Draw It or Lose It
CS 230 Project Software Design
Version 1.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	05/22/22	Ethan Klukkert	Completion of Executive Summary, Design Constraints, and Domain Model.
1.0	06/05/22	Ethan Klukkert	Completion of Evaluation
1.0	06/19/22	Ethan Klukkert	Completion of Recommendations

Executive Summary

The Draw It or Lose It application for The Gaming Room needs to provide an environment for unique games, teams, and players using one central game service component. The singleton design and class inheritance patterns will ensure this environment is secure and functional. Processing the games at the same time will allow users to interact with the system seamlessly.

Design Constraints

The design constraints for developing the game in a web-based distributed environment is the fact that the processes of the games must be done at the same time. So, the singleton game service has to manage many games synchronously, and the player clients must be able to interact with their game all at the same time to deliver an experience to users without their actions interfering with others' actions. Therefore the software application has to accommodate this design by allowing multiple threads and/or processes to execute, which will then require careful development.

Domain Model

The ProgramDriver contains the main function and runs the program. The driver can also use the SingletonTester to ensure that only one instance of the GameService exists at a time. The Entity class works as a template for the Game, Team, and Player classes. The three other classes inherit from the Entity class, ensuring that each class has an id, a name, and getter

methods for these properties. Using inheritance, the Game, Team, and Player classes have this basis to then add their specific attributes and overwriting methods.

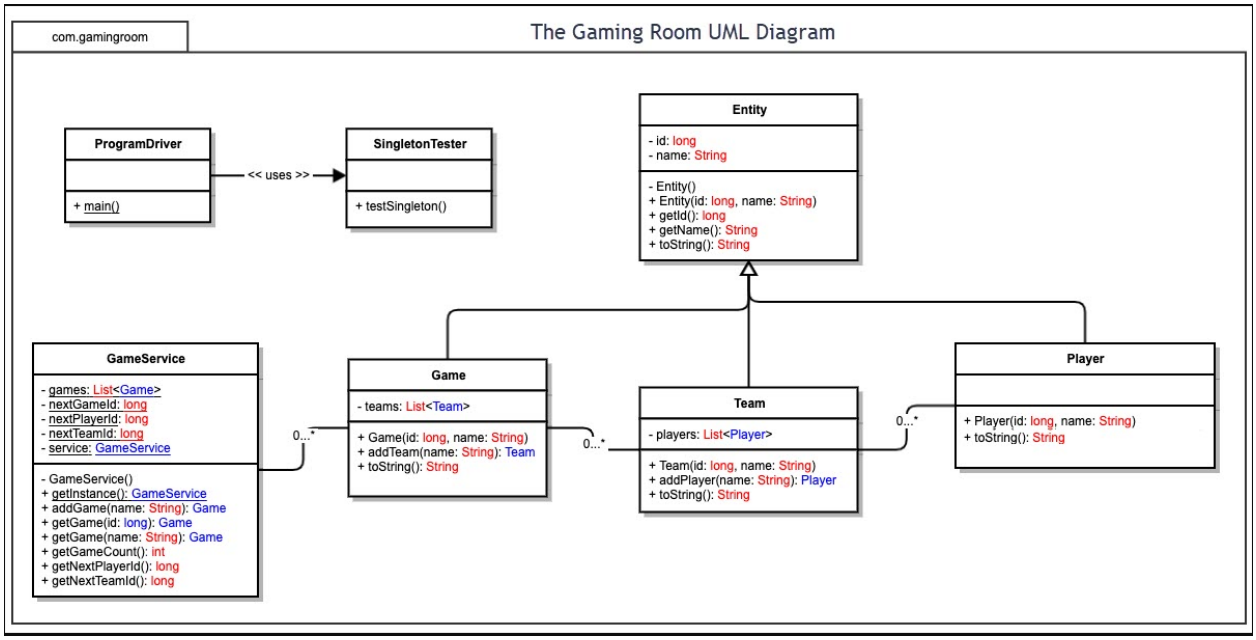
The Entity class serves as a basis for the other Game, Team, and Player classes to inherit from. An instance of the Entity, including the child classes, can only be created using an id and name in the constructor arguments. The default constructor is private and cannot be invoked for any classes, ensuring they at least have a name and id.

The Game class has an attribute that contains a list of Teams. It has a constructor, addTeam, and toString method.

The Team class has an attribute that contains a list of Players. It has a constructor, addPlayer, and toString method.

The Player class has no specific attributes, but only maintains the ones from the Entity class that it inherited from. It has a constructor and toString method. The Game, Team, and Player classes inherit the id and name attributes, and the getId and getName methods. The constructor and toString method is overwritten in each of these three child classes.

From this structure, we can see that the one and only GameService instance can have none or many Games, a Game can have none or many Teams, and a Team can have none or many Players. Each object that can “contain” another is done through an attribute that is a list of the other objects, also called encapsulation. The encapsulation allows an Iterator design pattern to use the data safely without affecting data directly. The abstraction of the methods allows the programmer to make games, teams, and players unique without knowing how it is implemented, saving time. Potentially, polymorphism could be implemented by making a container of Entities, and calling the “toString()” method on each in the list. Since the list could contain any child object, the proper “toString()” method is called depending on the specific class that object is a part of. These concepts are what allow the application to be structured to suit the needs of the overall design for uniqueness, efficiency, and programmability.



Evaluation

Development Requirements	Mac	Linux	Windows	Mobile Devices
Server Side	Mac has good security overall. This would help with cheaters on the game application, and stability from outside attacks. MacOS has to be purchased with the hardware, since it is licensed only through their official hardware.	Linux has good security overall. This helps with cheaters and attacks on the servers. The complexity can make it hard to troubleshoot the servers, however. Linux is free, so no licensing is required.	Has the least security of the three, but decent security nonetheless. The popularity makes it more likely to be accepted for various off-site server services. Windows needs to be licensed per PC system.	Has good security depending on the specific OS. Mobile is inefficient for hosting a web application, and may as well use the main operating systems for deployment and maintenance. Licensing is overcomplicated depending on the specific OS.
Client Side	Integrates with iOS platforms effectively. If Mobile Device platforms are	Due to the diversity in Linux distributions, let alone the operating	The popularity of this OS makes it imperative to make the game application run	iPhone and iPad combine effectively with MacOS, since the platforms are

	crucial, this OS makes it easier since it is integrated with one of the major mobile devices (iPhone). Development could be ported here to mobile easier than the other two platforms. Less popular than Windows, but still good for specialized development.	systems, it is imperative to use a REST api development process. This standardization will allow systems to run the game application without OS issues. This diversity makes Linux harder to work with on the client side.	on this OS. The REST api standard, as well as programming language standards (optional automating of a Java update for users) are necessary. Cost and time should be spent to make the gameplay as streamline as possible, since this OS is very popular, from a business standpoint.	made to work with each other effectively. Users on mobile expect the game to run similar to other OS. Expertise on performance and optimization of the game is necessary. Time should be invested to improve the UI of the game, since the smaller screens make it difficult to see multiple game features.
Development Tools	Apple has its own specialized comprehensive dev tool, Xcode IDE. Although different dev teams may be needed for this specialty, it does allow support for some mainstream languages. This IDE is free to download and use without licensing.	Linux is free, making it a great budget option for business constraints. However, the development tools are decentralized, and finding the right dev team will be difficult. Linux obviously can run any language though.	Most development tools are free to use on Windows. Due to popularity, finding dev teams for the app will be much easier and cost effective versus the other OS.	Performance is important, so C++ and C skills should be considered on the development team. Licensing costs will be considerable due to the various distributions of mobile platforms.

Recommendations

These are the recommendations for the client to execute the project successfully:

1. **Operating Platform**: The recommended operating platform is Microsoft Windows. This platform is the most common across development teams and users alike, and has the most flexibility. The platform's creator Microsoft also has its own cloud-based service available for use, making the setup costs and time much less compared to other platforms like Linux. MacOS does have its own cloud service, but the hardware limitations and less popularity make it slightly more hassle to expand and get setup.

2. **Operating Systems Architectures**: The Windows system allows for flexible development of various architectures. The architecture that should be used for this application is the client-server design pattern. This allows the client to run the app with minimal software requirements. The server will manage the data for individual games, players, and teams. The clients will manage the interactions and game processes that they each see. The interactions are consolidated to the server, then to the game instance, then each user will see a copy of the game instance in their browser of choice.

3. **Storage Management**: The storage needed for this application is managed by only putting the structure of the application necessary into disk storage. This includes the app's content such as images, team profiles, player profiles, and game history. These elements only need to be accessed occasionally for the game application to work on various platforms of users. Windows will allow for the flexible hardware upgrades in the future, should it be necessary. The small size of the images allow for many of them to be stored without much cost. History data is useful for developers and users, and can be stored as text or statistics without taking too much storage space. Because the overall storage can be small, the storage can be handled locally or over cloud, whichever is the better business decision.

4. **Memory Management**: The memory for this application is done on the client side, where each user should have information or a copy of the game instance on their platforms. This includes teams, users, user interactions, and game status. As a game session evolves, this data will change, and once the game is finished, the data should be deleted and minimal information moved to disk storage for history and useful statistics. Since the images and other data are arguably small, the images can be loaded according to what is easiest to work with. Whether that is loading information per game, per round, or continuously.

5. **Distributed Systems and Networks**: The system that should be implemented is the RESTful API software structure. This allows a distributed system of game sessions that can run on any platform, using a standardized communication method. This means the client must send the minimal information for the server to understand and send the correct information. The software in this system should be designed to be as flexible as possible, since it is unknown which operating system the users might use. The network for this design would then be

client-server, with many users connecting to a central server system. The servers need to be responsive to process all the information requests for the users. With a cloud-based architecture, the servers can be scaled according to user demand at almost any time.

6. **Security**: Due to a client-server architecture, the servers have to handle the user authentication and authorization. If the servers maintain the security, there are a few advantages and disadvantages. The advantage is that the user does not have to worry whether their account is secure on their end. All they must do is remember a password and other information. The user does not have to concern themselves with the security of their own system. The disadvantage is that if the server is breached, all the users on that server are exposed to abuse. The server must be secured as best as possible and minimize damage when it occurs using user roles and the principle of least privilege. In conclusion, servers maintaining security is the best way to handle issues, since the developers have access to this system, versus not having access to individual users when they have problems.