

***Draw It or Lose It***

# **CS 230 Project Software Design Template**

Version 1.3

## 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)4

[**Domain Model**](#_8h2ehzxfam4o)4

[**Evaluation**](#_2o15spng8stw)5

[**Recommendations**](#_m8aleynsvzvc)8

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.3 | 10/25/2020 | Bryan Dugan | **Update: Added final 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)

Our client, The Gaming Room, has a successful Android-based game app called *Draw It or Lose It*, in which multi-player teams compete to identify gradually-rendered drawings within a time limit (30 seconds for initial attempts, 15 seconds for opponent rebuttal attempts).

The Gaming Room now intend to develop a web-based, multi-platform version of the app. In order for *Draw It or Lose It* to function as required, the app must allow only a single, unique instance of its game service to exist at any given time. Similarly, game, team, and player names associated with the game service must also be unique.

In order to address the requirement that the app be multi-platform, the app is coded using Java. Java’s JVM allows for platform independence, providing the advantage of portability between disparate systems.

The app will implement a Singleton design pattern in order to ensure that no duplicate instance of the game service can be created while another service exists.

Iterator patterns implemented on lists of existing games, teams, and players will ensure that only uniquely-named instances of each class can be constructed. The iterator will check for existing instances that already match a name chosen for a new game, team, or player, and only create a new instance if an instance with that name does not already exist.

Counters will ensure that a unique ID is generated for each new instance of the aforementioned classes.

## [Design Constraints](#_2et92p0)

According to Imed Bouchrika, heterogeneity is one of the major challenges in implementing and administering a distributed system (Bouchrika, 2013). The app must function at the same standard of performance despite being available to users with different devices, operating systems, and networking capabilities and requirements. As the app is maintained and upgraded, it may need to interact with other, non-Java web apps and services requiring integration with code written in other programming languages. As Bouchrika (2013) writes, the challenges of heterogeneity also apply to the differing roles and areas of aptitude among teams involved on a project.

The need for scalability as the app grows its user base, will be a catalyst in changing back-end demands to keep the app running despite the increased load. In a server-based set-up, this would require continuous maintenance and installation of servers and other hardware to meet growing demand. Should a cloud-based setup meet requirements with reduced demands for physical infrastructure locally, security, system monitoring, and maintenance testing will remain critical concerns.

The degree to which the above challenges may be overcome will depend on continuous, iterative assessment, planning, testing, development, and adaptation between sprints. Careful, repeated consideration of both what is do-able and what is ideal (not necessarily the same!) will – as always – be crucial to overcoming constraints on design.

## [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)

A parent class – Entity – will provide the basic functionality for the Game, Team, and Player child classes that extend it. Each GameService instance has, as a member, a private list of games. Each Game instance has, as a member, a private list of teams. Each Team instance, as a member, has a private list of players. “Add” methods for Game, Team, and Player iteratively check these lists to ensure no duplicate names are created, and that unique ID numbers are assigned for each new instance. Each of the three child classes contains an override toString() method. The ProgramDriver tests for functionality of all classes, and invokes the SingletonTester to guarantee only one GameService instance.

Encapsulation is demonstrated by private members and methods – including the private default constructor for GameService. Abstraction is achieved through the methods for adding new Game, Team, and Player instances: the user selects a name for the new instance when calling the appropriate add method, but has no awareness of any lists, iterators, ID number counter variables, etc. “under the hood.” The add method either creates a new instance or returns information about an existing instance that already has the chosen name. Inheritance is evident in the members and methods shared by Entity and its child classes. Polymorphism is evident in the overridden method - toString() – inherited, but made unique for each class.

****

## 

## 

**Evaluation:**

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Macs are justifiably known for system stability, ease of maintenance, and reduced vulnerability to malware versus Windows. Macs can also run Windows or Linux well, using high-performing virtual machines, if need be. This can make a Mac a great platform for hosting a web-based application. On the other hand, the strict licensing and varying degrees of support provided to third parties for macOS infrastructure and development tools can present significant – and potentially insurmountable – time and project-level constraints. Overhead costs are higher than for Windows or Linux. | One of the greatest strengths of Linux is its open source nature. This allows hosts utilizing a Linux platform for their web-based applications a great deal of flexibility, in terms of apps and services available for server-side operations. It also allows for lower-level customization that can significantly increase security, and ensure ongoing compatibility. This customization places significant agency – and responsibility – in the hands of developers utilizing a Linux-based server. | Windows is robustly-supported by Microsoft and a sizable developer community. Since so many existing systems run on Windows, and it is the most familiar operating system for most users, there can be less training overhead required versus less common operating platforms. Unlike Linux, Windows is neither free nor open source. Windows is ubiquitous as the operating system for much of the world’s computing needs. As such, a high percentage of malware is written with Windows architecture and vulnerabilities in mind. | In general, mobile devices lack the hardware and networking muscle to physically host web-based applications. They can, however, run applications that provide administrative and maintenance capabilities for server and cloud-based services and applications.  Third party issues with iOS are similar to those discussed for Macs. |
| **Client Side** | As with server side, overhead and licensing issues can affect the ability to actually provide support for client-side operations. This overhead is time and money. Expertise may be lacking on how to use Mac-based software.  Java and its JVM will help ensure that the game app works for diverse clients. | Lower financial overhead may give way to significant overhead in time (and therefore, mounting costs) committed to training team members unfamiliar with the Linux environment, and its key utilities and apps.  Java and its JVM will help ensure that the game app works for diverse clients. | Issues impacting the server side are reflected in potential challenges for client-side operations. Vulnerabilities unique or more acute in the case of Windows increase the risks of harm from malware. This means more time and money committed to security. Users with malware compromised systems will present a constant threat.  Java and its JVM will help ensure that the game app works for diverse clients. | Front-end developers will need to remain mindful of the app’s presentation on mobile devices. This will require additional time and resource investment, as testing the mobile experience will remain its own related-but-separate area of focus within the app. From a performance perspective, it will be critical to remain mindful of users with minimum-performance mobile hardware. For iPhones, the idiosyncrasies of Apple and iOS will remain a potential stumbling block.  Java and its JVM will help ensure that the game app works for diverse clients. |
| **Development Tools** | Languages:  Java  Swift  Objective-C  C++  IDEs:  Eclipse  Xcode  IntelliJ  Others:  Homebrew (package manager)  Apple Developer Tools  Licensing costs will be higher. Proprietary tools place more specific demands on team knowledge base. | Languages:  C++  Java  Python  IDEs:  Brackets  Eclipse  Various third-party developer tools and utilities. Significant potential for wide variety of tools and customized architecture without significant overhead. | Languages:  C  C#  C++  .NET Framework  System libraries for various languages  IDEs:  Eclipse  Netbeans  Visual Studio  IntelliJ  Various native Windows and third-party apps and utilities. Hosting can be more costly, with higher overhead, than with Linux – with higher cost-for-performance and less flexibility in customization. | Languages:  C  C#  C++  Objective-C (iPhone)  Python  Swift (iPhone)  IDEs:  Eclipse  Visual Studio  Third party apps and utilities through App Store / Play Store (more plentiful for Android devices). A separate-but-connected development team may be ideal for ensuring specific focus on mobile design, implementation, and maintenance. Doman knowledge mobile architecture and principles of effective design is a must on the development team. |

**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**: Mindful of performance, security, stability, and scalability without significant recurring costs, I would recommend Linux-based hosting platform. I have had significant success working with Ubuntu in creating and managing RESTful APIS. Mac may ultimately be preferred depending on ease of training and implementation with current team and knowledge base. In either case, the app will be coded in a platform-independent programming language.
2. **Operating Systems Architectures**: The Linux kernel lies at the heart of the operating system, acting as a sort of wrapper, gatekeeper and messenger for the hardware. The kernel is further abstracted from the user side via the Linux shell interface, which is used to manage and run applications. As Linus is open source, there are also a wide variety of GUI set-ups available in different Linux implementations.
3. **Storage Management**: Linux implements LVM – Logical Volume Management. According to David Both (2013): “LVM allows for very flexible disk space management. It provides features like the ability to add disk space to a logical volume and its filesystem while that filesystem is mounted and active and it allows for the collection of multiple physical hard drives and partitions into a single volume group which can then be divided into logical volumes.”
4. **Memory Management**: Linux uses demand paging and copy on write (COW) – which ensures that identical child and parent processes sharing the same memory will first make a copy of shared page, then alter the copy. This also ensures that the originally shared pages remain untouched.
5. **Distributed Systems and Networks**: Java libraries, as well as system and third-party Linux apps and utilities, provide a solid base of support for network interactions utilizing Websockets protocols. Websockets are ideal for the frequent updating of both client and host using full duplex communication – perfect for a gaming service. Timeout and connectivity issues for mobile users will require greater, specific focus under a Websockets implementation, but the overall performance advantages over HTTP are significant. Python’s Pymongo library and several frameworks – Bottle, Flask, Django – offer robust ready-to-go solutions for server-side administration. All run very well in Linux.
6. **Security**: Software-level security begins with OOP principles in practice: abstraction of lower-level behavior, inherited private members, private constructors called indirectly via a program driver, unique identifiers assigned and verification of unique names ensured before creation of class instances. As discussed above, Linux’s open source architecture offers significant latitude for security-enhancing customizations. On top of Linux’s already more secure and less-specifically-targeted architecture, administrators can install and run utilities to enhance security.

**Works Cited**

Both, D. (2016). “A Linux user’s guide to Logical Volume Management”. Opensource.Com.

https://opensource.com/business/16/9/linux-users-guide-lvm. Accessed: 27 September 2020.

Bouchrika, I. (2013). “Challenges for a Distributed System”.

https://www.ejbtutorial.com/distributed-systems/challenges-for-a-distributed-system.

Accessed: 27 September 2020.