

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.2

## Table of Contents

[CS 230 Project Software Design Template 1](#_Toc1075004988)

[Table of Contents 1](#_Toc1737842232)

[Document Revision History 2](#_Toc1932238621)

[Executive Summary 2](#_Toc36477731)

[Requirements 3](#_Toc91811914)

[Design Constraints 3](#_Toc488258651)

[System Architecture View 3](#_Toc2125549134)

[Domain Model 3](#_Toc1432483354)

[Evaluation 4](#_Toc1528982840)

[Recommendations 4](#_Toc1877931273)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 05/26/2024 | Rob McGuine | Completed Executive Summary, Design Constraints, and Domain Model. Ready for review. |
| 1.1 | 06/09/2024 | Rob McGuine | Completed Evaluation table outlining Server, Client, and Development Tool considerations. Ready for review. |
| 1.2 | 06/22/2024 | Rob McGuine | Completed “Recommendations” section outlining the Operating Platform, Operating Systems Architectures, Storage Management, Memory Management, Distributed Systems and Networks, and Security. |

**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)

Creative Technology Solutions (“CTS”) is tasked with developing a web-based version of The Gaming Room's existing Android game, “Draw It or Lose It”. This new version will enable gameplay across multiple platforms, thereby expanding its accessibility and user base. The game will maintain its core mechanics, allowing teams to compete in guessing puzzles rendered from a library of stock drawings. Key software requirements include the ability to manage multiple teams and players, enforce unique names for games and teams, and ensure that only one game instance exists in memory at any time. To achieve these goals, CTS will employ modern web development practices, creating a robust and scalable architecture that supports the seamless integration of these features. This document outlines the proposed solution, providing the necessary details to guide the development process.

## 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)

Developing the game application in a web-based distributed environment presents several design constraints, which have significant implications on the application development process:

* **Scalability**: The application must efficiently handle varying numbers of users and teams, which requires scalable architecture.
  + Implication: The system should be designed using microservices or cloud-based solutions to ensure it can scale horizontally, handling increased load without degradation of performance.

* **Concurrency**: Only one instance of the game can exist in memory at a given time.
  + Implication: This necessitates the use of synchronization mechanisms and potentially a distributed cache or in-memory database like Redis to manage game state and ensure consistency across instances.

* **Real-time Data Processing**: The game involves real-time rendering of images and user interactions.
  + Implication: The application must be optimized for low latency and real-time data processing, possibly utilizing WebSockets or similar technologies to maintain a persistent connection between clients and the server.

* **Security**: Ensuring unique names for games and teams implies a need for secure authentication and data validation processes.
  + Implication: The system must implement robust security measures, including HTTPS, secure user authentication, and validation routines to prevent conflicts and unauthorized access.

* **Cross-Platform Compatibility**: The game should function seamlessly across different platforms and devices.
  + Implication: The front-end should be developed using responsive design principles and frameworks that support cross-platform compatibility, such as React or Angular, and ensure consistent user experience across devices.

* **Data Storage and Management**: The requirement to manage multiple teams and players involves complex data relationships.
  + Implication: The database design should be normalized and optimized for quick retrieval and updates, using relational databases like PostgreSQL or MySQL, possibly supplemented by NoSQL solutions for specific use cases.

* **Unique Identifiers**: The necessity for unique game, team, and player identifiers.
  + Implication: This requires careful planning of the identifier generation strategy, potentially leveraging UUIDs or a similar mechanism to ensure uniqueness across the distributed environment.

By addressing these constraints, CTS can develop a robust, efficient, and user-friendly web-based game application that meets The Gaming Room's requirements and provides an engaging experience for players across various platforms.

v

## [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)

**Classes**:

***Entity***: This is an abstract base class used as the Super parent. It contains common attributes and methods that are inherited by its subclasses (classes that derive from it). The diagram doesn't show any attributes or methods within the Entity class itself, but it suggests that subclasses inherit functionalities from Entity.

***ProgramDriver***: This class is the main driver class for the application. It has a *main()* method, the entry point of the program. It also has a method named *testSingleton()*, which suggests potential usage of the Singleton design pattern (more on that later).

***GameService***: This class manages the game logic and services. It has methods for adding and retrieving games, along with potentially managing the game state or interactions.

***Player***: This class represents a player in the game. It holds information about the player and potentially their actions or game state.

***Team***: This class represents a team within the game. It holds information about the team and potentially its members or game state.

**Relationships**:

***Inheritance***: The arrow pointing from Player and Team to Entity indicates inheritance. This means that Player and Team classes inherit attributes and methods from the Entity class.

***Aggregation***: The diamond symbol next to the games attribute in GameService signifies an aggregation relationship. This means that a GameService object "has a" collection of Game objects. The GameService object manages the lifecycle of the Game objects it holds, but the Game objects themselves are separate entities.

**Object-Oriented Principles**:

***Encapsulation***: The classes encapsulate their data (attributes) and functionality (methods) within themselves. This promotes data protection and modularity.

***Inheritance***: By using inheritance, the Player and Team classes can reuse common functionalities from the Entity class without code duplication. This promotes code reusability and maintainability.

***Aggregation***: The aggregation relationship between GameService and Game allows for managing collections of games within the GameService object. This promotes modularity and easier management of game objects.

***Singleton Pattern***: The *testSingleton()* method in ProgramDriver suggests use of the Singleton design pattern. This pattern ensures only one instance of a class exists, which can be useful for managing game states or global configurations.

**"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** | While Mac solutions can be used for server-side deployments using macOS Server, it is not commonly used in large-scale web hosting. Going with a macOS Server solution will come with a high initial cost due to hardware requirements, as well as macOS Server licensing.  **Characteristics**: Smooth integration with other Apple products, reliable.  **Advantages**: Great for development with a strong ecosystem for design and creative applications.  **Weaknesses**: Higher cost, less common for server deployments. | Linux provides some of the highest support for server-based deployments. Widely used web server deployments include Nginx and Apache, as well as many various application servers. Linux solutions are usually free and open-source, leading to low – or no – licensing costs. Enterprise licensing may be required for commercial use and elevated support.  **Characteristics**: Highly stable, flexible, and secure. Excellent community support.  **Advantages**: Cost-effective, robust, and scalable. Ideal for hosting web servers.  **Weaknesses**: May require more technical expertise for setup and maintenance. | Windows provides similar market share and support as Linux solutions for server-based deployments, especially in web-based (IIS) applications. Windows Server solutions are more commonly used in enterprise environments and come with a higher burden of licensing costs and management. Typical license costs include Windows Server, and potentially CALs (Client Access Licenses).  **Characteristics**: User-friendly, widely used in business environments.  **Advantages**: Strong support for enterprise applications, familiar to many developers.  **Weaknesses**: Licensing costs, potential for higher overhead. | Hosting directly on mobile devices is not typical for web applications. Mobile devices access web applications hosted on traditional server platforms. Mobile devices usually rely on backend servers (Linux, Mac, Windows) for data and services. Servers must be optimized for handling requests from mobile clients efficiently. This can allow for updates to be handled centrally on a backend server that clients access, thus increasing performance, security, and reliability. |
| **Client Side** | Requires ensuring compatibility with various browsers like Google Chrome and Firefox. However, with Safari being the default browser, thorough testing will be required for Safari support, adding to extra support work. Expertise in web technologies (HTML, CSS, JavaScript, responsive design). Tools like WebKit, Gecko for testing. | Similar to Mac, Linux requires ensuring compatibility with various browsers like Firefox and Google Chrome. Web apps typically run well. Expertise in web technologies (HTML, CSS, JavaScript, responsive design). Tools like WebKit, Gecko for testing. | Broad browser usage, including Edge, Chrome, and Firefox. Requires extensive cross-browser testing. Same web technologies expertise. Tools like BrowserStack for cross-browser testing can be helpful. | **iOS**: Need to ensure compatibility with Safari on iOS, responsive design critical for mobile experience. Expertise in mobile web development. Tools like Xcode for iOS-specific testing.  **Android**: Need to ensure compatibility with various Android browsers (Chrome primarily). Responsive design essential. Similar expertise in mobile web development. Tools like Android Studio for Android-specific testing.  **Characteristics**: Essential for reaching a wide mobile audience.    **Advantages**: Extensive user base, especially on Android. iOS users typically engage more with apps.    **Weaknesses**: Fragmentation in Android devices, strict App Store guidelines for iOS. |
| **Development Tools** | Languages that would be required for development on macOS: Swift (for native), JavaScript.  Tools that would be required for development for macOS: Xcode, VS Code.  One consideration for development is needing additional expertise for native macOS/iOS apps; both developing and supporting.  From an expense standpoint, Xcode is free, but macOS requires Apple hardware. | Languages that would be required for development on Linux: JavaScript, Python, PHP, Ruby, etc.  Tools that would be required for development for Linux: VS Code, Sublime Text, Eclipse.  One Cross-platform development possible with the same team.  From an expense standpoint, tools are mostly free and open source. | Languages that would be required for development on Windows: C#, JavaScript.  Tools that would be required for development for Windows: Visual Studio, VS Code.  One consideration for development is potentially needing additional expertise in .NET for backend services.  From an expense standpoint, Visual Studio Community is free; however, enterprise versions have costs. | **iOS**:  Languages that would be required for development in iOS: Swift, JavaScript (for web).  Tools that would be required for development for iOS: Xcode.  One consideration for development is potentially needing a separate team or additional skills for Swift development as it’s a “niche” language  From an expense standpoint, Xcode is free; however, it requires Apple hardware for development.  **Android**: Languages that would be required for development in iOS: Java, Kotlin, JavaScript (for web).  Tools that would be required for development for Android: Android Studio, VS Code.  One positive consideration is that similar team skills will be required for web development.  From an expense standpoint, Android Studio is free. |

## 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**: Creative Technology Solutions (“CTS”) recommends using a Linux-based server platform for hosting the “Draw It or Lose It” game application. A Linux-based server platform will be able to deliver a robust, secure, and widely supported solution – for both on-premises and in the cloud – making it an excellent choice for dynamic scalability and flexibility. Linux offers numerous distributions (“flavors”) such as Ubuntu Server, CentOS, and Red Hat Enterprise Linux (“RHEL”), which can cater to different business needs. While flavors such as Ubuntu and CentOS do not require a paid license – even for commercial use – and can be a great option to test a proof-of-concept or MVP for the game, The Gaming Room may want to consider an enterprise option like RHEL for commercial use, especially if they anticipate requiring professional support and guidance with their server operating platform.
2. **Operating Systems Architectures**:
   1. Monolithic Architecture:
      1. In a monolithic architecture, the entire operating system works in kernel mode, providing efficient performance due to minimal context switching. Linux's monolithic architecture includes subsystems for file management, memory management, process scheduling, and networking, all interacting closely within the kernel.
   2. Modular Kernal:
      1. Linux supports loadable kernel modules, allowing dynamic loading and unloading of features as needed without rebooting the system. This modularity adds flexibility and customization for specific hardware and software needs.
   3. Process Management:
      1. Linux uses a process management system based on process IDs (PIDs) and supports multitasking, process creation, and termination, making it suitable for handling multiple simultaneous game sessions efficiently.
3. **Storage Management**: CTS recommends using a combination for Network Attached Storage (“NAS”) and cloud storage services, such as Amazon S3, Google Cloud Storage, or Azure Storage. A NAS solution would provide a centralized, scalable solution for storing game assets and user data, all accessible over the network. In addition, it would facilitate easy backups, redundancy, and data sharing across different servers. Given that The Gaming Room is anticipating 200, 8MB-sized image files, we recommend hosting the image files in the cloud. Cloud solutions will offer high availability, durability, and scalability for storing large volumes of image files. In addition to storing objects, cloud providers will also offer a Content Delivery Network (“CDN”) solution to deliver files quickly to users worldwide. This means that The Gaming Room can store their images in a US datacenter, and the files can be cached in global datacenters closest to where a user may be requesting the file. As of this writing, [Google Cloud offers $0.14/month for 12 GiB of storage (excluding operations)](https://cloud.google.com/products/calculator?hl=en&dl=CiRjNzUzMzBjYS1mYmYxLTRiNTQtOTcwMy03MDFjZWMwNGJiYjEQCRokQkM1ODZCODktMUU1Ni00QUVCLUFCQ0YtMTk0ODBCMEUyNEVG), while [AWS is $0.28/month for 12 GB of storage (excluding operations)](https://calculator.aws/#/estimate?id=025e34f977e0c32d25bc6c6666565a71f7643670), and [Azure is $0.25/month](https://azure.com/e/27cd77672b4e425e84bc99faaf2d2cd9).
4. **Memory Management**:
   1. Virtual Memory:
      1. Linux uses virtual memory to extend physical memory by using disk space, which allows more applications to run simultaneously than the physical RAM would permit.
      2. Ensures that each process operates in its own protected address space, preventing one process from interfering with another.
   2. Paging & Swapping:
      1. Paging allows the system to retrieve data from secondary storage in fixed-size blocks, ensuring efficient use of memory.
      2. Swapping involves moving entire processes to disk when physical memory is low, which can free up RAM for active processes.
   3. Cache Management:
      1. Linux maintains a cache to store frequently accessed data in memory, reducing the need to access slower storage devices.
      2. Utilizes a least recently used (LRU) algorithm to manage cache contents, ensuring that the most frequently accessed data remains in memory.
5. **Distributed Systems and Networks**:
   1. Distributed Software:
      1. One consideration for the design would be to incorporate a microservices architecture, where each component (game logic, user management, image rendering, etc.) runs as a separate service. This would allow for quicker updates to specific parts of the application, which would allow for more features and support across platforms. Additionally, the services can communicate over a network using protocols such as HTTP/HTTPS and gRPC, which would further allow for platform-agnostic communication.
   2. Network Interconnectivity:
      1. RESTful APIs: RESTful APIs will enable communication between different services and planforms using standard web protocols.
      2. WebSockets: WebSockets will facilitate real-time communication between the clients and the servers, which will be essential for interactive gameplay.
      3. Load Balancers: Load Balancers should be utilized in order to distribute traffic across multiple servers to ensure high availability and reliability.
   3. Handling Dependencies:
      1. One strategy we recommend is implementing retry logic and fallback mechanism to handle events such as network outages. Another consideration may be to use distributed databases like Cassandra or MongoDB to ensure data consistency and availability is across different nodes.
6. **Security**:
   1. User Data Protection:
      1. Encryption: Encryption should be used to protect sensitive user data both at rest, as well as in-transit, using protocols like AES-256 and TLS.
      2. Authentication and Authorization: We recommend implementing OAuth 2.0 or OpenID Connect for secure user authentication. Furthermore, permissions should be managed via role-based access control (“RBAC”) with respect to the [Principle of Least Privilege](https://en.wikipedia.org/wiki/Principle_of_least_privilege).
   2. Platform Security:
      1. Firewalls: Firewall rules should be implemented to restrict access to essential services and protect against unauthorized access.
      2. Regular Updates: The operating system and software decencies should be regularly maintained and updated to protect against vulnerabilities.
      3. Authentication and Authorization: Similar to the above, we recommend permissions should be managed via role-based access control (“RBAC”) with respect to the Principle of Least Privilege, especially with public cloud solutions.
   3. Inter-Platform Security:
      1. Secure APIs: We recommend using API gateways to manage and secure API traffic, ensuring that only authorized requests are processed. Similarly, backend systems should not have public IP addresses; instead, utilize a Load Balancer, NAT, or API gateway to handle requests as opposed to backend nodes.
      2. Data Integrity: We recommend implementing checksums and digital signatures to ensure data integrity during transmission.