

<Name-of-Software-Application>

# **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 | 05/22/25 | Alexis Contreras | Initial draft for Software Design |

**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 has tasked CTS with redeveloping their Android-only app, Draw It or Lose It, into a scalable, web-based game. The original game involves players guessing images from a stock drawing library as they are rendered on-screen in real-time. The client has expressed uncertainty about how to set up the development and operating environment, which is why they brought CTS on board.

The new game must serve multiple platforms and support several teams per game, with each team consisting of multiple players. Every game, team, and player must have a unique name and ID, and the app should prevent duplicate names at the time of creation. Additionally, there must be only one game instance in memory at any time.

This design proposes a client-server architecture with a robust backend that supports scalable play across various devices, including desktops and mobile platforms. It will also account for critical aspects of platform design, including memory constraints, security infrastructure, and rendering performance (Silberschatz et al., 2018).

## Requirements

Business Requirements:

* Make the game accessible on the web, not just Android.
* Allow more than one team in a single game.
* Make sure each team and game has a unique name.

Technical Requirements:

* Use a Singleton pattern so there’s only one version of the game running in memory.
* Use the Iterator pattern to make sure no two teams or games have the same name.
* Use object-oriented design to keep the code organized and reusable.

## [Design Constraints](#_2et92p0)

Converting Draw It or Lose It from a mobile-only app to a web-based platform introduces several important system-level design constraints. First, the application will require separate environments for development, testing, and production. These environments should be cloud-based and scalable so the game can grow with user demand. For example, performance testing and future updates will need dedicated servers, and a production environment must be capable of handling increased traffic if the game gains popularity (Amazon Web Services, 2024).

Second, the game will need more robust memory and storage management. Since images will now be hosted and rendered on the server side instead of stored locally on a device, the system must be optimized to handle large image files efficiently and render them at a consistent speed. This means choosing appropriate storage solutions, like a cloud-based database or CDN, and ensuring enough memory is allocated for performance stability.

Security is another critical consideration. The Android version relied on the device’s built-in authentication, such as a Gmail login, but the web version will need a fully secure login system, including password hashing, session tokens, and role-based access. We also need to prevent users from logging into the same account across multiple devices simultaneously. Finally, as the web version will support more complex features like multiple teams and administrative control, the system must include an admin interface for managing users, teams, and game sessions securely (OWASP Foundation, 2023).

## [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 UML class diagram shows how the game is structured using a base class called Entity, which holds shared properties like the ID and name for each object. Classes such as Game, Team, and Player all extend from Entity, which keeps the design cleaner and avoids repeating the same code.

This design satisfies the client’s main requirements in a few key ways. By using inheritance, every class automatically gets a unique ID and name, helping ensure that no two games, teams, or players are created with duplicate values. Encapsulation is used to keep the ID and name fields private and only accessible through public getter methods, which protects the data from being changed accidentally. Abstraction is shown in how the Entity class represents common features of all game elements without needing to know their specific behaviors. Polymorphism comes into play with methods like toString() and getName() — each subclass can override these to behave in a way that makes sense for that specific type of object. Following principles of robust object-oriented design outlined by Sommerville (2016), using inheritance, encapsulation, and polymorphism ensures our domain model remains modular and maintainable as the game scales and new features are added.We also use the Singleton pattern in the GameService class to make sure only one instance of the game management system exists in memory at a time, which lines up with the client’s requirement of having a single active game session. The Iterator pattern is used to check for duplicates when adding new games, teams, or players, so everything stays unique and consistent. Overall, the domain model uses solid OOP principles to meet the client’s expectations while keeping the system organized and scalable.

**"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)

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** | Mac is good for development but not ideal for hosting servers. Its more expensive and doesn’t have hosting options. | Linux is probably the best pick for server-side stuff. It’s open source, lightweight, and most web servers are already built on Linux (Silberschatz et al., 2018). | Windows servers are okay but cost more and aren’t as common for hosting we app the also need more resources to run smoothly | Mobile devices aren’t bult to run servers, so they are not an option here. |
| **Client Side** | Supporting Mac users means making sure everything works on Safari and MacOS. Testing might take extra time, but its manageable. | Linux users are usually more tech savvy, so we need to make sure the browser based game works in Firefox or Chrome. | Windows is the most common OS, so it’ll need full support. Most tools and browsers are windows friendly. | Must make sure the web game is responsive so it works well on mobile screens. Most users will probably play from their phones anyway (Amazon Web Services, 2024). |
| **Development Tools** | Mac can run IDEs like Eclipse. Java and HTML/CSS/JavaScript are all supported, making it a solid choice for cross-platform web development (Sommerville, 2016). | Linux supports most tools we need, including VS Code, Eclipse and Java. It’s a great Dev environment | Windows supports all the major IDEs and is good for testing across different browsers and setups. | For mobile, browser testing tools like BrowserStack can ensure responsive layouts, and Android Studio or Xcode will be essential if we expand to native apps (National Institute of Standards and Technology, 2022). |

## 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 recommend Linux for hosting the web-based game. It’s reliable, cheap, and most of the internet already runs on it (Silberschatz et al., 2018).
2. **Operating Systems Architectures**: The Linux architecture we’ll use supports multitasking and multi-user setups, which is perfect for a web game where lots of people might be playing at once. It’s built for stability, and it handles resources really well (Silberschatz et al., 2018).
3. **Storage Management**: We should use a cloud-based database like MySQL or PostgreSQL, hosted on something like AWS or DigitalOcean, which are fast, scalable, and work great with Java web apps (Amazon Web Services, 2024).
4. **Memory Management**: Linux uses a smart memory management system that handles caching and swapping efficiently. That’ll help our game stay responsive and handle multiple users without crashing. Also, since we’re using the Singleton pattern, we avoid extra memory being used by multiple game instances.
5. **Distributed Systems and Networks**: To make the game work across devices, we can use a REST API on the backend that handles all the game logic and user interactions. The frontend will make HTTP requests to this backend. This way, people can play from phones, tablets, or computers as long as they have internet. We’ll need to make sure the system can handle temporary network outages and reconnect smoothly when needed.
6. **Security**: In addition to secure coding practices and HTTPS, adopting a structured cybersecurity framework, such as the one provided by the National Institute of Standards and Technology (2022), will help systematically identify, protect, detect, respond to, and recover from potential security incidents.

**References**:  
Amazon Web Services. (2024). AWS architecture center. https://aws.amazon.com/architecture/

OWASP Foundation. (2023). OWASP top ten: The ten most critical web application security risks. https://owasp.org/www-project-top-ten/

Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating system concepts (10th ed.). Wiley.  
  
Sommerville, I. (2016). Software engineering (10th ed.). Pearson.

National Institute of Standards and Technology. (2022). Framework for improving critical infrastructure cybersecurity. https://www.nist.gov/cyberframework