

Draw It or Lose It

# **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 | 09/21/2024 | Noelle Bishop | Filled in these sections: Executive Summary, Requirements, Design Constraints, Domain Model. |
| 1.1 | 10/6/2024 | Noelle Bishop | Filled in the Evaluation section. |
| 1.2 | 10/18/2024 | Noelle Bishop | Filled in the Recommendations section. |

## [Executive Summary](#_sbfa50wo7nsh)

The client is The Gaming Room, and they need a web-based version of their game “Draw It or Lose It”, currently only available as an Android application. The web-based game will be like its existing Android counterpart, but it will serve multiple platforms (i.e., mobile and desktop), fulfill specific code requirements detailed below, and give users a secure, session-based gaming experience.

## Requirements

1. Only one game instance can exist in memory at a time.
2. Each game instance must be able to support one or more teams at a time, each team with the ability to have multiple players assigned to it.
3. Unique names must be made for games, teams, and players. Users will be able to check whether a name is unique when choosing one for themselves or their team.
4. Teams/Players must be able to guess the answer to a puzzle in the form of a steadily rendered image. The puzzle images and answers are related to phrases, titles, or things.
5. The game must be able to render images from a library at a steady rate for 15 or 30 seconds depending on the type of round.
   1. If in a normal round: 30 seconds.
   2. If the team in the normal round fails, then every other team gets a guessing round lasting 15 seconds each.
6. Each of the four game rounds lasts for one minute.

## [Design Constraints](#_2et92p0)

1. The game must be web-based and accessible on desktop and mobile devices.
2. Multiple game instances cannot be simultaneously active.
3. Games, teams, and players cannot have repetitive names/identifiers.
4. The game is session-based, based on the criteria in Requirements.

*Implications of Design Constraints on Application Development*

The web-based design constraint impacts how the game is developed and on what website(s) it’s hosted. Perhaps the game could be made entirely in HTML by a team with coding experience in JavaScript, HTML, and CSS. Alternatively, a development team could make the game in Unity and C#, then upload it using Unity’s WebGL conversion capability. Either way, the result is a web-based game as the constraint dictates.

Because the game must be web-based *and* compatible with mobile and desktop devices, it must be compatible with different website browsers like Chrome, Edge, Firefox, and Safari. These are popular web browsers that many (if not most) users will be trying to play the game from. It must also be compatible with different operating systems like MacOS, iOS, Linux, and Windows in addition to its existing Android OS compatibility. These desktop and mobile OSs are the bases for the website that would host the game. This website would, ideally, be secured using HTTPS to protect user and game data from theft or cheating.

The ideal development team, then, must be experienced with web-based applications, different web browsers, screen scalability, keyboard versus touch controls, and cross-platform compatibility to fulfill the client’s request for a functioning web-based game. Similarly, the need for mobile and desktop access may affect the game’s user interface (UI). Perhaps a mobile version could be made with touch-screen-friendly buttons while the desktop version relies more on keyboard and mouse use.

Data transmission per game session is something to consider, as mobile users may have less storage available than desktop users do. HTML session storage may be enough, but if the game involves making an account and storing user records of wins and losses, then long-term local storage should be considered. The library of stock images may require cloud storage communication.

## [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 below features every class found in the com.gamingroom package. ProgramDriver has a usage dependency (indicated by the arrow labeled “<< uses >>”) on SingletonTester, meaning it uses SingletonTester as part of its implementation. GameServices’ getInstance() function uses a singleton pattern to check for one game instance existing at a time, fulfilling the related technical requirement. SingletonTester is a way for the programmer to check if the singleton pattern is working as intended, providing an efficient method of error detection.

The subclasses Game, Team, and Player inherit from the superclass Entity. GameService is associated with Game, which is associated with Team, which is associated with Player. Each of these associations also indicates cardinality, meaning GameService can have 0 or many Games, Game can have 0 or many teams, and Team can have 0 or many players. This relates to the game’s capability of assigning multiple players to a team and having multiple teams in a session. GameService has the capability to manage/store 0 or more games, but only one is allowed to be active at a time.

The inheritance described above reduces redundancy in the code, making development more streamlined overall. Instead of the same functions and variables appearing in Game, Team, and Player, Entity stores them for those three classes to access via its functions. Reduced redundancy makes code maintenance more efficient because a developer only needs to go to one place rather than three to update variables or functions.

The private variables in Entity are encapsulated, which ensures data about game sessions won’t accidentally be altered. Instead of accessing the variables directly, Entity’s subclasses use its get() functions to access the information in a safe way. Game sessions’ unique identifiers will not be compromised as a result.

Although not explicitly shown in the diagram, each toString() function in Game, Team, and Player uses the override command to create specific instances of toString(), a function also found in their superclass, Entity. In Java, this is Run Time Polymorphism. It fulfills the requirement of displaying unique game, team, and player names for the user (or developer) by altering the toString() function based on each class.

The functions addGame(), addTeam(), and addPlayer() use an iterator pattern with ArrayLists to check if a game, team, or player currently exists by name in the game instance. If it does, the existing instance is returned so as not to create duplicates.

**"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)

Please see the following pages for the Evaluation table.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | The Mac OS server capabilities have been discontinued since 2022, so Mac does not offer a built-in server-based deployment method. Running a server for a web-based app on Mac requires 1) buying a license for Mac OS and 2) a web server (e.g., Apache) with cloud communication capability. Mac is Unix-based, so it’s like and compatible with existing computer structures, but the lack of a built-in server-based deployment could cost extra time and money in its server set up. | Linux is known for being open source (i.e., free!), customizable, and possibly difficult to learn. It does offer built-in server-based deployment options like Ubuntu Server, which offers high scalability, frequent updates, and deployment/management tools. As discussed further below, using Linux can be less user-friendly if the user is looking for a more automated experience, thus proving to be a possible disadvantage. | Windows is a familiar OS for many. It is already integrated well with other Microsoft technologies like .NET that could provide a seamless, cheaper experience for the development team. Like Mac OS, developing a web-based app using Windows requires purchase of a business license to do so. Further, using Windows could limit developers to other Windows-licensed tools that work better than open-source ones in the integration of everything. Windows Server offers built-in server-based deployment through Internet Information Services (IIS), which works well with apps made using .NET frameworks. | Server-side requirements for mobile applications share less similarities than the three desktop-based OSs do. I couldn’t find built-in server-based deployment options specifically for mobile devices, especially not for business use, so it seems that implementing a web-based app here would involve the mobile devices interacting with remote servers similarly to Mac OS’s server implementation. Instead, the mobile devices are more like the clients that the server interacts with. The ease of access and advertisement from mobile OS app stores can broaden the game’s reach, but app store requirements can be costly and time-consuming to get through/adhere to. |
| **Client Side** | Mac hardware and licensing add to the costs of development. However, free, integrated tools like Xcode are more budget friendly. Developers would need to know how to use Mac-specific products, but once/if they do, development should be a relatively quick and streamlined process thanks to Mac’s built-in dev tools. Server development would need to be done using Apache or similar. Experience with Swift might be necessary. | Linux app development could require special training or outsourcing if the OS is unfamiliar to the available development team. Implementing server/client structures on Linux could prove to be a more manual rather than automatic experience, which places more control in the developer’s hands but requires more expertise on *what* to do. Thankfully, Linux can work on a wide range of hardware and is compatible with lots of free/open-source software, reducing costs overall. Developing in Linux requires knowledge and experience in using a command line interface. Since this is similar to working with some versions of SQL, skill overlap could save resources if SQL is used for database management. | The limitations of using Windows are traded off with lots of documentation and readily available support from Microsoft staff and other developers. The built-in IIS could simplify the development process, but if it’s not used, Apache is a great choice that’s compatible with Mac, Linux, and Windows, making development more streamlined for all. For all implementations across OSs, web browser compatibility must be tested, and cross-platform compatible tools like React and React Native (for desktop and mobile app development, respectively) are likely to garner the most success. React-based apps can be hosted on Apache servers, further boosting compatibility. | Mobile development comes with its own requirements. While the app for Mac, Linux, and Windows could use Apache, Python, and mySQL in their development processes, for example, mobile development would use its own software development kits (SDKs) and different programming languages. The differences grow when considering which mobile OS to develop for. Further, touch-based controls need to be considered for mobile users. |
| **Development Tools** | Mac, Linux, and Windows can all use the Visual Studio IDE for development, which requires purchase for business use. JetBrains WebStorm is another IDE compatible with the three desktop OSs, and it requires a paid subscription for professional use. Benefits of using these IDEs are ease of use, cross-OS compatibility, and web-based development features. They offer large, documented products that let developers work with JavaScript, HTML, CSS, and other relevant languages for building the web-based app. | Visual Studio and JetBrains WebStorm are potential IDEs (paid, discussed more in Mac section). The cross-compatibility of these products lends itself to using only one development team for desktop development. Additionally, Git is a system that works with every one of these desktop OSs to keep track of versions and updates. Having a free-to-use tool like this makes maintenance across OSs more streamlined, again suggesting the need for only one development team. | Visual Studio and JetBrains WebStorm are potential IDEs (paid, discussed more in Mac section). Visual Studio Business edition costs $45/user per month while WebStorm costs $15.90/user per year, to give an example of estimated costs of using either IDE. As it’s been discussed, one development team is likely suitable for desktop development across the OSs. Still, front-end and back-end specializations would be beneficial within the single team so that the user experience (front-end) is developed separately from the web app’s core functionality (back-end). | Apps for both Mac OS and iOS can be developed using Xcode, a free IDE designed for Apple products. This overlap can reduce costs and labor requirements. Still, multiple development teams may be needed for desktop and mobile development, since the two share so many differences. React Native could be the ideal method for developing the app across mobile OSs. Using this would require knowledge of JavaScript, but it could greatly streamline the process thanks to its robust library. Further, it’s a free/open-source tool, so it would lower the cost of mobile development. |

## Recommendations

1. **Operating Platform**:
   1. Linux is the recommended operating system for this project, to be used in tandem with a cloud-based distribution platform like Microsoft Azure or Amazon Web Services (AWS).
   2. The strength of Linux lies in its open-source nature. Compared to Mac and Windows business costs, Linux is the much cheaper alternative. Although working with Linux has a learning curve, the pay-off is its compatibility with a variety of web development tools, meaning the development team could customize their work layout easily. The use of a cloud platform like Azure also reduces the overhead for Linux development because it would eliminate the need for server deployment.
   3. Azure or AWS would both be viable options given their large company-backing, pay-as-you-go bases, and storage capability (discussed further in storage management). Additionally, these platforms fulfill the need for cross-compatibility across desktop and mobile operating systems/devices. As the client grows their user base, the cloud infrastructure grows with them, meaning affordability is scaled to the game’s popularity.
2. **Operating Systems Architectures**:
   1. Using Linux instances and cloud services, containerized architecture is recommended. Containerization refers to bundling all the necessary files for an application’s deployment, making it easily transferrable between platforms.
   2. Docker is a tool that would effectively package everything needed for the game to be easily deployed via cloud distribution. Further, it would modularize development should the team want to specialize in different parts of the project (e.g., the game itself vs. only the images vs. the database).
   3. Kubernetes, another Linux-compatible tool, should then be used to manage the Docker containers. Kubernetes essentially lets developers determine how and when Docker containers run based on resource management. Grouping and scaling containers are also possible with this tool. If a container fails, it can be rebooted from this overhead vantage point. As an additional bonus, Azure has an available Azure Kubernetes Service (AKS) that would reduce the learning curve of using Kubernetes from scratch.
   4. Docker requires a paid subscription for business-related uses, but Kubernetes is open-source and free to use, making the two an affordable combo, especially given the free usage of Linux. If Azure is chosen over AWS, then AKS could be used as part of the Azure subscription, instead of implementing Kubernetes as something separate.
3. **Storage Management**:
   1. Given the size and number of high-definition images needed for the game’s design, cloud storage via Azure or AWS is recommended.
   2. The image library can be stored in its own container via the cloud, and the built-in content delivery networks (CDNs) of either service would allow for quick, effective image delivery at the correct gameplay moments. Given the necessity for the images to be rendered at a steady rate per the in-game timer, the quickness of CDNs (thanks to their caching) would fulfill this requirement.
   3. Other data like active game sessions, team/player names and states, and other temporary in-game data can be stored via cloud caches. Azure Cache for Redis, as part of an Azure subscription, would allow for “thousands of simultaneous users”, according to the product page, making it suitable for a growing user base.
   4. More permanent data, like persistent user accounts and their unique names, could be achieved through cloud database services. Both Azure and AWS offer SQL options to create and keep relational databases of this long-term information in cloud storage. A benefit of using cloud storage is how easily backed-up this kind of important information is.
4. **Memory Management**:
   1. Using containers and container management allows the game instances to scale based on resource needs. In other words, AKS or Kubernetes can manage how much memory each container uses based on need, further making Docker plus Kubernetes the recommended combination. Kubernetes offers container lifecycle management (i.e., memory clean-up) to easily keep the game’s containers in a like-new state.
   2. Additionally, cloud monitoring tools exist to give the development team a peek into how the cloud-based infrastructure is holding up in terms of performance and memory. Prometheus is an open-source (free!) cloud monitoring tool that works well with a Kubernetes set-up, so it’s an effective and affordable choice for this project to monitor the game’s performance and troubleshoot potential memory-hogging processes.
   3. The singleton pattern used in the game’s code (held within a cloud container) ensures that only one game instance exists in memory at a time, per the crucial requirement. Azure Cache for Redis (previously discussed) can effectively produce this singular game instance per user, per request.
   4. Using a per-request/pay-as-you-go cloud framework means that memory is dynamically allocated as it’s needed. The memory needed for image deployment through CDNs is cached for quick access. These cloud-related features should reduce the needed RAM for each user’s experience.
5. **Distributed Systems and Networks**:
   1. Cloud services like Azure offer load balancing services to properly distribute and optimize various parts of the project across Linux instances, in this case. In tandem with a cloud monitoring tool like Prometheus, the ways that the game uses resources and responds to real-time requests (such as the game’s code requesting an image to be displayed) can be effectively managed. Connectivity issues and outages can be discovered and re-balanced using these features/tools. As the user base grows, load balancing can distribute web traffic to handle high-traffic gaming spikes gracefully.
   2. The CDNs available on Azure and AWS produce the images needed for gameplay regardless of a device’s storage restrictions, thanks to its cache-storage mechanism.
   3. Using RESTful APIs would allow for seamless interactions between the game’s website on different browsers and the cloud servers the game’s data is hosted on. These APIs would exist as Docker containers managed by Kubernetes, managed by a cloud service.
   4. Since the game and the users will be interacting in real-time during play sessions, WebSocket Secure (WSS) connections should be used for continuous in-game requests. WSS is compatible with HTTPS (discussed in next section) and maintains the connection between the client (website) and server (cloud) until the game is over.
6. **Security**:
   1. Given the requirement for unique names for players and teams, role-based access control (RBAC) is one recommended security measure to take. Users can create accounts with unique usernames, and the development team can allow for and encourage strong passwords. Users would then be given “player” level access when performing different in-game actions, effectively allowing them to only interact with the game in intended ways. Alternatively, “admin” level users could interact with the game via specific codes and a command line for debugging purposes. This sensitive username and password data would then be encrypted and stored in a secure cloud database, which admin users could access.
      1. “Encryption at rest” is a security aspect of cloud services like Azure. As sensitive data is written to storage, it is encrypted with a key that then decrypts it when it’s needed for use. This process keeps long-term data resistant to attacks.
   2. Using HTTPS to ensure secure connections between the web browsers and the cloud servers is another necessary security measure to take. HTTPS encrypts game data and user communications in real time using the SSL/TLS protocol. This protects the back-and-forth data exchange that goes on between users, the game, and the cloud from malicious eyes. HTTPS also authenticates the game’s website, kind of like verifying its user status in an RBAC operation, to ensure the site is not a malicious fake.
   3. Automatic Linux security updates can be set up across instances so that the game instances are always up to date. Security updates often patch vulnerabilities, so staying updated is a crucial first line of defense.
   4. Both Azure and AWS offer built-in security features, such as protection from distributed denial of service (DDoS) attacks, RBAC-like features related to identity and privilege management, and logging (for retrospective attack or vulnerability identification). Since these features are part of either subscription, they make cloud services the recommended choice of both platform and security.