a large database of images on their devices, we should go with an on-premises solution. Sending high resolution images to clients repeatedly will utilize a lot of bandwidth and resources and doing this will not be cost effective when choosing a cloud solution as we would typically be charged per gigabyte of network traffic. With an on-premises server, we can take advantage of a much higher, if not unlimited data cap which allows us to keep our monthly operating costs low. The only exception to this would be if we found a cloud solution that offered a high enough data cap or unlimited plan and was priced to compete with an on-site internet plan and general server upkeep.
3. **Storage Management**: Because Draw It or Lose it is relatively simple in nature and will not require a lot of user data to be stored, we should take advantage of microservices to store our user data in a SQL database and to perform user authentication to the server. These services can then send authorization tokens to our main game server and to the clients to carry out authenticated game-specific functionality. This will take away the burden of managing our own secure database and takes advantage of using a low cost microservice to perform only the lightweight operations we need them to in order to make our application work successfully. By using a microservice we also mitigate the possibilities of a distributed denial of service attack on our user data and user authorization system.
4. **Memory Management**: The Linux operating platform utilizes a swap space, which is a section of the hard disk that is used for caching and swapping in/out memory in RAM that isn’t needed now but might need to be called in again shortly. This will help the operating system run at high performance and help protect us from locking up system memory that we don’t need for the current task(s) being processed. Regarding the server application from a development standpoint, we will need to allocate enough system memory to handle all the server’s functional code, all the game’s data, and the tokenized player authentication. Additionally, our server will need to allocate enough memory to handle upwards of the entire image database in a worst-case scenario, as the game clients will be requesting the image data from our server, so loading the image data into memory could be a costly action across hundreds of instances of games. For this purpose, we should make sure that there is at least 2 Gigabytes of system memory available solely for having the entire collection of images loaded so that they can be quickly served through client requests. Swap space will come in handy when not all 1.6 GB are loaded and can potentially earn our server some small gains when this is the case. On the client end however we will be taking advantage of a much lower memory footprint as stated above and will only require the player clients to allocate somewhere between 8-32 Megabytes for the images in comparison.
5. **Distributed Systems and Networks**: Draw It or Lose It’s multiplatform specifics will be handled by providing a universal REST API for the clients to communicate with the server. Using an API in this manner eliminates any platform specific requirements aside from implementing a compatible and suitable HTTP client. By using dropwizard with our Java server application we can quickly build a REST API that exposes both public endpoints as well as endpoints that require various levels of authentication. We will be always relying on stable internet connections between clients and server, as this game will demand connectivity. Our game will establish socket connections between the server application and the client applications to allow for duplex data transmissions to occur while a connection is open. The server application will mainly be responsible for managing the connection state(s) and pushing updated data from individual clients to only the clients that must be informed of the updates, such as when a player has made a guess or when a new image is to be rendered. To increase uptime on the server, we could implement a distributed server application that takes advantage of RPCs to run multiple instances of the server at different locations, with only a single instance handling client/server connection at any time. We could store a relatively lightweight snapshot of the current server’s state in a database table at set intervals so that if at any point in time there is a drop in connection from the master server, a backup server could be waiting to take over as the master and then transfer the server’s state and connection data from this snapshot and then update the microservices about the change in server IP address to push out to the clients. If this happens, the clients will then connect to the new server reported by the microservices and resume where they had left off nearly seamlessly.
6. **Security**: Any sort of stored user data will be cryptographically hashed and stored inside of a restricted database. This will provide two layers of protection from unauthorized data access, as any unauthorized user would need to know how to both access the database information, and the proper keys to decrypt the data. Because we have chosen to use microservices for our user database and authentication we will need to select one that has a proven track record of high security measures to protect our player’s data. By offloading our player data and authentication to microservices, we also make it harder for malicious users to track the IP address that the data is coming from. With the socket connections a user could trace the IP to our game server, but even if they manage to get into our game server, we will not be storing any player information there or in the server’s memory.