

Draw It or Lose It

# **CS 230 Project Software Design**

Version 1.2

## Table of Contents

[**CS 230 Project Software Design Template**](#_l6ti7uoag22u)1

[**Table of Contents**](#_30j0zll)2

[**Document Revision History**](#_grjogdjh5fi8)2

[**Executive Summary**](#_sbfa50wo7nsh)3

[**Design Constraints**](#_2et92p0)3

[**System Architecture View**](#_ilbxbyevv6b6)3

[**Domain Model**](#_8h2ehzxfam4o)3

[**Evaluation**](#_2o15spng8stw)3

[**Recommendations**](#_m8aleynsvzvc)5

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 01/22/22 | Mark Holden | Initial Design |
| 1.1 | 02/06/22 | Mark Holden | Evaluation Section |
| 1.2 | 02/17/22 | Mark Holden | Recommendations Section |

## [Executive Summary](#_sbfa50wo7nsh)

Draw It or Lose It is a multi-player, team-based game that allows users to guess what a drawing is as it is rendered at a steady rate. Each round, a single team has the opportunity to guess what is being rendered while the other teams watch. The guessing phase of the game is timed to be 30 seconds and if the selected team guesses correctly, the round ends and only that team receives points. If the team fails to correctly guess the drawing, the other teams are all allowed 15 seconds to offer a single guess. All other teams perform this guess at the same time without seeing the other teams’ responses, and they are all revealed at the same time. All teams with correct answers are awarded points.

To set up this game experience, users will create a player when they begin using the application. Once a user has created their player, they may either create a team or join an existing team. Once a user is ready to start the game, they will indicate they are ready with a “ready” button. There will be a time limit set for this process based on user feedback and at the expiration of the time, all users that are not already on a team will be automatically assigned a team and the game will start. If all users indicate that they are ready to start the game before that time elapses, the game will begin immediately.

Once the game begins, the teams take turns as the first team to guess based on a randomized order of the teams. During gameplay, any user can make a guess, and if any player guesses correctly at any time during the 30 second guessing phase, the team gets the points. If no player on that team guesses correctly during the time limit, the other teams will have 15 seconds to decide what their guess will be. This guessing procedure will be each player having the ability to enter a single guess that is displayed to all other users on the team, and the ability to vote on suggestions from other teammates. At the end of the 15 seconds, the team guess will be made by the following selection process. If there is consensus on a single answer, then that answer is given. If there are two or more options, the one with the most votes from other players on that team is given. If there are two or more options with the same number of votes, the team’s guess will be randomly selected from the options that have the highest number of votes.

After all teams have had a chance to be the first team to guess, the game is over, and the winner is determined by how many points each team earned. Additional features such as a leaderboard and other tracking for previous winners are out of scope for this document.

## [Design Constraints](#_2et92p0)

To start a game, the game must have at least one team with at least one player.

To create a new game, the game’s name must be unique within the system. To prevent conflicts, the system will queue requests to add games and process them in the order received.

To create a new team, the team’s name must be unique within the system. To prevent conflicts, the system will queue requests to add teams and process them in the order received.

A maximum of one game can be loaded at a single time.

When adding a team to a game, the team must be checked to ensure that a minimum of one player is on the team.

Past wins are not recorded in the scope of this document.

## [System Architecture View](#_ilbxbyevv6b6)

The System will implement [onion/clean/hexagonal architecture](https://medium.com/@edamtoft/onion-vs-clean-vs-hexagonal-architecture-9ad94a27da91) to replace dependencies on data storage and other systems with interfaces that can be replaced at any time with another implementation of the interface. The core of the system will be the Draw It or Lose It domain models, and the services will provide use cases that a UI can use to display all necessary information. That will prevent business logic from existing in the UI and allowing it to change or be ported into other systems without having to replicate any business/game logic.

Development will take place using Test Driven Development following the three rules of test-driven development given by Robert C. Martin (2005):

You are not allowed to write any production code unless it is to make a failing unit test pass.

You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.

You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Developers will utilize JUnit for unit testing in Eclipse.

## [Domain Model](#_8h2ehzxfam4o)

**Diagram

Description automatically generated**The below domain model for Draw It or Lose It includes the Game Service in the lower right-hand corner that will maintain a singular instance of itself (the private constructor ensures that no other part of the system can create another one). That service will have methods that allow Games to be added or retrieved and fulfill the requirement that there be only one game in memory at a time.

The next domain classes are the Game, Team, and the Player. These classes all inherit from the base class, Entity, that contains common methods and fields. Inheritance helps to fulfill the software requirements efficiently because the fields in the parent class must only be written once and can be used by any child classes.

The Game class has a public constructor used by the Game Service to create new Game objects. [As a side note, I would prefer to have the data type for Id be a GUID/UUID rather than long to ensure they are universally unique, and not have to pass them to the Game constructor – we can just let the Entity class take care of that. I made that change in the code and UML diagram.] The Game class also has a method to add a Team which creates a new instance of the Team class and add it to the private List of teams contained within the Game class. The Game class also has a ToString method that overrides the base functionality to return a human readable string with the format "Game [id={id}, name={name}]".

The Team class has a public constructor used by the Game class to create new Team objects. The Team class also has a method to add a player which creates a new instance of a player and adds it to the private list of teams contained within the Team class. The Team class also has a toString method that overrides the base functionality to return a human readable string with the format “Team [id={id}, name={name}]".

The Player class has a public constructor used by the Team class to create new player objects. The Player class also has a toString method that overrides the base functionality to return a human readable string with the format “Player [id={id}, name={name}]".

## [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** | One of the biggest advantages of using a Macintosh for the server-side application is that it is unlikely that any significant malware will be written to attack the operating system (OS). One disadvantage is that even though it appears that you can run NGINX to host the application, you would still have to buy an overpriced Macintosh just to put it in an (air conditioned) closet. You can actually get a virtual machine (VM) on Amazon Web Services (AWS) that runs Macintosh, but that appears to be the exception, rather than the rule; Microsoft Azure does not have Macintosh VMs or app services. The biggest disadvantage in my opinion is the lack of documentation, community resources, and support from third-party services, that would help get this set up and keep it running smoothly. Requiring somewhat specialized knowledge to keep a server running is a poor long-term strategy. | Linux seems to be one of the go-to OS for hosting web applications because it does not require a license, and it has less OS overhead than Windows. It has the advantage over Macintosh that there is a huge community of developers to answer questions about it, and a great deal of documentation. One possible disadvantage is that it is extremely common for developers to use Windows machines to write the software, so unless you are using Docker, the code will be executing on a different operating system than it was written on, so if the framework has differences in how the application programming interface (API) works with the OS, the behavior could be different. For example, .Net can be run on Linux directly, but there is an API that translates the C# instructions into instructions that Linux can run. The translation of that into what Linux does could be slightly different, but it is probably not a major concern. | | Windows seems like the other go-to OS for hosting server-side applications. It is easy and familiar and has the advantage that you can run the exact same operating system on your development machine as the actual server that hosts the application. You can technically do this with Linux too, and I’m sure some software engineers do, but from the few software engineering jobs that I have had, Windows appears to be more common. For hosting on a physical machine or even a virtual machine, Windows offers the power of Internet Information Services (IIS) to host the application, which based on my experience seems to be a common thing for developers to be familiar with and able to troubleshoot readily. | While you technically *can* host the server-side application on either Android or iOS, the hardware available is such that it is not suitable for running a web server for any significant user traffic. Some characteristics to the Android device’s credit is that they are extremely affordable, and to the credit of both Android and iOS devices is that they are small, so they can fit just about anywhere. This advantage is diminished somewhat by the fact that using AWS or Azure means that the footprint of the physical server is irrelevant because it is in a datacenter somewhere that and there is no reason to care. There are also some other devices out there that could fall into this category which suffer from the same pitfall in terms of processing power. According to Cawley (2018), even a Raspberry Pi can be used to run a web server. |
| **Client Side** | This is .Net’s time to shine. One of the beautiful things about .Net is that you can run the same code on Macintosh as you can on Linux and Windows. That means that you can develop the software once, focusing all your effort into getting it right rather than developing the same thing multiple times for different platforms. Another option could be writing the Macintosh specific application in Swift or Objective-C. There are a great many Swift and Objective-C developers that can take on this task, and it also has the advantage of being able to share code with AppleTv and iOS. Writing for the AppleTv wasn’t a requirement, but it could be seen as an advantage for developing natively for Macintosh in Swift. | | As mentioned previously, it is recommended to use a cross platform framework such as .Net to write the application once and have it just *work* on Linux. That would require no further development cost, time, or expertise beyond potentially having specific instructions for running the application on Linux. Even though you can develop an application for Linux in just about any language you desire, they are all going to have different frameworks and packages for interacting with the operating system. If it is decided that the application will be rewritten in C so it will run faster on Linux, that doubles the development time, possibly more if there aren’t any C developers available. | Once again, .Net is going to get the crown, and on Windows that seems obvious. Assuming that the application had not already been developed in .Net from either the Macintosh version or the Linux version and Windows was not the first platform to write the game for, you could port whatever application was written for Linux over to Windows, which would mean that you get to reuse many of the core libraries and business logic, but anything that touches a package that isn’t available on Windows or uses an OS specific command has to be replaced. If Windows was the first OS that the game was being developed for it would be a complete no-brainer to develop it in .Net and have it immediately deployable on both Linux and Macintosh. | Mobile devices are going to be a little tricky. First, if the application was developed in Swift for Macintosh, that gives native iOS development in Swiift a head start on other languages. I am not a Swift developer, so it is unclear to me how much of the Macintosh code could be reused for iOS, but there is probably some. Then, the application would have to be developed again in Kotlin or Java. On the other hand if the Macintosh application was just the same .Net application as on Linux and Windows, then it might be beneficial to use the Xamarin framework to stay in C# and develop for both Android and iOS at the same time. There is some code in Xamarin apps that is not shared between the two platforms, but all of the core business logic is shared. |
| **Development Tools** | In the event that the application is being developed natively in Swift however, the developers must have a Macintosh computer and XCode, since XCode only runs on a Macintosh. With regards to the server-side application, if the server is being run on a Macintosh, even if it was a .Net application, software to host the application, such as NGINX, would still be required to serve the application. From what I can tell, there is no licensing costs for the XCode software, but you have to have an overpriced laptop to install it, and to submit apps to the App Store, you have to join the Apple Developer Program, which costs $99. Another option that available on a Macintosh development machine is Visual Studio Code and .Net! One of the companies that I worked for previously had only Macintosh development machines, so it was either VSCode or JetBrains Rider. | According to the [JetBrains website](https://www.jetbrains.com/rider/), Rider is an IDE that will run on Linux. From what I know of Linux, developers running that OS are the types of folks who are very comfortable in the command line, and sometimes use vi as their primary text editor. In other words, wizards. Based on the advertisements from JetBrains, it would seem straightforward to use Rider on Linux to develop a cross-platform .Net application that would run on Macintosh, Linux, and Windows. If a different language were chosen, JetBrains has an IDE for that too. They offer a cross-platform IDE for C and C++ called CLion. | | To develop .Net applications on Windows, the premier IDE is Visual Studio. [Visual Studio Professional subscriptions](https://visualstudio.microsoft.com/vs/pricing/?tab=business) are $45. Another option is [JetBrains Rider](https://www.jetbrains.com/rider/buy/#commercial) which is about $400 per year, but it has ReSharper built in, and that is an additional thing that one would definitely want to add on to Visual Studio if you go that route. I have heard good things about Rider, and there are some people who prefer it over Visual Studio. One more option to develop for .Net on Windows is Visual Studio Code. VSCode is free, open source, and has a ton of extensions that can be installed to assist in development, most of which are free. | To develop cross platform mobile applications in C# with Xamarin, one can use Visual Studio, VSCode, or JetBrains Rider with an additional plugin. To go the native route, Android studio is probably the way to go for Kotlin/Java, but the only way to develop Swift in in XCode on a Macintosh. That is another reason to advocate for Xamarin. Not only do you develop the application once for two platforms, but the IDE isn’t being held hostage by one hardware provider.  Another thing to consider with mobile development is that one will probably want some testing devices. Most developers are happy to use their own device to test applications, but usually they will only have one phone, so there will be an entire platform that is only testing in a simulator unless devices are provided. |

## Recommendations

In addition to the above considerations given, the below recommendations also account for the familiarity to the platforms and frameworks that our current development team has, which will benefit Draw It or Lose It by reducing the time it would take to familiarize themselves with a new framework and platform.

1. **Operating Platform**:
   1. The primary recommendation for the operating platform to use is Microsoft Azure Platform as a Service (PaaS). Using Azure allows the development and direct deployment of .NET 6+ applications directly to Azure App (Application) Services from Azure Pipelines. Azure Pipelines can be configured as a Continuous Integration/Continuous Delivery (CI/CD) pipeline. CI/CD means that as developers work, their work is immediately built and deployed to the appropriate test environments. There are many frameworks that are compatible with Azure but based on the high proficiency of the team in .NET, .NET is recommended as the development framework to use with Azure PaaS. As mentioned previously, this framework allows for cross platform development in C#, which allows our team of .NET developers to rapidly deploy to any platform as the needs of the business evolve over time.
   2. The alternate recommendation is to use Digital Ocean could services to host a Kubernetes cluster and deploy containerized .NET applications to the cluster using Docker. An advantage to this recommendation is that the Kubernetes cluster can be hosted anywhere, even on physical on-premises hardware without requiring any code changes. There are some minor drawbacks such as the configuration of the cluster is much more complex than simply using Azure Web Apps, and there are minor performance implications of running multiple layers of platforms.
2. **Operating Systems Architectures**:
   1. The primaryrecommendation for the server operating system is Microsoft Windows. This choice has less of an impact than the operating platform because with a choice of using the .NET framework to develop the back-end application required for Draw It or Lose It, the application can run on Windows, Linux, or Macintosh. The feature set of Windows is robust and supports the .NET runtime very well. It is also one of the operating systems that is compatible with Microsoft Azure. For the client operating system, since customers will be bringing their own device to use, the most common operating systems will be Android and iOS. To rapidly support both platforms with a version of the game, it is recommended to use the new .NET Multi-platform App User Interface (MAUI). This framework is the replacement for Xamarin, and in addition to supporting Android and iOS, it also supports Windows and Macintosh apps (Ramel, 2021).
   2. The alternate recommendation for operating system is Linux. Linux is also supported by both the .NET runtime and by Azure, so in theory, the systems can function in a similar manner.
3. **Storage Management**:
   1. Theprimary recommendation for storage management is to start with Elasticsearch for data persistence for large records and utilize Redis for caching as much commonly requested data as close to the end user as possible to increase system performance.
   2. The alternate recommendation is to stand up a full-blown SQL instance, preferably in Azure in the same region as the app services. This may be necessary in the future anyway based on how the system changes over time, but it is not the primary recommendation as a first step because it may not be necessary and would add complexity. If the SQL database option is used, there will still be some data that is stored in Elasticsearch and Redis for performance reasons.
4. **Memory Management**:
   1. The primary recommendation for memory management is to utilize the stack – that is, short lived function scoped variables with basic data types and limited capacity – for as much processing as possible. The heap – longer term memory with much larger capacity – will be used ad hoc when creating objects by using the ‘new’ keyword, and .NET garbage collection will be relied on to clean up resources as needed. The reason for this recommendation is that in the initial stages of development, it is not known what components of the system the performance bottlenecks will be, so the added time and complexity to optimize things will be wasted. As the system matures and the performance issues are identified, they will be optimized as needed for performance. In the initial phases of development, direct allocation of memory will be avoided.
   2. The alternate recommendation is to determine the memory that will need to be allocated during gameplay and allocate all the memory that the program will need when it is started. This has the potential benefit that all the memory for the application will be contiguous and allow for faster lookups. The drawback to this is that the memory management code will have to be written before the application code, and many code changes in the application will require a code change in the memory management class.
5. **Distributed Systems and Networks**:
   1. The first recommendation is to utilize a Content Distribution Network (CDN) for the game images. As the most data heavy part of the application, latency based on geographic distance from a data center where the images are stored can have an impact on how long a user waits to play a game. By using a CDN, the user’s device can request the image, and it will be served to them from whichever data center will give them the best download performance. Utilizing a CDN service such as Azure CDN means that all the replication and management is taken care of by the third party, and the images will be seamlessly available.
   2. Another recommendation (not an alternate, this is recommended in addition) is that all back-end systems required for Draw It or Lose It be hosted in the same data center, or at least in the same region. This will reduce the round trip between internal system communication that happens before a response is even sent to the user’s device.
   3. A third recommendation (also recommended in addition to the above) regarding networks is that there should be a load balancing system that can direct app traffic to instances of the service that are under the least amount of load at that moment and re-route traffic when a service is having an issue. There should be a minimum of two of each of the application critical services running in different regions so that if one region is degraded, the end users will be using the unaffected region without even knowing there was an issue.
6. **Security**:
   1. The primary recommendation for user authentication and authorization is to utilize the built-in Azure App Service auth. The best reason to use this auth is that Microsoft handles keeping up with industry best practices for security, and there is a minimal amount of code required to have a secure application (Cephalin et. al., 2021). In addition to using Microsoft’s authentication and authorization, data will be sent over secure https connections. Since authorization is being handled by Azure in this case, there is another benefit that no passwords will be stored whatsoever.
   2. An alternate recommendation is to use another third-party authentication and authorization server such as Fusion Auth or Auth0. When using these third-party systems, if the data is managed by Draw It or Lose It, passwords must be stored salted and hashed, and only the salted hashed password compared to a user entered password (Arias, 2019). Any user Personally Identifiable Information (PII) is to be stored on servers with privileged access only. For example, when a production database contains PII, developers must be restricted to only non-production databases that have had the PII stripped from the records.

**References**

Arias, D. (2019, September 30). *How to hash passwords: One-way road to enhanced security*. Auth0. Retrieved February 18, 2022, from https://auth0.com/blog/hashing-passwords-one-way-road-to-security/

Cawley, C. (2018, June 13). *7 great project ideas for using a raspberry pi as a server*. MUO. https://www.makeuseof.com/tag/raspberry-pi-server-project-ideas/

Cephalin, Chenderson, M., rwike77, cgillum, v-kents, knicholasa, mkarmark, johndowns, BryanTrach-MSFT, DCtheGeek, jelledruyts, mmacy, Howell, J. W., & davidmu1. (2021, November 23). *Authentication and authorization - Azure app service.* Authentication and authorization - Azure App Service | Microsoft Docs. https://docs.microsoft.com/en-us/azure/app-service/overview-authentication-authorization

Damtoft, E. (2021, November 30). *Onion vs Clean vs Hexagonal Architecture*. Medium. https://medium.com/@edamtoft/onion-vs-clean-vs-hexagonal-architecture-9ad94a27da91

Martin, R. C. (2005). *The Three Laws of TDD*. ArticleS.UncleBob.TheThreeRulesOfTdd. <http://www.butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd>

Ramel, D. (2021, July 27). *Microsoft replaces Xamarin toolkits with new .NET MAUI alternatives*. Visual Studio Magazine. https://visualstudiomagazine.com/articles/2021/07/27/net-maui-kits.aspx