

Web Migration Strategy for “Draw It or Lose It”

# **Software Design Document**

Version 0.1

## Table of Contents

[**Software**](#_Toc115077317) **Design Document**

[**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 |
| --- | --- | --- | --- |
| 0.1 | 03/23/25 | Christian Decker | Fill in sections: Executive Summary, Design Constraints, and Domain Model. |

## [Executive Summary](#_sbfa50wo7nsh)

Creative Technology Solutions proposes a server-side Java with web frontend approach to migrate The Gaming Room's Draw It or Lose It Android application to a cross-platform web environment. This strategy preserves the existing Java codebase as a backend service while developing a new web frontend using modern web technologies. The approach maintains the core game logic, including the critical singleton pattern that ensures only one game instance exists in memory, while enabling deployment across multiple platforms through standard web browsers.

By leveraging the existing Java investment and creating a clean separation between frontend and backend components, this migration path offers the most efficient route to market while addressing all software requirements. The resulting architecture will support multiple teams with multiple players, maintain unique identifiers for games, teams, and players, and ensure only one instance of the game exists in memory at any time.

## Requirements

These requirements are for a different project, but I’ve included them here to provide a template.

1. Develop a mobile app compatible with both iPhone and Android smartphones.

2. Ensure the app is optimized for app store listings on both platforms.

3. Implement features that enhance user engagement and experience.

4. Maintain development within the allocated budget.

## [Design Constraints](#_2et92p0)

1. **Client-Server Architecture.** The existing monolithic Java application must be restructured into a client-server architecture.

This fundamental architectural shift is necessary because web browsers cannot directly execute Java code, especially with the deprecation of Java applets. By separating the presentation layer (frontend) from the business logic (backend), we can maintain the existing game logic while providing a web interface accessible across platforms. This also enables centralized game state management, addressing the requirement that only one game instance can exist in memory.

1. **Real-Time Communication Requirements.** The web frontend must maintain synchronized state with the Java backend in real-time.  
     
   Time-sensitive gameplay elements, such as the 60-second rounds and progressive drawing reveals, require near-instantaneous communication between clients and server. WebSockets or similar technology must be implemented to ensure all players see consistent game state and timing. Without this, the core gameplay experience would be compromised as different players would see different game states.
2. **State Management and Persistence.** The Java backend must maintain game state across multiple distributed clients.  
     
   The current implementation uses in-memory collections (ArrayList) to track games, teams, and players, assuming a single application instance. In a web environment with multiple concurrent users, the backend must reliably maintain this state while handling disconnections, rejoins, and session management. This directly supports the requirement for unique entity names and identifiers across the distributed system.
3. **Asset Delivery Optimization.** The "large library of stock drawings" must be efficiently delivered to web clients.  
     
   Game performance depends on timely delivery of drawing assets that progressively reveal over 30 seconds. Strategies for preloading, caching, and progressive loading must be implemented to ensure smooth gameplay across varying network conditions. This constraint directly impacts core gameplay, as drawings must be fully revealed at precisely the 30-second mark.
4. **Cross-Platform UI Compatibility.** The web interface must function consistently across mobile and desktop browsers.  
     
   As the application transitions from Android-only to multi-platform deployment, the user interface must adapt to different screen sizes and input methods (touch vs. mouse/keyboard). This represents a significant constraint as the UI must be completely redeveloped while maintaining the game's intuitive drawing and guessing mechanics across all platforms.

## [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)

**"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.**

**Class Relationships**

As illustrated by the UML class diagram above, the domain model consists of five primary classes with the following relationships:

* **Entity** (Abstract Base Class)
  + Serves as the foundation for the inheritance hierarchy
  + Contains common attributes (id, name) and methods shared by all entity types
  + Provides basic accessor methods (getId(), getName())
  + There is also a toString() method, which is overridden by subclasses.
* **GameService** (Singleton)
  + Implements the singleton pattern through private constructor and static getService() method
  + Manages the collection of all Game objects
  + Provides methods for game creation, retrieval, and ID generation
  + Maintains unique IDs for games, teams, and players
* **Game** (extends Entity)
  + Inherits from Entity base class
  + Contains a collection of Team objects
  + Provides methods to add and manage teams
  + Ensures team name uniqueness within the game
* **Team** (extends Entity)
  + Inherits from Entity base class
  + Contains a collection of Player objects
  + Provides methods to add and manage players
  + Ensures player name uniqueness within the team
* **Player** (extends Entity)
  + Inherits from Entity base class
  + Represents end users of the application
  + Contains player-specific attributes and behaviors

There are two supporting classes:

* ProgramDriver
  + Application entry point that demonstrates the domain model functionality
* SingletonTester
  + Validates the singleton implementation of GameService

**Object-Oriented Principles Demonstrated**

* **Inheritance.** The diagram shows a clear inheritance hierarchy with Entity as the parent class for Game, Team, and Player. This approach efficiently fulfills the requirement for unique identifiers by centralizing ID and name management in the Entity class. All subclasses inherit these properties and their associated methods, promoting code reuse and consistent behavior.
* **Encapsulation.** All classes demonstrate proper encapsulation through private attributes and public accessor methods. The Entity class hides its default constructor, forcing instantiation through the parameterized constructor that requires ID and name. This ensures all entities are properly initialized and prevents invalid object states.
* **Singleton Pattern.** The GameService class implements the singleton pattern, directly addressing the requirement that "only one instance of the game can exist in memory at any given time." The private constructor and static getInstance() method ensure that only one GameService object is created throughout the application's lifecycle, providing centralized control over game creation and management.
* **Composition.** The domain model uses composition to represent the "has-a" relationships between classes. Games contain Teams, and Teams contain Players, represented by the collection attributes in each class. This hierarchical structure directly supports the requirement that "a game will have the ability to have one or more teams involved" and "each team will have multiple players assigned to it."
* **Identity and Uniqueness Management.** The model enforces the requirement that "game and team names must be unique" through validation logic in the addGame(), addTeam(), and addPlayer() methods. Each method first checks if an object with the given name already exists before creating a new one, ensuring uniqueness at each level of the hierarchy.

**Fulfillment of Software Requirements**

The domain model efficiently addresses all client requirements:

1. **Multiple Teams Per Game**: The Game class maintains a collection of Team objects, allowing one or more teams in each game.
2. **Multiple Players Per Team**: The Team class maintains a collection of Player objects, allowing multiple players to be assigned to each team.
3. **Name Uniqueness**: Name uniqueness is enforced at each level through validation before object creation.
4. **Single Game Instance**: The singleton pattern in GameService ensures only one game instance exists in memory at any time.

## [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.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Mac servers offer robust performance with Unix-based stability, but at a premium cost compared to alternatives. macOS Server software has been deprecated, so deployment would likely use standard macOS with web server software installed. Hosting costs include the Apple hardware premium and macOS licensing. While technically capable of hosting web applications, Mac servers are less common in production environments, with limited scalability options compared to Linux or Windows solutions. Support for modern web technologies is excellent, but the cost-benefit ratio doesn't favor Mac for dedicated server deployments. | Linux provides an ideal server-side platform with excellent stability, security, and performance characteristics for web applications. Its open-source nature eliminates licensing costs, allowing The Gaming Room to deploy on various distributions like Ubuntu Server or CentOS without additional software expenses. Linux excels in resource efficiency, making it cost-effective when scaling to accommodate thousands of players. The platform supports all major web server technologies (Apache, Nginx) and application stacks (Node.js, Python, Java). Linux's command-line focus enables powerful automation and efficient remote management, though it requires specialized expertise compared to GUI-focused alternatives. | Windows Server offers a robust, enterprise-grade platform with strong security features and comprehensive management tools through Windows Server Manager. It provides excellent compatibility with .NET technologies and integrates seamlessly with other Microsoft products. However, significant licensing costs apply, including both server licenses and Client Access Licenses (CALs) based on user count, which could become expensive as the player base grows. Windows servers generally require more system resources than Linux alternatives, potentially increasing hosting costs. The platform offers excellent scalability through features like load balancing and clustering, but at higher operational costs compared to Linux solutions. | Mobile devices are not suitable for hosting server-side web applications. Smartphones and tablets lack the processing power, memory, and network infrastructure required for hosting services accessible to thousands of concurrent users. Mobile operating systems (iOS, Android) are designed for client-side consumption rather than server deployment. Additionally, mobile devices face constraints including unstable network connectivity, power limitations, and absence of server software ecosystems. For Draw It or Lose It, mobile platforms should strictly be considered client endpoints, not hosting solutions. |
| **Client Side** | Developing for Mac clients requires minimal additional effort when building a web-based application, as modern browsers like Safari, Chrome, and Firefox adhere to web standards. The primary consideration is testing compatibility with Safari, which sometimes has unique rendering behaviors. Mac users typically keep their systems updated, reducing compatibility issues with older browser versions. The development team should include at least one Mac testing environment, adding moderate cost. Cross-browser testing tools can help ensure consistent experience across Safari and other browsers. Responsive design principles will ensure proper display across various Mac screen resolutions. With proper web standards adherence, Mac-specific development overhead is relatively low. | Client-side development for Linux users presents minimal additional challenges when using web standards. Testing should include Firefox and Chrome on major distributions like Ubuntu, as these are common Linux user choices. Linux users often run cutting-edge browser versions, reducing backward compatibility concerns. The development team should maintain at least one Linux testing environment, which adds minimal cost due to free Linux distributions. The primary consideration is ensuring JavaScript libraries and CSS frameworks are compatible with Firefox and Chrome rendering engines. Since Linux users typically have technical proficiency, they may be more tolerant of minor UI inconsistencies, but a professional product should maintain consistent quality across all platforms. | Windows client support requires testing across multiple browsers, particularly Edge and Chrome. Windows has the largest desktop market share, making thorough testing on this platform essential. Browser version fragmentation can be significant, as many Windows users run outdated browser versions, requiring additional compatibility considerations and polyfills for newer JavaScript features. The development team likely already has Windows development environments, so additional costs are minimal. Testing should verify proper rendering across various screen resolutions and DPI settings, as Windows devices span a wide range of display configurations. Responsive design principles and careful cross-browser testing will ensure a consistent experience for all Windows users. | Mobile client development requires significant additional considerations, including testing across iOS (Safari) and Android (Chrome, Samsung Browser) on various screen sizes and device capabilities. Responsive design must account for touch interactions, smaller screens, and variable network conditions. Testing should include tablet and phone form factors. Development requires either physical device testing labs or cloud testing services, both adding cost. Performance optimization becomes critical, as mobile devices have less processing power and may operate on cellular networks. The team should implement progressive enhancement strategies to ensure core functionality works across all devices while enhancing the experience on more capable ones. Mobile-specific development could increase development time by 20-30%. |
| **Development Tools** | Development for Mac deployment can leverage several powerful tools and languages. Xcode provides an integrated development environment optimized for Mac development, though for web applications, cross-platform IDEs like Visual Studio Code or JetBrains WebStorm are equally effective. Common web development languages include JavaScript (with frameworks like React, Angular, or Vue.js) for client-side, and Node.js, Python (Django, Flask), or Ruby on Rails for server-side components. Mac-specific development benefits from native Terminal access to Unix commands and tools. While Xcode is free, JetBrains products require licensing fees (approximately $150-200 per developer annually). Container technologies like Docker work well on Mac, facilitating consistent development and deployment environments. Most web development tools have excellent Mac support, making it a comfortable platform for developers. | Linux development environments offer comprehensive tool support for web application development. Popular IDEs include free options like Visual Studio Code and Eclipse, or commercial products like JetBrains WebStorm/IntelliJ IDEA ($150-200 per developer annually). The command-line interface excels for server management and deployment automation using bash scripting. Key languages and frameworks include JavaScript/TypeScript (with Node.js, React, Angular, Vue.js), Python (with Django, Flask), and Java (with Spring). Docker and Kubernetes are particularly well-supported for containerization and orchestration. Git integration is seamless for version control. The LAMP stack (Linux, Apache, MySQL, PHP) remains popular for web hosting. Most development tools are free and open-source, significantly reducing licensing costs. Linux offers an ideal environment for both development and production deployment, creating a consistent pipeline. | Windows provides excellent support for web development through various tools and languages. Visual Studio offers comprehensive features for web development, with versions ranging from free (Community Edition) to enterprise ($1,199-$5,999 per developer). Visual Studio Code provides a popular free alternative. For web applications, developers typically use JavaScript/TypeScript with frameworks like React, Angular, or Vue.js, alongside backend technologies such as ASP.NET, Node.js, or PHP. Windows Subsystem for Linux (WSL) allows access to Linux tools within Windows. IIS (Internet Information Services) is built into Windows for local testing. Azure DevOps integration streamlines deployment to Microsoft's cloud services. SQLServer Management Studio facilitates database development. While many tools are free or have free tiers, enterprise-grade Microsoft tools can add significant licensing costs. | Mobile development tools focus on client-side implementations tailored to various devices. For native development, Android Studio (free) is used for Android development with Java or Kotlin, while Xcode (free, requires Mac) enables iOS development with Swift or Objective-C. For web-based applications, responsive frameworks like Bootstrap, Tailwind CSS, or Foundation are essential. Tools like Xamarin ($1,199-$5,999 as part of Visual Studio) or React Native (free) enable cross-platform development. Browser simulation and testing platforms such as BrowserStack ($29-$399/month) allow testing across multiple devices. Progressive Web App (PWA) technologies enable web applications to function similarly to native apps, using service workers and manifests. Mobile development requires additional expertise in responsive design patterns, touch interactions, and platform-specific behaviors, potentially necessitating specialists on the development team. |

## Recommendations

## Operating Platform

After careful analysis of The Gaming Room's requirements for the Draw It or Lose It web migration, we recommend **Linux** as the optimal server-side operating platform. This recommendation is based on several key considerations:

* **Cost Efficiency**: Linux's open-source nature eliminates licensing costs, which is particularly important as the game scales to accommodate thousands of players. Unlike Windows Server with its CAL licensing model or Mac with its hardware premium, Linux provides enterprise-grade capabilities without ongoing licensing expenses.
* **Resource Optimization**: Linux's lightweight architecture requires fewer system resources than Windows alternatives, allowing for more efficient hardware utilization and reduced hosting costs. This efficiency becomes increasingly valuable as player numbers grow.
* **Java Compatibility**: As the existing Draw It or Lose It codebase is written in Java, Linux offers excellent Java runtime support, making the migration of backend services straightforward.
* **Scalability**: Linux excels in containerization and cloud deployment scenarios, providing flexible scaling options as user demand fluctuates.
* **Stability and Security**: Linux distributions like Ubuntu Server or CentOS offer industry-leading uptime and robust security models, critical for maintaining uninterrupted gameplay experiences.

## Operating Systems Architectures

The recommended Linux platform features a modular architecture uniquely suited to hosting web-based gaming applications:

* **Kernel Architecture**: The Linux kernel sits at the core, managing hardware resources, process scheduling, and system calls. This architecture separates the kernel (running in privileged mode) from user applications (running in user mode), creating a stable foundation for the Draw It or Lose It services.
* **File System Hierarchy**: Linux implements a unified hierarchical file system that abstracts hardware details, allowing consistent file operations regardless of the underlying storage technology. This standardization simplifies deployment across different hosting environments.
* **Process Management**: Linux utilizes lightweight process creation and efficient inter-process communication mechanisms, which are essential for handling multiple concurrent game sessions and maintaining the required singleton pattern for game instances.
* **Device Independence**: The architecture provides hardware abstraction, enabling deployment across various server configurations without code modifications.
* **Multi-User Design**: Built from the ground up for concurrent multi-user access, Linux's architecture naturally aligns with web-based multiplayer gaming requirements.

## Storage Management

For the Draw It or Lose It migration, we recommend implementing a hybrid storage management system:

* **Primary Game Data**: PostgreSQL database for structured data storage, including game states, user profiles, team configurations, and session information. PostgreSQL offers ACID compliance, robust transaction support, and excellent performance characteristics for the concurrent access patterns of multiplayer gaming.
* **Drawing Assets**: A combination of server-side file storage for the master drawing library and CDN (Content Delivery Network) distribution for client-side delivery. This approach optimizes the delivery of the "large library of stock drawings" mentioned in the design constraints.
* **Caching Layer**: Redis in-memory data store for high-speed access to frequently used game data, particularly important for maintaining the 60-second round timing and drawing reveal schedules.
* **File System Organization**: Implementation of separate storage partitions for system files, application code, databases, and user-generated content, allowing for targeted backup strategies and security policies.

This hybrid approach addresses both the performance requirements for time-sensitive gameplay and the scalability needs for accommodating growing user numbers.

## Memory Management

The Linux operating system employs several memory management techniques that will benefit Draw It or Lose It:

* **Virtual Memory Subsystem**: Linux's virtual memory system creates an abstraction layer that allows the application to operate as if it has more memory than physically available. This is particularly important for maintaining the game's singleton pattern, as it enables efficient memory utilization across multiple user sessions.
* **Process Isolation**: Each game instance runs in its own protected memory space, preventing errors in one session from affecting others. This isolation is critical for maintaining stable gameplay experiences.
* **Dynamic Memory Allocation**: The Java runtime environment on Linux provides efficient garbage collection and memory management, automatically reclaiming memory from objects no longer in use. This addresses potential memory leaks in long-running game sessions.
* **Shared Libraries**: Common code components can be loaded once and shared across multiple processes, reducing the overall memory footprint of concurrent game sessions.
* **Swapping Management**: Linux intelligently moves less frequently accessed memory pages to disk when physical memory is constrained, maintaining performance for active gameplay sessions.

These memory management techniques ensure that Draw It or Lose It can maintain stable performance even during peak usage periods.

## Distributed Systems and Networks

To enable Draw It or Lose It's cross-platform functionality, we recommend implementing a distributed architecture with the following components:

* **RESTful API Layer**: Implementing a standardized REST API for non-time-critical operations like user authentication, game setup, and score retrieval. This provides consistent access methods across different client platforms.
* **WebSocket Communication**: Using WebSocket protocol for real-time bidirectional communication during active gameplay, ensuring synchronized drawing reveals and timely updates across all connected clients.
* **Microservices Architecture**: Breaking the server functionality into discrete services (authentication, game management, asset delivery) to allow independent scaling and maintenance of different system components.
* **Load Balancing**: Implementing a load balancer in front of server instances to distribute client connections evenly and provide failover capabilities.
* **Redundancy Planning**: Deploying redundant server instances across different availability zones to mitigate the impact of regional outages.
* **Network Latency Mitigation**: Implementing predictive algorithms and client-side interpolation to compensate for network latency, particularly important for the time-sensitive aspects of gameplay.
* **Offline Capability**: Developing limited offline functionality in the web client, allowing users to resume gameplay when connectivity is restored.

This distributed approach ensures reliable communication between various platforms while maintaining a consistent user experience despite potential connectivity challenges.

## Security

To protect user information and ensure game integrity, we recommend implementing a comprehensive security strategy:

* **Transport Layer Security**: Mandating HTTPS/TLS 1.3 for all client-server communication, encrypting data in transit and preventing man-in-the-middle attacks.
* **Authentication Framework**: Implementing OAuth 2.0 with JWT (JSON Web Tokens) for secure authentication across platforms, allowing integration with popular identity providers while maintaining centralized access control.
* **Data Encryption**: Encrypting sensitive user data at rest in the PostgreSQL database using AES-256 encryption, with proper key management protocols.
* **Linux Security Modules**: Utilizing Linux's built-in security modules (SELinux or AppArmor) to implement mandatory access controls, limiting potential damage from security breaches.
* **Regular Security Updates**: Establishing automated security patching for the Linux server environment, addressing vulnerabilities promptly.
* **Web Application Firewall**: Deploying a WAF to filter and monitor HTTP traffic, protecting against common web attacks like SQL injection, cross-site scripting, and DDoS attempts.
* **Rate Limiting**: Implementing API rate limiting to prevent abuse and resource exhaustion attacks.
* **Security Auditing**: Configuring comprehensive logging and monitoring to detect unusual patterns and potential security incidents.

The recommended Linux platform provides robust built-in security capabilities that, when properly configured, will protect both user information and game integrity across all supported platforms.