

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

*<* Please note: While this section is not being assessed, it will support your outline of the design constraints below. *In your summary, identify each of the client’s business and technical requirements in a clear and concise manner.>*

## [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.

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** | <Evaluate Mac for its characteristics, advantages, and weaknesses for hosting a web-based software application.> | <Evaluate Linux for its characteristics, advantages, and weaknesses for hosting a web-based software application.> | <Evaluate Windows for its characteristics, advantages, and weaknesses for hosting a web-based software application.> | <Evaluate Mobile Devices for their characteristics, advantages, and weaknesses for hosting a web-based software application.> |
| **Client Side** | <Determine the software development considerations (cost, time, expertise) that are necessary for supporting multiple types of clients as they pertain to Mac.> | <Determine the software development considerations (cost, time, expertise) that are necessary for supporting multiple types of clients as they pertain to Linux.> | <Determine the software development considerations (cost, time, expertise) that are necessary for supporting multiple types of clients as they pertain to Windows.> | <Determine the software development considerations (cost, time, expertise) that are necessary for supporting multiple types of clients as they pertain to Mobile Devices.> |
| **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.> |

## 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**: <Recommend an appropriate operating platform that will allow The Gaming Room to expand Draw It or Lose It to other computing environments.>
2. **Operating Systems Architectures**: <Describe the details of the chosen operating platform architectures.>
3. **Storage Management**: <Identify an appropriate storage management system to be used with the recommended operating platform.>
4. **Memory Management**: <Explain how the recommended operating platform uses memory management techniques for the Draw It or Lose It software.>
5. **Distributed Systems and Networks**: <Knowing that the client would like Draw It or Lose It to communicate between various platforms, explain how this may be accomplished with distributed software and the network that connects the devices. Consider the dependencies between the components within the distributed systems and networks (connectivity, outages, and so on).>
6. **Security**: <Security is a must-have for the client. Explain how to protect user information on and between various platforms. Consider the user protection and security capabilities of the recommended operating platform.>