

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc199073985)

[Table of Contents 2](#_Toc199073986)

[Document Revision History 2](#_Toc199073987)

[Executive Summary 3](#_Toc199073988)

[Requirements 3](#_Toc199073989)

[Design Constraints 3](#_Toc199073990)

[System Architecture View 4](#_Toc199073991)

[Domain Model 4](#_Toc199073992)

[Evaluation 6](#_Toc199073993)

[Recommendations 8](#_Toc199073994)

## [Document Revision History](#_grjogdjh5fi8)

|  |  |  |  |
| --- | --- | --- | --- |
| **Title** | **Name** | **Date** | **Notes** |
| Lead Developer | Spencer Reed | 5/8/25 | Draft |
| Lead Developer | Spencer Reed | 5/22/25 | Added UML diagram |
| Lead Developer | Spencer Reed | 6/5/25 | Added analysis of differing environments – server & client side |
| Lead Developer | Spencer Reed | 6/22/25 | Finalized recommendations |

## [Executive Summary](#_sbfa50wo7nsh)

Creative Technology Solutions (CTS) has been engaged by The Gaming Room to develop a web-based version of their existing Android game, *Draw It or Lose It*. The new version must be accessible via modern web browsers and maintain core gameplay elements: multiple teams guessing what is being drawn in real time from a library of stock images.

To meet this objective, CTS proposes a Java-based solution using a web application framework, in partnership with front-end technologies such as HTML, CSS, and JavaScript to allow scalability, cross-platform compatibility, and maintainability.

The solution must meet the objectives:

* Unique identification of game, team, and player instances
* Only one active game in memory
* Real-time gameplay synchronized across multiple clients
* Centralized control of timing, game flow, and user sessions

CTS will also provide guidance for setting up the Java development environment, including server hosting, tools for building, and a relational database for storing game metadata.

## Requirements

The following are the business and technical requirements respectively:

**Business Requirements**

* + Transition from mobile-only to browser-accessible experience
  + Allow multiple teams and players per game
  + Promote wider adoption through web access
  + Ensure intuitive game creation and participation process

**Technical Requirements**

* + Web-based game with Java backend
  + Single instance of the game in memory at any time
  + Game, team, and player names must be unique
  + Rounds are timed: 1 minute each, full image shown at 30 seconds
  + If active team fails to guess, opposing teams have 15 seconds to respond
  + Players join teams within games, and data must persist across sessions
  + Backend must handle game state, timing, and player actions
  + Frontend must interact with the Java backend via RESTful APIs

## [Design Constraints](#_2et92p0)

<Identify the design constraints for developing the game application in a web-based distributed environment and explain the implications of the design constraints on application development.>

1. Single Game Instance (Memory Constraint)

**Constraint:** Only one active game can exist in memory at a time

**Implication:** Implement a singleton pattern to ensure thread-safe handling of a single game object across all sessions. Proper synchronization must be in place to avoid concurrency issues.

1. Unique Naming for Entities

**Constraint:** Game, team, and player names must be unique

**Implication:** Enforce uniqueness through validations of unique identifiers for each game, team, and player. Unique indexes should exist for each instance on the backend database and be accessed in the service layer and database-level

1. Real-Time Timers and Game Flow

**Constraint:** Image rendering and guesses are time-dependent

**Implication:** Requires use of Java’s *ScheduledExecutorService* to manage countdowns and enforce time-based triggers (e.g., when to reveal images or switch turns)

1. Web-based Access (Browser Platform)

**Constraint:** Must be browser-compatible and mobile-responsive

**Implication:** The backend will expose RESTful endpoints, and client-side JavaScript will consume them using AJAX

1. Distributed Multiplayer

**Constraint:** Players connect from different locations/devices

**Implication:** Must use session management to track active users and their associations.

1. Image Rendering from Stock Library

**Constraint:** Drawings must come from a predefined library

**Implication:** Store images on the server with metadata stored in a relational database. Ensure efficient retrieval and sequencing based on the game clock

## [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.**

The domain model uses a class structure that models the real-world entities of the game, including games, teams, players, and service logic. The relationships are as follows:

* **Entity**: This is the base class for Game, Team, and Player and it encapsulates share attributes (id, name) and behaviors (getId(), getName(), toString()).
* **GameService** is a *singleton* class that manages a list of Game objects and provides methods to add, retrieve, and manage games, teams, and players.
* **Game** contains a list of Team objects (0..\*), and each team belongs to a specific game.
* **Team** contains a list of Player objects (0..\*), and each player belongs to a specific team.
* **ProgramDriver** and **SingletonTester** are utility classes for initiating and testing the singleton pattern in **GameService**

This class structure captures the real-world relationships:

* A game has many teams
* A team has many players
* All share a common identity structure through inheritance

This structure also demonstrates key Object-Oriented Programming principles:

1. **Encapsulation** – Each class encapsulates its own data and methods
   1. Fields like *games, teams,* and *players* are kept private and accessed/manipulated by public methods
   2. Promotes data integrity and security
2. **Inheritance** – The **Entity** class is a base class for **Game, Team,** and **Player**
   1. Common attributes and behaviors (id, name, getId(), getName()) are inherited, reducing code duplication
   2. Enables polymorphic handling of game entities as needed
3. **Abstraction** – The diagram focuses on relevant fields & methods
   1. Internal implementation details are *abstracted*
   2. Simplifies the complexity and keeps the game design modular
4. **Singleton Pattern** – The **GameService** class implements the singleton design pattern
   1. getInstance() ensures only one instance of the service exists, enforcing the client’s requirement that only one game instance can be in memory at a time
   2. Centralizes the control of game state and prevents inconsistencies
5. **Composition / Aggregation**
   1. A **Game** is composed of multiple **Teams**, while each **Team** is composed of multiple **Players**
   2. These “has-a” relationships reflect real-world structures and enable hierarchical control and lifecycle management

This model fulfills the requirements efficiently through the use of a singleton pattern to ensure only one single active game at a time. With the usage of unique IDs and Names, all entities inherit from **Entity**, which ensures consistent ID and name tracking. Additionally, the use of the List object allows multiple teams per game and multiple players per team. All of these come together to provide scalability of the game, enabling the game to be extended to support game rules, and rendering logic without disrupting the core domain model.

## [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** | Suitable for development; less ideal for production due to licensing and limited server use | Preferred for hosting Java apps; secure, stable, and widely supported in cloud environments | Viable but heavier and more costly; less optimized for open-source Java stacks | Not suitable for hosting; limited processing power and connectivity |
| **Client Side** | Higher cost due to Apple hardware; requires separate responsive design for Safari | Low cost, but low user base; ensure compatibility with Firefox and Chrome | Most widely used desktop platform; requires broad browser testing | Requires responsive UI and touch support; increases testing score |
| **Development Tools** | Supports many tools such as Eclipse for development; Good for local development, but not identical to Linux servers | Excellent for Java development; many natively-running tools available | Strong IDE support; differs from Linux production environment; Some configuration needed for additional setup | Focus would be on testing responsiveness; Chrome devtools and emulators essential |

## Recommendations

1. **Operating Platform**: It’s our recommendation to deploy *Draw It or Lose It* on a Linux-based server environment as this is widely supported, secure, lightweight, and cost-effective, making it ideal for scalable web application deployment. It is highly compatible with Java-based backends and is the industry standard for hosting web apps.
2. **Operating Systems Architectures**: Linux follows a modular, layered architecture that supports multitasking, multithreading, and strong memory/process isolation. Linux runs very well on both physical and virtual machines, and its open-source nature allows configuration tailored to the application’s resource needs. It also supports modern cloud platforms which enable easy expansion and platform independence.
3. **Storage Management**: We recommend to use a relational database such as MySQL for structured data along with cloud-based object storage (e.g., Amazon S3) to store static assets like stock drawing images. This approach separates transactional data from static files, ensuring performance and scalability.
4. **Memory Management**: Linux utilizes virtual memory management with paging and demand loading in order to optimize RAM usage. The Java Virtual Machine (JVM) running on Linux also provides robust garbage collection and heap management. These ensure that only the active game instance is kept in memory and memory is reclaimed efficiently as objects are no longer needed.
5. **Distributed Systems and Networks**: *Draw It or Lose It* can support communication across platforms using a RESTful API and WebSocket architecture. REST handles standard requests (joining teams, submitting answers), while WebSockets enable real-time updates. These components rely on consistent network connectivity, so in order to mitigate outages and dependencies, we recommend implementing graceful error handling, retry logic, and hosting in a cloud environment with redundancy and load balancing.
6. **Security**: In order to protect user information we recommend to:
   1. Use HTTPS to encrypt all data in transit
   2. Store sensitive data securely using hashed passwords & database access controls
   3. Implement authentication tokens for session management
   4. Secure APIs with role-based access control and input validation to prevent injection attacks
   5. Ensure server environments are hardened and regularly updated, with firewalls and intrusion detection systems in place

Linux’s built-in security features make it a solid foundation for deploying secure applications

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Development Requirements | Mac | Linux | Windows | Mobile Devices |
| Server Side | macOS is suitable for development environments but less common for production server hosting due to licensing limitations and lower adoption in enterprise hosting. It can be used for local testing but is not typically recommended for deployment at scale. | Linux is the preferred platform for hosting Java-based web applications. It is secure, lightweight, stable, and widely supported in cloud environments. Linux is highly scalable and ideal for production deployment. | Windows servers are viable but tend to incur higher licensing costs. They are heavier and may require additional configuration for Java-based stacks. Not commonly used in cloud-native Java deployments. | Mobile devices are not suitable for server hosting due to limited resources, inconsistent connectivity, and OS limitations. They are client devices and should not be considered for hosting the backend. |
| Client Side | macOS has a smaller market share and higher hardware cost. Development must ensure compatibility with Safari and various screen sizes. Responsive design is critical. | Linux desktops have a low user base but require compatibility with browsers like Firefox and Chrome. Emphasis should be on standards-compliant HTML, CSS, and JS to ensure functionality. | Windows is the most widely used desktop platform. Applications must be tested thoroughly across major browsers (Chrome, Edge, Firefox) to ensure cross-browser compatibility. | Mobile platforms require a responsive user interface, touch input support, and compatibility with mobile browsers on Android and iOS. Extensive testing across device sizes and orientations is essential. |
| Development Tools | macOS supports a variety of development tools including Eclipse and IntelliJ IDEA. It’s suitable for local development and offers a smooth developer experience but differs from most production environments. | Linux has strong support for Java development and many tools run natively. Developers can build and test in an environment similar to production, ensuring fewer discrepancies. | Windows provides powerful IDEs like Visual Studio and IntelliJ IDEA. However, differences from Linux-based production environments may require configuration adjustments. | Development focuses on testing responsiveness using browser dev tools, mobile emulators (e.g., Android Studio, Xcode), and frameworks like Bootstrap. Native mobile tools may be used for edge case testing or performance validation. |