

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 3.0

## 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 4**](#_Toc115077324)

[**Evaluation 5**](#_Toc115077325)

[**Recommendations 7**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/15/23 | Mike Sabeiha | Initial assessment and constraint analysis |
| 2.0 | 10/01/23 | Mike Sabeiha | Evaluation and Recommendations |
| 3.0 | 10/15/23 | Mike Sabeiha | Final Recommendations |

**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 Client, The Gaming Room, would like to take their current Android based game—Draw It or Lose It—and convert it to a multi-platform, web-based game that can be accessed from multiple OS and run on various hardware systems. The game challenges one or more teams to guess a concept by interpreting hints provided by images shared over a 30-second time span. If one team fails, another team then gets 15-seconds to guess correctly. The game requires an online environment to be established and the Client has retained our services to complete this portion of the project.

## Requirements

The Client will require an online environment where the game can be run, a server to host, a universally interpreted language, as well as account security, graphical user interface, and digital animation to complete the game. The game will need to run with only one instance at a time but must be available for several teams to compete during each instance and with several players on each team. It must also be able to uniquely identify each instance as well as each team playing during each instance by not only tracking identifying information, but also ensuring no identities are used by multiple entities simultaneously.

## [Design Constraints](#_2et92p0)

The online environment will need to be built adaptable to many OS and hardware systems and be coded in a universally interpreted language. I recommend using .NET framework and an object-oriented language such as Java. The game can be hosted on AWS for scaled pricing that keeps the client from paying for server usage and space they don’t need. This also enhances the security of user accounts as we can build off the AWS security systems already in place.

The GUI will need to be built in an efficient and portable manner and digital animation will need to be scaled easily. If the game is to be web-based and run on numerous platforms, we can code one uniform back end, but the front end will need to be adaptive and will need to adjust for screen size, controllers, etc. This adds to the number of labor hours needed to get the job done but is crucial to ensuring that all users have an enjoyable experience.

As for the software body itself, it will need to instantiate and protect one instance of the game loop at a time while also managing a database of teams, players, and images used for hinting to players. The database containing drawings will likely be static and accessed only by the design team, however, the database containing team and player information will be dynamic, expanding and being accessed as needed. A singleton pattern can cover the game instance but a factory method with a will likely be the solution for creating unique identifiers. There must also be an iterator to run the checks on new identifiers that ensures there are no duplicates. A façade will ensure that players are able to interact with the game in a controlled way that does not allow them to inadvertently engage other parts of the whole game.

## [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 game is stored in the com.gamingroom package. Within that, the ProgramDriver class holds the main driver class that runs the entire program. Within the ProgramDriver class, the SingletonTester class is accessed to ensure that only one instance of the game is running at any time.

From ProgramDriver, the class GameService, handles the functionality of the game. GameService can store games by their unique ID, instantiate and protect the single running instance of the game, as well as control access to the instance. Further, it allows the system to add new games, new teams, and new players as well as access them later. This demonstrates the principle of encapsulation which ensures that certain methods can only be accessed by certain objects which helps to satisfy the clients request for there to only be one instance of the game at a time.

The Entity class is an abstract class that is used as the basis for the Game class, Team class, and Player class. This demonstrates not only the principle of Abstraction but also the principle of Inheritance and satisfies the clients request for a game that can create multiple unique objects that track individual games, teams, and players. As each of these classes is used to create new objects, their parent-child relationship to Entity is utilized to ensure efficiency and uniformity in the production of new objects. These classes not only manage databases, they also add new objects to those databases, ensure their identifiers are unique, and allow the system to access them later.

The Entity class also presents a clear case of polymorphism. The abstract class allows the creation and management of a unique ID. However, the sub-classes also have their own unique methods for assigning a unique ID. This design, which runs a method inherited from a parent class, but uses a different name, is a good example of polymorphism and supports the clients decision to have a system that effectively tracks unique entity names through the game.

Within each subclass, its also worth noting that each instance of the game requires at least one game object, and each game object requires at least one team object, and that each team object requires at least one player object. This represents the principles of Association, Aggregation, and Composition. The design demonstrates clear dependency between each of the Entity subclasses as well as the Game and GameService classes in that one needs the other. This works toward meeting the clients request for a game that allows only one instance of the game, with at least one team and one player on that team while also allowing the capacity for many players on many teams to compete within a unique game.

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Using MacOS as a platform for service side systems would be helpful if we were focusing on Mac/iOS users, however, it’s lower compatibility with other OS would create unnecessary hurdles in the development of a multi-platform app | Hosting the game on a Linux platform would allow freedom in design and greater efficiency, but would create a talent gap if our client doesn’t have the expertise to run a Linux based server | Most web-service platforms offer windows as an OS of choice so hosting the game on a Windows-based service will be both time and cost efficient especially if a .NET framework is used to ensure versatility | Hosting a game on mobile devices is unreliable, though theoretically possible, perhaps a crypto-fantasy that would improve the IPO price of the company but unfeasible from our standpoint |
| **Client Side** | It would be absolutely necessary to ensure the game functions well on the clients side for Mac/iOS users, but the added labor-hours can be mediated using .NET framework so as to avoid making a Mac/iOS specific version of the game | Very few users will engage the app from Linux systems but if they do, it is web-based so our only concern would be to ensure the game renders properly on the user end | Most of our desktop clients will access the game via windows so ensuring that our web-based game renders properly on windows devices is essential | Most of our clients will likely access the game via mobile devices so ensure the game is compatible with iOS and Android is essential, we must also ensure the game renders properly to improve the user’s experience |
| **Development Tools** | Mac hosts a number of creative development tools that would provide an artistic advantage over other systems, but with most technology available today, the advantage is marginal and our priority in developing this game is cost efficiency rather than artistic excellence | Linux offers much freedom and advanced capabilities for software development, however, it’s an esoteric platform and using it would create unnecessary knowledge gaps that would be difficult to overcome | With .NET and Java, developing the game on windows will ensure our staff are not hobbled by gaps in expertise and allow us to create one version of the game that will work on most all OS and Hardware platforms | Developing the game on a .NET framework saves us the effort of creating multiple versions of the game, but if we needed, JavaScript works well for universally adaptive game development |

## 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**: The best operating platform is WindowsOS which would allow us to quickly produce and maintain the game without needing to overcome any knowledge games or technical incompatibilities. It would also ensure we can build the game on the .NET framework which will help ensure the game is universally accessible.
2. **Operating Systems Architectures**: The game will be hosted on a third-party server so development will be streamlined and considerations such as hardware, kernel, etc. will be boilerplate. This improves reliability and makes our backend nimbler. If we need to shift to a more robust system, we can do that quickly without having to rebuild our own server.
3. **Storage Management**: The game will be stored on the cloud which will be backed up and protected within the cloud environment, though it would be helpful to retain copies of the game on local memory for the sake of redundancy. Storing on the cloud also allows us to expand as needed without needing to buy and install physical hardware. We would simply make an election with our cloud service to provide us with greater server space.
4. **Memory Management**: Choosing to host the game on the cloud means that cost will correlate directly to the amount of data flow on our system. To ensure we aren’t over-using the server for unintended tasks, we’ll need to build a system to observe and manage data flows assuming the web-service doesn’t have one pre-built.
5. **Distributed Systems and Networks**: The game will rely on a robust API between many different platforms to ensure the right data is being received, processed, and distributed. A traffic management system will need to be part of the game’s code and ensure that the right IP is activating the right player ID and so forth. Hosting this on the cloud ensures reliability as we can likely trust their systems to handle large traffic flows without failure.
6. **Security**: We can rely on the built-in security systems of the third-party web service, but it would be helpful to add our own security features to ensure users aren’t unwittingly sharing private information with us as well as monitor communications between players. As long as we prevent any high-risk information entering our system, we can ensure it doesn’t leave our system and thereby avoid data breaches.