

Draw It or Lose It

# **CS 230 Project Software Design**

Version 1.2

## 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 | 05/23/20 | Zaq Hoffman | Original Creation of *Draw It or Lose It* environment |
| 1.1 | 06/20/20 | Zaq Hoffman | Addition of Platform Evaluation |
| 1.2 | 06/21/20 | Zaq Hoffman | Addition of Recommendations |

**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, *The Gaming Room,* wants to develop a web-based version of their Android game, *Draw It or Lose It*. The game in question consists of the game service capable of running multiple game instances, where games can have multiple teams, and teams can have multiple players. Games, teams, and players should each have unique names and have unique identifiers to minimize internal bookkeeping errors such as a player participating on multiple teams, a single team in multiple games, etc. I propose we use the Singleton Pattern for the game service to maintain a single instance of the service in memory and use the Iterator Pattern to track the unique instances of games, teams, and players in memory. Additionally, the game, team, and player objects share similar traits and functions, so these objects could inherit common traits and functions for greater efficiency on a resource-limited web-based environment.

## [Design Constraints](#_2et92p0)

Singleton Pattern GameService –

A single static instance of a GameService object will allow for centralized bookkeeping of all other objects. This constraint mitigates issues of memory leakage and keeps the design simple yet effective for web-based environments.

Inherited Features -

Though at face value the Game, Team, and Player objects appear to have varying functionality, the operations to create and track these entities in memory are similar. For simplicity and efficiency, these objects can inherit from a parent object, Entity, all the shared variables and methods, such as the name and Id variables. Any additional individual functionality can be added without major refactoring.

Iterator Pattern Game, Team, Player –

Since there will exist multiple Games, Teams, and Players, and all these should be unique objects in memory, a simple iterator pattern to manipulate and search by inherited features, the name and id variables, will streamline object tracking in memory, and allow users to have unique team and players names.

## [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)

This UML diagram indicates generally good programming practices, such as using scope to hide underlying systems for stability and safety, and compartmentalization for modularity, as well as common object-oriented principles like inheritance and the Singleton and Iterator design patterns. The ProgramDriver class initializes and runs the GameService class, and SingletonTester class. GameService acts as the controller for all the other objects and follows the singleton pattern by creating a single private static instance of itself. The GameSerivce instance has an association (a list member variable) with zero to many Game objects, which each have an association (a list member variable) with zero to many Team objects, which each have an association (a list member variable) with zero to many Player objects. The Game, Team, and Player classes inherit common attributes and methods, such as id, name, and accessors, from a parent Entity class. These classes also use accessors in GameService to extract and assign static id variables and name variables in to maintain unique instance identification, following the iterator design pattern.

****

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | +Lower setup (if MacOS familiar)  +Higher security (due to less leaky proprietaries)  =Moderate price (due to less leaky proprietaries)  -Lower fine control (most functionality comes built-in so now tweaking)  -Lower portability (barring websites, Mac servers cater to Mac clients) | +Lower price (open source software, variety of hardware)  +Higher fine control (customizable for any needs)  +Higher portability (caters to nearly all clients)  +Higher security (addable extensive security)  - Higher setup (lots of plugs to be plug-and-play) | =Moderate security (greater target by malicious agents)  -Higher setup (many configurations)  =Moderate control (more features than mac)  =Moderate Portability  (caters to many clients)  =Moderate price (varies on size) | Hosting web-software on a normal large scale serve-client model is not viable due to limited resources. P2P, distributed server, or other small-scale models may be viable but is not the focus of this evaluation. |
| **Client Side** | There are fewer configurations, fewer security concerns so fewer overall considerations to develop for Mac but requires specific Mac expertise and licensing. However, fewer users have Mac than Windows so return on investment (ROI) will be lower. | Like Mac, linux requires specific expertise, and holds less market share of user machines, so ROI is lower, but also requires more development time and cost for many configurations. | Windows is the most widely used computer, so has high ROI. Windows has many configurations so requires more cost and time, but lower expertise since its more standard platform. | There are limited configurations for mobile devices as compared to computers. UI design is unique on mobile and requires specific expertise. Differences in mobile platforms (windows vs apple vs android) stack with mobile concerns. High ROI since most people have mobile devices and use them for apps. |
| **Development Tools** | Native Lang: Objective C and C  Web Lang: HTML, C++ JavaScript, CSS  IDE: Xcode - not free  Machine: Mac  Added Licensing – Apple developer program | Native Lang: C  Web Lang: HTML, JavaScript, CSS  IDE: free such as Eclipse or Netbeans  Machine – Any | Native Lang: C, C++  Web Lang: HTML, C++ JavaScript, CSS  IDE: free such as Eclipse or Netbeans  Machine - Any | Native Lang: C, C++, Java  App Lang: HTML, C++, C, JavaScript, CSS, Java  IDE: free such as Eclipse or Netbeans  Machine - Any |

## 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**: The best operating platform to host the *Draw It or Lose It* application will most likely be Linux or Windows. A Mac OS server will only or mainly cater to mac users, and that only represents a small fraction of the player base. If aimed at expansion of users, using a Linux or Windows server will be optimal because of its compatibility with many other platforms., including Windows, Linux, Mac, and Android. While it may require slightly more technical know how to setup, one of these OS will grant the company more features, expandability, and control than using a Mac OS server.
2. **Operating Systems Architectures**: The architecture of the OS should most likely run in a Client-Server and Layered architecture pattern. Hosting the application on a server or servers centralizes the synchronization events present in the time-sensitive application. It also limits the amount of resources necessary to run the client-side application. A layered approach provides abstraction and allows the application to interact with the clients’ devices safely and easily. Layering also separates the functionality from other systems such as user interface, which may differ across platforms, and so needs specific tailoring.
3. **Storage Management**: Being a web-based application, most of the data for the application should be stored centrally on the server side. For most client systems, it will be sufficient to only locally store temporary files holding data received from the server, pulling them as needed from the servers and accessing them locally for efficiency. This will limit server traffic from multiple requests for the same data (for example, the same image file used multiple times in game) and will not be too intrusive on client systems.
4. **Memory Management**: On the server side, the memory should mainly be used to load and send application data for clients, like image files, synchronization information, etc. Theses interactions will be numerous and frequent, and its important that server-side memory usage is limited to these events and not much more. Additional functionality comes from the memory of clients’ systems. Though use of client memory should also be limited, taking the work such as UI, rendering, etc. is necessary for efficient use of resources across the network. Care should be taken to include individual optimization considerations for the client systems OS. Since all systems handle their memory slightly differently, the same implementation of memory management will not be optimized for every client computer.
5. **Distributed Systems and Networks**: Distributed servers present many strengths to the expandability and access of an application, but also many considerations for effective use. A web application is certainly a good approach, but the question arises how best to implement it. Physically owned servers are the most straightforward, but then you would have to run and own servers in multiple locations, and synchronize them constantly, not to mention it requires extensive expertise to upkeep and maintain physical servers. And when a system like this goes down, you better have a backup and a contingency plan or you’re going to have a bad time. The use of cloud architecture allows a simple way to get all the power of running your own servers without the hassles of physical servers. Cloud servers are easily expandable, maintained by 3rd parties, can be setup virtually anywhere, and offer multiple contingencies for outages.
6. **Security**: Security is of the utmost importance, and should be baked into the application, not as an afterthought. A username and password, or some other authentication system, stand as the first line of defense for limiting users from admin level access and securing individual user accounts. The application should also have some form of anonymity, never revealing names, ages, locations, or IP addresses of users, in fact this kind of information should be used in a very limited capacity and if it must be used, it should be encrypted. Finally, servers should have security measures in place to keep the limited sensitive information safe from malicious agents.