

<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 | <07/18/2004> | <Sarvarbek> | Initial draft of software design document and game application. |

**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)

*Draw It or Lose It* is currently only on Android, but The Gaming Room wants it playable in any web browser. We will put one small Java program on the server called GameService. This single program holds every game’s data, hands out unique names and IDs, and makes sure only one copy of itself runs. With this in place, we can quickly build a simple web front-end and later move the game to cloud or other devices.

## Requirements

*Each match must allow several teams, and every team must hold several players. All games, team, and player names must be unique so users can’t pick duplicates. Only one GameService object should ever exist, and it must assign unique numeric IDs to everything it creates. The whole application must run anywhere Java runs and must connect over secure HTTPS or WebSocket.*

## [Design Constraints](#_2et92p0)

The server must keep the true game state, so players don’t lose progress if they refresh or disconnect. Because the game should scale later, the server cannot store data in a single user session; it must remain stateless. We will write everything in Java so we can reuse code from the Android version. The first release must run inside one small Linux container, so it needs to be light on CPU and memory. All traffic must stay encrypted, and any stored user data must be safely hashed or salted.

## [System Architecture View](#_ilbxbyevv6b6)

For this assignment we do not need a full architecture diagram. At a high level, the game will run on one small Java server (for now a single Linux container). Browsers connect to that server over HTTPS or WebSocket, send moves, and receive updates. No other tires or hardware details are required at this stage.

## [Domain Model](#_8h2ehzxfam4o)

The UML diagram shows a clean hierarchy. At the top is an Entity class that stores two common pieces of data—id and name. Three other classes—Game, Team, and Player—inherit from Entity, so they automatically get those two fields. A Game owns many Team objects, and each Team owns many Player objects; these “has-many” links are drawn with the lines ending in crow’s-feet.

GameService is a singleton: only one copy of it exists, and it holds a list of all games. That class hands out the next unique IDs for games, teams, and players. When you add a game, team, or player, the code first loops through the existing list (an iterator pattern) to see if the name already exists, which enforces the “unique name” rule. Finally, two simple driver classes—ProgramDriver and SingletonTester—exercise the code and prove that every call to GameService.getInstance() returns the same object.

The diagram demonstrates three main object-oriented principles. **Inheritance** removes duplicated code by letting Game, Team, and Player share fields from Entity. **Encapsulation** keeps each class’s data private and exposes only simple methods like getName(). **Abstraction** hides the list and ID logic inside GameService, so the rest of the program just asks for new objects without worrying about how they are stored or identified. Together these patterns satisfy the client’s needs with clear, reusable code.

**"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** | Fine for local testing, but pricey and rarely offered by cloud hosts. | Free, stable, and the top production choice thanks to Docker/Kubernetes support. | Runs Java well but adds license cost, higher RAM use, and more maintenance. | Hardware limits make them unsuitable for real web hosting. |
| **Client Side** | Safari quirks mean extra browser testing, but the user base is smaller; development cost is mainly the price of a Mac for QA. | Desktop Linux users run Chrome and Firefox like other platforms, so no special code is required and support costs are minimal. | Windows has the biggest desktop share, so thorough testing on Edge and legacy browsers adds time and QA effort. | Most traffic now comes from mobile; the web client must be touch-friendly, responsive, and tolerant of slow networks, which increases design and testing time. |
| **Development Tools** | IntelliJ IDEA, Eclipse, or VS Code plus Homebrew for Git, Maven, and Docker; Xcode is handy for artwork. | OpenJDK, Maven/Gradle, IntelliJ or Eclipse, Docker, Kubernetes, and rich command-line tools come pre-packaged and free. | VS Code, IntelliJ/Eclipse, Git, and Docker Desktop work well; WSL 2 gives a Linux-like shell for builds. | Android Studio for Android, Xcode for iOS, Chrome DevTools’ mobile simulator, and optional React Native or Flutter for hybrid apps. |

## 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. For The Gaming Room's server platform, I recommend using Ubuntu Server 22.04 LTS deployed on a cloud provider like AWS, Google Cloud, or Azure. This choice makes the most sense for several reasons. First, Ubuntu LTS gives us five years of security updates without forcing major upgrades that could break the game. Since it's free and open source, we won't have licensing costs eating into the budget. Ubuntu works great with Docker containers, which we'll need for scaling the game as more players join. The server version is lightweight too - it only needs about 512MB of RAM and 2.5GB of disk space to run, leaving plenty of resources for the actual game. Most cloud providers have Ubuntu as a standard option with good documentation, so deployment and troubleshooting will be straightforward. The huge community support means we can find solutions to problems quickly without expensive consultants.
2. Ubuntu Server uses monolithic Linux kernel architecture, which basically means the core operating system runs as one large program with everything built in, but it can load extra modules when needed. The kernel manages all the important stuff through different subsystems. For process scheduling, Linux uses something called the Completely Fair Scheduler (CFS) that keeps track of how much CPU time each process gets and tries to distribute it evenly - this is important so no single game session hogs all the resources. The Virtual File System layer lets us use different types of storage transparently, whether it's the regular ext4 filesystem, fast temporary storage in RAM, or network storage.

In the user space (where our game application runs), everything communicates with the kernel through system calls. The systemd service manager makes sure all our services start in the right order and automatically restarts them if they crash. The networking is handled in layers - the kernel deals with the low-level TCP/IP stuff while our application handles the game-specific protocols on top. For containers, Linux uses namespaces to isolate processes from each other (so one container can't mess with another) and cgroups to limit how much CPU, memory, and network each container can use. This isolation is crucial for stability when running multiple game instances.

1. For storage, we need a multi-tier approach to handle different types of data efficiently. The main filesystem should be ext4 because it's mature and reliable with good performance. It has journaling, which means if the server crashes, we won't lose data or corrupt the filesystem. We should use Logical Volume Manager (LVM) to organize our storage - this lets us resize partitions on the fly without taking the game offline. I'd set up separate volumes for different purposes: database files on fast SSDs, logs on regular storage with compression to save space, and temporary files in RAM for speed.

For the database, PostgreSQL is the way to go. It uses something called MVCC (Multi-Version Concurrency Control) which means multiple players can read and write data at the same time without locking issues. We should put the Write-Ahead Logs on separate fast drives to improve performance. For frequently accessed data like active game sessions, Redis is perfect as an in-memory cache. It's single-threaded so there are no concurrency issues, and it's incredibly fast. We'll configure it to save snapshots every 15 minutes and keep an append-only file for recovery if it crashes. With Redis Cluster, we can spread the load across multiple servers as we grow.

1. Memory Management

Memory management happens at multiple levels in our architecture. At the Linux kernel level, there's a buddy allocator that manages physical memory by splitting and combining blocks of different sizes to minimize fragmentation. The kernel also uses a slab allocator to cache commonly used objects like network buffers, which speeds up allocation significantly. It keeps track of which memory pages are frequently used (hot) and which aren't (cold) using LRU lists, so it knows what to keep in RAM and what can be swapped out if needed.

Each process gets its own virtual address space, which on 64-bit systems is huge (256TB). The CPU's Translation Lookaside Buffer (TLB) caches the mappings between virtual and physical addresses for speed. Linux can also use Transparent Huge Pages, which are 2MB instead of the normal 4KB, reducing overhead for large memory allocations.

For the Java application, we'll use the G1 garbage collector which divides the heap into regions and can clean them up independently. This helps keep pause times short - we'll target 200ms maximum to maintain smooth gameplay. We should start with a 1GB heap and adjust based on actual usage. In Docker containers, we'll set memory limits to prevent any single container from using too much RAM and affecting others. A 2GB hard limit with a 1.5GB soft limit should work well initially.

1. Distributed Systems and Networks

The network architecture needs to support both REST APIs for game setup and WebSockets for real-time gameplay. We'll optimize the Linux TCP stack with features like TCP Fast Open to reduce connection setup time and TCP\_NODELAY to ensure game packets are sent immediately without buffering. For load balancing, we'll use either HAProxy or NGINX with consistent hashing based on player session IDs - this ensures players always connect to the same game server instance during a session.

Service discovery is handled through Consul, which keeps track of which services are running, where and their health status. If a server goes down, Consul automatically removes it from the rotation. For handling game events that don't need immediate processing, we'll use Apache Kafka as a message queue. It can handle huge volumes of events and guarantees they won't be lost even if a server crashes. Participate in partition topics for parallel processing and replicate data across multiple brokers for fault tolerance.

For state synchronization between servers, we'll use Redis Pub/Sub for real-time updates and implement optimistic locking to handle concurrent modifications. Circuit breakers (using Hystrix or similar) will prevent cascading failures - if one service starts failing, the circuit breaker trips and stops sending requests to it until it recovers. We'll deploy across multiple availability zones so if one data center has issues, players can still connect to others.

1. Security

Security needs to be implemented at every layer of the system. At the OS level, we'll enable SELinux to restrict what each process can do, even if it gets compromised. We'll use auditd to log all important system events like file access and network connections. The kernel will be hardened with sysctl settings to prevent common attacks - things like disabling IP forwarding, enabling SYN cookies for flood protection, and turning off unnecessary network features.

For network security, we'll configure the firewall with iptables to only allow necessary ports (443 for HTTPS, WebSocket port, and SSH only from our management network). Fail2ban will monitor logs and automatically block IP addresses that show suspicious behavior, like too many failed login attempts. We'll implement rate limiting to prevent any single player from overwhelming the server with requests.

At the application level, Spring Security will handle authentication and authorization. We'll use JWT tokens for stateless authentication with short expiration times (15 minutes) and refresh tokens for convenience. All passwords will be hashed with bcrypt (cost factor 12) which is specifically designed to be slow and resist brute force attacks. For sensitive data like personal information, we'll use AES-256 encryption with keys stored in a secure vault service that rotates them regularly.

All network communication will use TLS 1.3 with perfect forward secrecy, meaning even if someone steals our encryption keys later, they can't decrypt past communications. For containers, we'll scan all images for vulnerabilities during the build process and only run containers as non-root users with minimal permissions. We'll use centralized logging with the ELK stack to monitor security events and set up alerts for suspicious activities. Regular compliance scans will ensure we're following security best practices and meeting any regulatory requirements.