

***Draw It or Lose It***

# **CS 230 Project Software Design Template**

Version 1.0.2

## 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 |
| --- | --- | --- | --- |
| 1.0.0 | 7/20/25 | Noah Stubbs | Added Executive Summary, Design Constraints, Domain Model |
| 1.0.1 | 8/2/25 | Noah Stubbs | Added ‘Evaluation’ section |
| 1.0.2 | 8/15/25 | Noah Stubbs | Added ‘recommendations’ section |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room is expanding their mobile game, *Draw It or Lose It*, to be cross-platform and web-based. The proposed solution is a scalable, object-oriented Java (unless we’re ready to finally pivot to Kotlin) application that is built around a focal point of reusable patterns and a strict memory model. The application will support multiple teams and players, maintain unique naming, and allow for future deployment to additional platforms.

## Requirements

* Must support one or more teams
* Supports multiple players on each team
* Game and team names must be unique (to prevent duplicates)
* Only one instance of the game can exist in memory at any given time
* System must assign unique identities to every game, team, and player
* Game built for a web-based, distributed environment for scalability

## [Design Constraints](#_2et92p0)

* Singleton Pattern; supports only one instance of the game existing in memory. Ensures single point of control for game state, enforcing memory management and preventing session conflicts
* Unique Naming Requirement; game and team names must be one-of-a-kind
* Distributed Deployment; must be web-deployable; avoid local state dependencies, use centralized data handling
* Identifier Management; every entity requires unique IDs generated and tracked efficiently. Necessary, but will add complexity to class responsibilities
* Scalability Concerns: The design must support many concurrent users without performance degradation, so resource management and design clarity are critical from the start.

## [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)

* Entity parent class holds common attributes (id, name) with accompanying methods; exemplifies inheritance and code reuse.
* Game, Team, and Player all extend Entity; further reenforcing inheritance and encapsulation.
* Private Constructor and public getInstance() imply that GameService is a singleton. This will be what ensures only one instance manages state across the app.
* Each Game owns a list of Team objects, each Team owns a list of Player objects, modeling a “has-a” relationship / nested object hierarchy.

**"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)

|  | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | MacOS is Unix-based, so it offers stability and built-in support for many server tools. However, sticking with the theme of the rest of the Apple ecosystem, there is a high premium on their identity of “it just works”. Licensing a Mac server will be the most expensive option, and the required Mac hardware is going to limit scalability and cloud options. Perfectly suitable for development, testing, or staging, but not practical nor cost-effecting for hosting large-scale web apps. | \*Gold Standard\* Linux is the industry due to its stability, scalability, and flexibility. It supports (basically) everything out of the box. Highly customizable, excellent community support, free distributions, and already the backbone of most cloud hosting providers. It's the best choice for deploying a scalable, cost-effective web-app. | Windows Server offers robust .NET-based applications and a familiar environment for devs that are already used to Microsoft tools. Downsides are that it comes with licensing fees and is less resource-efficient than Linux. It *can* host modern apps fine; it’s better suited for enterprise environments already married to Microsoft infrastructure. Perfectly capable, but not the most cost-effective. | Mobile devices are not viable for server-side hosting. They lack stability and processing power. Overall they are fundamentally not designed to serve applications. There’s no real server-based deployment method for iOS nor Android (outside of edge case technicalities like personal file sharing or niche IoT applications.). |
| **Client Side** | Straightforward since Safari comes preinstalled and Mac also supports Chrome/Firefox. Cost can increase as Mac-only devices are needed for testing. Since the app is browser based, developer expertise in macOS isn’t critical, BUT Safari historically lags behind in supporting the latest HTML5/CSS3 features, so a macOS specific QA is necessary. | Linux users typically run Firefox or Chrome, both of which follow modern web standards closely. As an open-source platform, Linux again avoids licensing costs and offers wide and free dev tools. Downside is that there can be significant variation between distributions, so testing across a few major distros is ideal. It *shouldn’t* matter, but as soon as we assume it won’t, it magically will. | Windows has the largest desktop market share, so it’s debatably the top priority for client-side testing. Chrome and Edge are the most used, and they are well supported. Development costs and expertise can be considered “moderate” or “average”. Most teams are already familiar with the platform, and testing is easier due to its ubiquity. | Developing for both iOS and Android is more demanding. There’s a large peripheral shift to small screens with only touch input. This typically isn’t extremely difficult to work around, though it does require deliberate design decisions. Not only can the browsers vary from each other; all browsers on iOS are forced to run on a Safari backbone. So, Chrome on Android will have differences from Chrome on iOS. Testing on real devices is critical. Cost and time will increase to validate UI and performance across many variables (different screen sizes, OS versions, etc.) |
| **Development Tools** | If targeting Mac, Xcode (Apple’s free IDE) has native tools, but since it’s a web-based app, cross platform IDEs like Visual Studio Vode or Eclipse are more than sufficient and widely supported. Web-Dev toolchains like Node.js, React, and Docker are fully supported. Main limitation is hardware; must have Mac to run macOS. | Again, Linux supports (basically) everything. Open-source tools like VS Code (p.s. this is what I use for everything. Java for you, C++ for my other class, and Rust for me all in the same IDE), Eclipse, Atom, and full CLI toolchains like vim + tmux + Git. Web stack tech like Node.js, MongoDB, and Docker work exceptionally well in Linux environments. Most customizable and scriptable for automations. All tools are free and again, Linux has no licensing fees, so it’s the most cost-efficient platform. Downsides are that slightly less mainstream IDEs and distros may lack some GUI-based tools. | Visual Studio (the big bloated one, not Visual Studio Code, Microsoft again demonstrating they come up with naming schemes on the “bring your kid to work” days) is the dominant IDE and offers excellent support for full-stack development, especially with ASP.NET if desired. Web frameworks like React, Vue, or Angular work well on Windows. Tools like Docker Desktop and WSL (Windows Subsystem for Linux) bring Linux-native compatibility into the mix. Licensing costs vary; Visual Studio Community is free, but Pro and Enterprise editions aren’t. Hardware costs is again “moderate” or “average” with slew of device options available. | Most web frameworks support responsive design and mobile-first development. Testing frameworks like BrowserStack or Sauce Labs may be needed to emulate various devices, which could add subscription costs. Tools like Chrome DevTools (mobile emulation) are free but not always enough. Real device testing is ideal but comes with the obvious significant jump in cost and complexity. |

**Recommendations**

**Operating Platform:**

* Linux server platform, deployed in containers (like Docker)
  + Lowest cost
  + Easy to scale
  + Widest tooling
  + Mature networking/storage stacks
  + Clean path to web-based, multiplatform access

**Operating Systems Architectures:**

* Monolithic kernel with loadable modules
  + Multiple services (API, game logic, worker jobs) can run separately but work together
  + Containerizations keep each service isolated and easy to update without downtime
  + Uses fast file systems and good I/O handling for web apps

**Storage Management:**

* Main database
  + PostgreSQL for storing user accounts, games, teams, and scores
* Cache
  + Redis for quick access to temporary data like game sessions
* Game files/assets
  + Stored in object storage and delivered by a CDN
* Backups
  + Automated, encrypted, and restorable in case of failure
  + Encrypted at rest

**Memory Management:**

* OS uses virtual memory; containers have set memory limits so one can’t overload the server
* The java app will have fixed memory limits and a garbage collector tuned for low pauses
* Large data streamed vice loaded all at once, temp data to Redis vice filling server memory

**Distributed Systems and Networks:**

* All clients connect to the same HTTPS API
* REST for most actions, WebSockets for live game updates
* Services check own health, retry failed requests, and still work (limited) if one part goes down
* Game data stored centrally, seamless reconnect after a drop

**Security:**

* Login
  + OAuth 2.1 with short-term tokens
* Encryption
  + Encrypted storage for all sensitive data
  + TLS 1.3 for all communications
* Access control
  + Permissions based on roles
* Safe coding
  + Protects against common attacks
* Operations
  + Network Segmentation
  + Keep services separate
  + Minimal access rights
  + Log security events
  + Regular patches