

<Draw It or Lose It> CS 230 Project Software Design Template Version 1.0   \*\*Table of Contents\*\*

CS 230 Project Software Design Template 1  
Table of Contents 2  
Document Revision History 2  
Executive Summary 3  
Requirements 3  
Design Constraints 3  
System Architecture View 3  
Domain Model 4  
Evaluation 4  
Recommendations 5

**Document Revision History**

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/22/2023 | [Your Name] | Initial draft of the Software Design Document for Project One. |
| 1.1 | 10/15/2023 | [Your Name] | Added Evaluation section for Project Two. |
| 2.0 | 11/10/2023 | [Your Name] | Finalized Recommendations section for Project Three; updated Executive Summary. |

**Executive Summary**

This document outlines the software design for expanding The Gaming Room's "Draw It or Lose It" game from a single-platform Android application to a multi-platform, web-enabled game service. The solution addresses the client's need for cross-platform compatibility, real-time communication, scalable architecture, and robust security. The recommended approach involves developing native clients for target operating systems (Android, iOS, Windows, macOS) that all connect to a centralized, Linux-based game server hosted in a cloud environment. This design ensures a consistent user experience, efficient resource management, and a secure foundation for future growth.

**Requirements**

*Business Requirements:*

* Expand the game to multiple platforms (iOS, Windows, Mac, Web) to reach a wider audience.
* Maintain a consistent user experience and game state across all platforms.
* Implement a subscription model to generate recurring revenue.
* Protect intellectual property and user data.

*Technical Requirements:*

* The game must support multiple teams and simultaneous players in a single game session.
* The application must be able to run 24/7 with minimal downtime.
* The system must securely handle user authentication and profile data.
* The architecture must be scalable to handle a growing number of players and game instances.
* The game must synchronize in real-time across all connected clients.

**Design Constraints**

The primary design constraint is developing for a **web-based distributed environment**. This requires the application to be platform-agnostic on the server-side and to use standardized communication protocols (like HTTP and WebSockets) to interact with diverse clients. The implications include:

* **Network Dependency:** The game's core functionality requires a stable internet connection, impacting design decisions for handling latency and offline scenarios.
* **Security:** Data transmitted over the network must be encrypted, and the server must be hardened against attacks.
* **State Management:** Game state must be managed centrally on the server to maintain consistency across all clients, which adds complexity to the server architecture but is essential for a fair and synchronized experience.

**System Architecture View**

The system will follow a client-server architecture. The server, hosted on a Linux platform in the cloud, acts as the central authority for game logic, state, and user data. Client applications on various devices communicate with the server via RESTful APIs for administrative tasks (login, team creation) and WebSockets for real-time game events (drawing, guessing, timers). This logical separation allows for robust, scalable, and secure multi-platform support.

**Domain Model**

The provided UML class diagram illustrates the core objects in the "Draw It or Lose It" system. Key classes include:

* **GameService:** A singleton that manages active games and acts as the central controller.
* **Game:** Contains the game state, including the current round, teams, and the target word.
* **Team:** Comprises multiple **Player** objects. A team has a score and a designated drawer for each round.
* **Player:** Represents a user, with attributes like a unique identifier and username.

The relationships demonstrate object-oriented principles such as:

* **Encapsulation:** Each class contains its relevant data and methods (e.g., the Game class manages its own list of Team objects).
* **Composition:** A Game is composed of Team objects, and a Team is composed of Player objects, meaning these parts cannot exist without their whole.
* **Singleton Pattern:** The GameService ensures a single point of access for managing games, which efficiently fulfills the requirement for a centralized game manager.

**Evaluation**

| Development Requirements | Mac | Linux | Windows | Mobile Devices |
| --- | --- | --- | --- | --- |
| **Server Side** | macOS Server is a stable Unix-based platform but is niche in the web hosting market, with higher hardware costs and less community support compared to Linux. It is not recommended for a cost-effective, scalable web service. | **Linux is the industry standard for web servers.** It is open-source, highly stable, and offers superior performance and customization. Vast community support and compatibility with cloud platforms make it the ideal choice. | Windows Server is a robust, user-friendly platform with excellent support for .NET applications. However, it involves significant licensing costs and is generally less performant for high-concurrency, non-.NET workloads compared to Linux. | Mobile operating systems like Android and iOS are not designed to host scalable, multi-user backend services. They are client-side platforms. |
| **Client Side** | Developing a native Mac client requires expertise in Swift or Objective-C and the macOS SDK. This adds to development time and cost but provides the best performance and user experience on Apple desktops. | A desktop Linux client would be a lower priority due to a smaller market share. It could be developed using cross-platform tools like Java or Qt, but the primary focus should be on web, mobile, and Windows. | The Windows client has a large potential user base. Development would require C# and .NET or C++ and the Win32 API, representing a significant but necessary investment in time and expertise. | Mobile is the original platform and remains critical. Android (Java/Kotlin) and iOS (Swift) require separate, native development efforts, which is the most time-consuming and costly aspect of client-side development. |
| **Development Tools** | Xcode is the primary IDE for macOS development, using Swift or Objective-C as the main languages. | Server-side development on Linux can utilize a wide range of tools, including IntelliJ IDEA, Visual Studio Code, and Eclipse, with languages like Java, Python, or Node.js. | Windows development commonly uses Visual Studio with C# or C++. Cross-platform tools like .NET MAUI could also be considered for client apps. | Android Studio (for Android with Java/Kotlin) and Xcode (for iOS with Swift) are the standard, platform-specific IDEs and toolkits. |

**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:** We recommend deploying the game's backend services on a **Linux server platform**, specifically within a cloud environment such as Amazon Web Services (AWS) or Google Cloud Platform (GCP). This recommendation is based on Linux's cost-effectiveness (no licensing fees), renowned stability and performance for network applications, and its flexibility as a neutral, high-performance hub for client applications on all other platforms.
2. **Operating Systems Architectures:** The recommended Linux platform utilizes a **monolithic kernel** architecture. In this model, the core operating system kernel manages all system resources (CPU, memory, etc.) in a single, integrated address space. This provides superior performance for server workloads due to efficient inter-process communication within the kernel, which is essential for handling the real-time, concurrent connections of *Draw It or Lose It*.
3. **Storage Management:** We recommend a hybrid storage management system for the Linux platform. The server itself will use the **ext4** filesystem (or a cloud-equivalent) for the OS and static game assets. For dynamic game data (user profiles, active game states, scores), we recommend using a **Relational Database Management System (RDBMS) like PostgreSQL**. This separates large, static files from frequently accessed, relational data, optimizing both storage I/O and application logic.
4. **Memory Management:** The Linux platform employs sophisticated memory management techniques crucial for a responsive game. It uses a **demand-paging system with virtual memory**, giving the game server process the illusion of a large, contiguous address space. This isolates the game server in its own protected memory space, prevents crashes from affecting the entire system, and allows the kernel to efficiently allocate physical RAM to the most active data, such as current game states and active user connections, ensuring low latency.
5. **Distributed Systems and Networks:** *Draw It or Lose It* is a distributed system where clients on various devices interact with the central Linux server. This is accomplished by using **WebSockets** for real-time communication (drawing data, chat) and **RESTful APIs over HTTPS** for operations like login. Dependencies include:
   * **Connectivity:** The system requires internet access. Software must handle network interruptions gracefully with reconnection logic.
   * **Latency:** To minimize lag, the server will be hosted in a geographically central cloud region.
   * **Outages:** The cloud infrastructure should be deployed across multiple availability zones, with regular backups to mitigate the impact of outages and allow for recovery.
6. **Security:** Protecting user information is achieved through a layered security approach on the Linux platform:
   * **Data in Transit:** All client-server communication is encrypted using **TLS (Transport Layer Security) 1.3 or higher**.
   * **Data at Rest:** User passwords are **hashed and salted** (using bcrypt) in the database. The server's filesystem and database are encrypted using cloud provider tools.
   * **Platform Security:** The Linux server will be "hardened" by following the principle of least privilege, disabling unnecessary services, and using a firewall (iptables/ufw) to restrict access. A robust token-based authentication system (e.g., JWT) will ensure users can only access their authorized data.