

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 | 09/28/25 | Ismail Mokoena | <Brief description of changes in this revision> |

**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 Gaming Room project’s goal is to develop a web-based game based on the game ‘Draw It or Lose It’ that serves multiple platforms/Operating Systems. Multiple teams consisting of a variety of players, engaging in four rounds, lasting a minute per round. During each teams turn, The application renders images from a library pool of images. By the 30 second mark, the image is fully rendered. If the team does not guess by the expiration time, each member on the other team has the opportunity to guess, with a 15 second time limit. Currently, ‘Draw it or Lose it’ is only available via Android app

## Requirements

*- A game will have the ability for multiplayer, having one or more teams involved.*

*- Each team will have multiple players assigned to it.*

*- Game and team names must be unique.*

*-Only one instance of the game can exist in memory at any given time.*

1. Increase reach by supporting iOS, Android, and desktop web users.  
2. Deliver a web-based, responsive HTML interface that communicates with a scalable server backend.  
3. Keep implementation and operational costs manageable (consider cloud hosting and containerization).  
4. Ensure successful publication in Apple App Store and Google Play Store if native/wrapper apps are used.  
5. Minimize maintenance complexity while enabling high availability and scalability for thousands of players.

## [Design Constraints](#_2et92p0)

Cross-platform compatibility (iOS + Android + desktop browsers), responsive UI, cloud-ready server architecture, adherence to app store policies (if native/wrapped), limited budget, and a target to reuse as much front-end code as possible between platforms.

*Only one instance of the game can exist in memory at any given time*.

To reach a wider audience platforms supported must be web-based application and the mobile application.   
This will include further training in maintaining the web program after completion.

Budget can also be a big concern due to the complexity of creating a consistent experience amongst all

users, because not all web browsers are made equally

**Cross-platform support**

* The game needs to run on different systems (Windows, Mac, Linux, mobile).
* This means the code and user experience have to stay consistent everywhere.
* Constraint: You can’t build something that only works well on one platform.

**Distributed environment**

* The game is not running on just one machine. It’s running across a network (servers + clients).
* That means network speed, internet reliability, and syncing between devices all matter.
* Constraint: The design has to handle latency, dropped connections, and multiple players at once.

**Security**

* User information (accounts, scores, game sessions, etc.) must be protected.
* Data moving between devices and servers has to be encrypted.
* Constraint: You can’t expose user data or send it in plain text, even if that would be “faster” to build.

**Performance / scalability**

* The system needs to handle many players at once, not just one.
* It should also be able to grow as more users join, without a full rewrite.
* Constraint: You can’t design something that only works for a small number of users.

**Cost and maintainability**

* The solution should not require expensive licensing for every platform.
* It should be easy to update and support long term.
* Constraint: High-cost or high-maintenance platforms (like needing separate, totally different codebases for each OS) are less desirable.

**Consistent player experience**

* Players on different platforms should still feel like they’re playing the same game.
* Constraint: UI and behavior have to be predictable, even though the hardware is different (phone vs PC, etc.).

A web-first approach (responsive single-page application) reduces duplication of business logic, allows immediate desktop browser support, and can be packaged as a Progressive Web App (PWA) or wrapped in native shells (Capacitor, Cordova, or Flutter) where native device APIs or app-store presence is required. Server-side should be stateless where possible, use containerization (Docker) and orchestration (Kubernetes) for scaling, and be deployed on a platform that balances cost, familiarity, and vendor support.

## [Domain Model](#_8h2ehzxfam4o)

Below shows that we created a package containing the classes for Entity, Team, GameService, Game,

Player, Program Driver, and SingletonTester. We developed the Entity class as a base class that has an

inheritance relationship with the Game, Team, and Player classes. There is an associative relationship

between GameService, Game, Team, and Player classes. Each instance of GameService “has-a”

(represented by 0…\*) instance of the Game class, which in turn has an instance of the Team class, which

in turn has an instance of the player class. The dependency on the instances throughout the program

creates the structure needed to iterate through lists that exist within the Team and Player classes that

help enact a requirement from the Gaming Room to ensure that there is only one instance of each team

and each player during any given game. The program driver class is shown to use the SingletonTester

class to ensure that there is only one instance of the GameService at any time

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Mac OS has easy  accessibility and  server  configurability with  an easy to use GUI  (Graphical user  Interface),  featuring flexible  terminal  commands.  Available only on  Apple-products. | Linux is cost-  efÏcient, mostly  free, and mostly  open source. It  can be difÏcult to  navigate based on  user-experience.  It has a command  shell for simple  server  configurations | Licensing fees can  be costly, making  it more expensive  than Linux, yet is  more accessible  than macOS. User  friendly GUI  (Graphical User  Interface) and  contains a  command  prompt.  Accessible and  works with server  services such as  Microsoft Azure | Mobile device  specifications vary  model to model, by  extension person to  person. Hardware  can effect the ability  to run software as  most applications  are not cross-  platform  compatible |
| **Client Side** | MacOS is made by  Apple.  It may be less  desirable for  external attacks  and adds a feature  of safety to its  users. It has a sleek  design that  connects  seamlessly to other  Apple-branded  products. Though  when it comes to  hardware, Apple  tends to be a bit  rigid, lacking  customizability.  Because of the lack  of Graphics  Memory, Mac OS  isn’t good for  gaming, and can be  considered  expensive since it  is exclusive to  Apple-branded | Linux is open-  source, stable,  reliable, highly  cost effective and  customizable.  Linux does have a  lack of software  and hardware  availability,  commercial  support, a steep  learning curve,  and compatibility  issues. It is not  necessarily the  most user-friendly  to those not of an  IT/Computer  background, and  has less exposure  when compared  to other major  operating  systems. | Windows is the  most widely used  personal  computer  operating system.  It is user-friendly,  has great  software  compatibility and  hardware  compatibility with  customization  possibilities. Most  games released  for Linux or mac  are also released  for Windows.  Most games  developed for  older versions will  still work on  newer operating  systems. | Mobile devices have  weaker hardware  than what  traditional PCs  contain. They are  highly convenient  and multipurpose  with features for  communications,  location tracking,  NFC payment  options, cameras,  and various others.  Mobile devices are  increasingly  common and are  great for purposes  of entertainment.  Cost can vary based  on specific devices.  Can offer seamless  transitionary  experiences with  other operating  systems (such as iOS  seamless  compatibility with  macOS). |
| **Development Tools** | <Identify the relevant programming languages and tools (IDEs and other tools) that are used to build this type of software for deploying on Mac.> | <Identify the relevant programming languages and tools (IDEs and other tools) that are used to build this type of software for deploying on Linux.> | <Identify the relevant programming languages and tools (IDEs and other tools) that are used to build this type of software for deploying on Windows.> | <Identify the relevant programming languages and tools (IDEs and other tools) that are used to build this type of software for deploying on Mobile Devices.> |

## Evaluation

Below is a platform-by-platform evaluation focused on server-side hosting feasibility, licensing implications, client-side compatibility, and development/tooling impact.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Platform | Server-side hosting suitability | Advantages | Weaknesses / Risks | Licensing & Cost Notes |
| Linux (e.g., Ubuntu, CentOS, Debian) | Highly suitable — industry-standard for web servers; excellent support in cloud providers and containers. | Open-source (low OS licensing cost), mature tooling (Nginx, Apache, systemd), excellent container/K8s support, strong security/hardening practices, broad community support. | Requires Linux expertise for administration; fragmentation between distributions can trip inexperienced teams; commercial support available at cost (e.g., Red Hat). | Minimal OS licensing cost for community distros. Commercial Linux (RHEL) or enterprise support incurs subscription fees; cloud VM costs still apply. |
| Windows Server | Suitable — common in enterprises, first-class support for .NET applications and Microsoft stacks. | Easier if existing stack is .NET (especially .NET Framework / Windows-only components), integrated with Active Directory and Windows tooling, vendor support from Microsoft. | Higher licensing cost, larger attack surface historically (requires strong patching), less common for container-first deployments (though Windows containers exist). | Windows Server requires licensing (per-core / CALs); using Windows-based cloud VMs carries Windows licensing costs. Consider .NET Core / .NET 6+ on Linux to avoid Windows Server licensing. |
| macOS (Apple hardware required) | Not ideal for general web hosting at scale — Apple hardware requirement makes it costly and uncommon for general-purpose hosting. | Necessary if you need native macOS-only services, or for iOS build servers (Xcode/macOS is required to build signed iOS apps). | High hardware cost, not cloud-native in typical providers (few macOS cloud options; macOS Server is deprecated), limited ecosystem for large-scale web hosting and automation compared to Linux/Windows. | mac hardware acquisition and maintenance are the main costs. Licensing for macOS is tied to Apple hardware; Xcode is free but provisioning and App Store developer program (~$99/year) is required for iOS distribution. |
| Mobile (iOS & Android) - client-side | Client apps will connect to server; server can be hosted on any of the above platforms — mobile platforms aren't used for hosting the backend. | Wide reach when supporting both iOS and Android; responsive web + PWA can handle many use cases; native wrappers allow App Store distribution and device APIs. | Native development doubles platform-specific work if using fully native stacks; iOS requires macOS for final builds and App Store signing. Performance nuances across devices must be tested. | iOS app distribution requires Apple Developer Program ($99/year). Google Play console registration is a one-time fee (~$25). |

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Mac servers are easy to manage and user-friendly, but they are costly and less common in large-scale deployments. | Linux servers are open-source, reliable, and widely used for web apps. They are secure and easy to customize. | Windows servers are powerful and integrate well with Microsoft tools but require licensing and more maintenance. | Mobile devices have limited server capability but can connect to cloud servers to sync and access data. |
| **Client Side** | Developing for Mac requires Apple-specific tools and devices, which can increase cost and limit flexibility. | Linux client development is flexible and cost-effective, but user interface design may vary by distribution. | Windows clients are common and easy to deploy, but development can take longer due to varied configurations. | Mobile client development needs to support Android and iOS, requiring separate design and testing efforts. |
| **Development Tools** | For Mac, developers can use Swift, Xcode, and .NET MAUI for cross-platform builds. | For Linux, developers often use Python, Java, or C++ with IDEs like Eclipse or VS Code. | For Windows, Visual Studio is the main development tool, supporting C#, .NET, and Unity for game development. | For Mobile, Android Studio and Xcode are used with languages like Kotlin, Swift, and Flutter for cross-platform apps. |

**Client-Side & Cross-Platform Development Considerations**

To support desktop browsers (Linux, macOS, Windows) and mobile devices (iOS, Android) with a single, maintainable codebase, the development approach should prioritize a web-first responsive SPA and reusable components. Key considerations:

- Cross-browser compatibility: test and support modern versions of Chrome, Edge, Firefox, and Safari. Use feature detection/polyfills and an automated test matrix (e.g., BrowserStack, Playwright).

- Responsive layout and accessibility: implement a mobile-first responsive design using CSS Grid/Flexbox and component libraries; follow WCAG accessibility basics.

- Networking and offline behavior: adopt progressive enhancement and consider PWA features (service workers, caching) to improve perceived performance and resilience.

- Native device features: if device APIs are required (push notifications, advanced sensors), use a cross-platform framework (React Native, Flutter) or wrapper (Capacitor), or implement thin native modules.

- Testing and QA: automated unit, integration, and end-to-end tests; include device farm testing for a sample of real devices and OS versions.

## Estimated Development Effort & Team Impact

Below are qualitative estimates and recommended team structure and skills required.

Recommended core team:  
• 1–2 Backend engineers (Node.js/Python/Java/.NET) familiar with scalable web services, databases, and containerization.  
• 1–2 Frontend engineers (React/Vue/Angular) experienced in responsive design and SPA architecture.  
• 1 Mobile developer (or 1 cross-platform developer) if targeting native wrappers or deeper native integration (React Native/Flutter),  
• 1 DevOps/SRE (or shared contractor) to setup CI/CD, container orchestration, monitoring, and deployment automation.  
• 1 QA engineer to manage automated tests and device testing.

If using purely web + PWA + wrapper approach, a smaller team can achieve the goal faster; choosing fully native iOS + Android increases time and cost significantly.

Rough relative effort (approx):  
- Web-first SPA + PWA + wrappers (recommended): Medium cost/time, lower long-term maintenance.  
- Fully native iOS + Android + web: High cost/time, higher maintenance (duplicate logic and feature parity challenges).  
- Backend on Linux (cloud containers): Lower operational licensing cost, medium engineering effort for setup and automation.  
- Backend on Windows Server (if using legacy .NET Framework): Higher licensing cost, moderate engineering effort.

## Development Tools, Languages & Licensing

Typical stack choices and their implications:  
- Backend: Node.js (JavaScript/TypeScript), Python (Django/Flask/FastAPI), Java (Spring), or .NET (Core/6+) — choice driven by team expertise.  
- Frontend: HTML5, CSS3, JavaScript/TypeScript, modern SPA frameworks (React recommended for component reuse), component libraries.  
- Mobile: React Native or Flutter for cross-platform native-feeling apps; Swift (native iOS) and Kotlin (native Android) if choosing native.  
- Databases: PostgreSQL, MySQL, or cloud-managed DBs; Redis for caching and session data; consider managed services to reduce ops overhead.  
- Dev Tools / IDEs: VS Code (free), IntelliJ / WebStorm (paid options), Xcode (free on macOS), Android Studio (free), Visual Studio (Community free, Professional/Enterprise paid).  
- CI/CD: GitHub Actions, GitLab CI, Jenkins, or cloud CI; container registry (Docker Hub, ECR) and orchestration (Kubernetes).

Licensing notes:  
- Many core developer tools and languages are open-source and free to use. IDEs like WebStorm or IntelliJ IDEA Ultimate carry licensing fees for teams.  
- Apple Developer Program (~$99/year) is required for iOS distribution. Google Play console has a one-time developer registration fee (~$25).  
- Enterprise Visual Studio or specialized commercial tooling and testing clouds may add costs depending on team choices.

## Recommendations

1. Host the backend on Linux-based cloud VMs or managed container services (AWS, GCP, Azure). Linux provides lower OS licensing cost and the broadest ecosystem for web hosting.  
2. Containerize services (Docker) and use orchestration (Kubernetes or managed K8s) for scaling to thousands of players. Use load balancers and autoscaling groups.  
3. Build a web-first responsive SPA (React + TypeScript) and implement it as a PWA. Use wrappers (Capacitor or Flutter) only if App Store presence or additional device APIs are required.  
4. For iOS builds and App Store signing, maintain at least one macOS build agent (mac mini or hosted macOS CI) because Apple requires macOS for code signing.  
5. Favor cross-platform solutions (React Native or Flutter) only if the product needs heavy native integrations or if a near-native performance is needed; otherwise, SPA + PWA is more cost-effective.  
6. Plan for automated testing across browsers and real-device testing for mobile platforms; leverage device farms for wide coverage.

## 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**: I would recommend Windows, because it is the most widely used and available operating system.
2. **Operating Systems Architectures**: The Windows API (Application Programming Interface) is the programming Interface of the Windows OS. It provides services used by all Windows based applications to enable applications to provide a Graphical User interface (GUI), access system resources, incorporate audio and much more. Some major categories of Windows API functions include: Base Services, Component Services, Graphics & Multimedia Messaging, Networking and Web Services
3. **Storage Management**: Within Windows 10, one of the modern iterations of the Windows OS, is a feature called “Storage Sense”. The purpose of this feature is to allow the user to allow windows to automatically free up drive space when items that are no longer needed are cluttering the drive space, such as temporary files and items in your recycle bin that you may forget about
4. **Memory Management**Windows uses techniques such as Swapping (By the use of the swap- in, or moving the program from the hard disk to the RAM, and swap-out, or moving the program from the RAM to the hard disk.)
5. **Distributed Systems and Networks**: By using something like the HTTP (Hypertext Transfer Protocol) you can load web pages using hyperlinks. It acts as an application layer protocol designed to transfer information between networked devices and runs on top of other layers of the network protocol stack. Using an HTTP allows information between clients and servers to be transferred. HTTP relies on the TCP reliability. It uses a multiplexing technique which sends multiple requests through a single TCP connection, making it more efÏcient.
6. **Security**: Since Windows is the most widely used operating platform, security can be a concern.

Windows does provide a basic anti-virus and firewall system to help combat computer viruses

and other malwares. If you choose to not use Windows pre-installed software, a third-party anti-virus and anti-malware software would be recommended for security