Draw It or Lose it

CS 230 Project Software Design

Version 3.0

Table of Contents

Version	Date	Author	Comments
1.0	11/9/24	Creston Getz	First version created; problems identified.
2.0	11/26/24	Creston Getz	Evaluation of different operating systems
3.0	12/8/24	Creston Getz	Recommendations are discussed.

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.

<Write a summary to introduce the software design problem and present a solution. Be sure to provide the client with any critical information they must know in order to proceed with the process you are proposing.>

The Gaming Room wants to develop the game win it or lose it to be on a web-based platform currently only available on Android. The underlying software must have certain requirements given by the client. Based on the requirements OOP and or design patterns should be used. To ensure a game can have many teams and a team will have many players, OOP composition should be used. To ensure every game, team, and player is unique the iterator pattern should be used to look for existing names. For only one instance of a game to exist in memory, the singleton design pattern should be used. Or having unique identifiers for the game, team, and player classes. To proceed with this solution other considerations such as budget, platform compliance, and legal issues should also be discussed with the client. The desired end solution for the user needs to be discussed.

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.>*

Given technical requirements by client

- A game will have the ability to have one or more teams involved.
- Each team will have multiple players assigned to it.
- Game and team names must be unique to allow users to check whether a name is in use when choosing a team name.
- Only one instance of the game can exist in memory at any given time.

Design Constraints

Hardware will be discussed later based on software decisions. Designing the game for app and web-based use will have different constraints.

- If we only want one game to exist in memory at a given time. The singleton design pattern should be used for the game class. Or unique identifiers for every game, team and or player.
- The iterator design pattern can be used to verify every game, team and player name is unique. This should be done on the addClass() methods. It should iterate over the lists and search for existing names. What should the method do when an existing name is found?
- Composistion and encapsulation should be used for the teams and players classes. By using a private list of objects and methods to manipulate them, encapsulation can be maintained. Not only will this support the iterator pattern, but it will also ensure a team can have many players and a game can have many teams. The client does not mention a limit on teams or players.

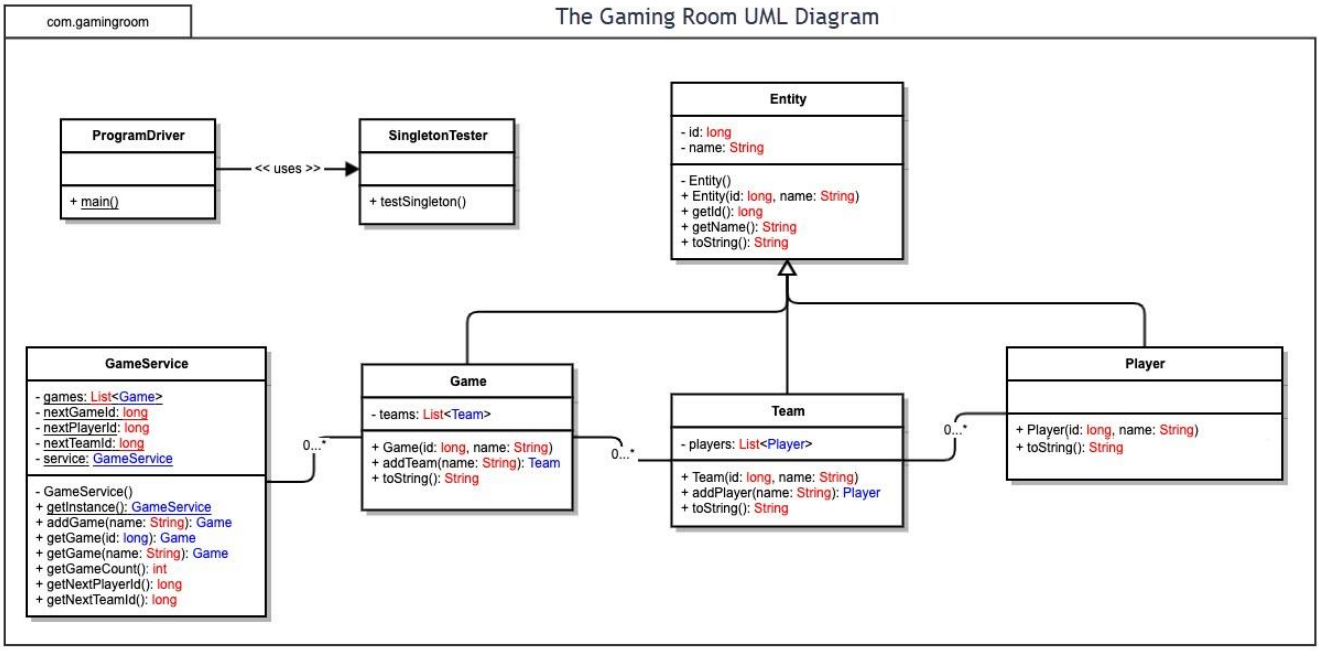
- The client wants the web-based version to serve many platforms. Does that have any compatibility constraints? Can players play cross platform? The current game is an Android app. Will the web-based version be able to play with them?

- Due to many platforms are their compliance constraints? Does web based have any conflicting rules compared to Android?
- Is the client on a schedule or do they have a budget?
- Are there any team constraints? What is the skill level of the developers? Do they have the necessary knowledge?
- Are there any legal constraints? Does client have any legal constraints on top of platform compliance?

System Architecture View

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.

The UML diagram below uses many OOP principles. This will help ensure that the software requirements are met. Starting with GameService which uses the singleton design pattern seen with getInstance(), this ensures that only one GameService can exist at a time. Unlike GameService there can be many games, team, and player objects. The 0...* between them indicates a 0 too many relationships. GameService can have many games, but many games can only have one GameService. These relationships help fulfill the software requirements, a game can have many teams, and each team can have many players. It also ensures that game and team names are unique. On a team there is only one list of players, and a game has one list of teams. This is an example of encapsulation and composition. Using an Inheritance game, team, and player inherit from entity. It allows for code like getId and getName not to be repeated. It also enables the toString method to be overridden based on the class/object it is used on.



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	Advantages • Secure • Flexable • Developer friendly • Easy-to-use • Some-third party-cloud support • Cheaper licensing costs Weaknesses • Expensive • Limited scalability • Lacks native cloud support (macOS sever) Characteristics • Runs only on apple hardware • UNIX-style system • Designed for client side • Expensive hardware and not ideal for scaling	Advantages • Free OS • Highly scalable • Secure • Extensive framework support • Lots of sever support like Apache and SQL • Extensive cloud support such as AWS • Open source • Vast documentation Weaknesses • Hard to use and setup Characteristics • Most common OS for webhosting • Lacks a GUI • Cheapest option (both licensing and hardware)	Advantages • Integrates with Microsoft cloud and sever tools like Azure, IIS and MSSQL • Scalable • ASP.NET apps well supported Weaknesses • Security • More Expensive than Linux • Not open source • Harder to learn than macOS • Requires more hardware Characteristics • GUI • Powerful with Microsoft tools • Expensive licensing costs	Advantages • Cost effective • Easy to use • More portable • Energy efficient Weaknesses • Performance • Storage • Not designed for server deployment. • Security • Scalability Characteristics • Best for testing very small or local websites • Portable and may be used remotely • Not ideal for web hosting in most cases • No licensing costs

Client Side	MacOS will require general knowledge of macOS and languages such as Swift. MacOS is often updated so experience with maintenance will be needed to keep the game running. Testing on all web browsers to ensure the game works is also required. The team should know various web frameworks such as Rails or Django. MacOS has tools to do this such as Safari Web Inspector. Development should be streamlined and quick. However, due to the lack of tools like native cloud support. Development time may be slowed. MacOS for smaller applications can be cheaper, however when you put it scale it can be extremely expensive.	Linux is harder to use than MacOS and Windows. It is the cheapest option. However, it will require developers who are experienced with Linux. Developers should also have experience in tools such as Node.Js and Java.Spring and DevOps. Back-end development is key to ensure the game will run on an HTML interface on all platforms. Development time can vary depending on the developer's skill. However, due to vast documentation and open-source tools, it can be fast. And to keep the game running experience with maintenance and testing tools will be needed.	Windows can be more expensive than Linux but cheaper than MacOS. The software to use Windows such as VS code can be costly for the enterprise edition. Developers will need experience in tools such as IIS or .NET and ASP.NET. A server license is often needed to host a server on Windows. While VS code and IIS come at a cost there are free open-source options available. Windows like MacOS is updated often so maintenance is important. To ensure that the game runs on all platforms knowledge of testing tools is key. Windows provides browser testing tools for a cost. Development time will likely take the longest of these four.	Mobile devices would not be ideal for supporting multiple clients. Developers will need experience in mobile languages such as Xcode, Android Studio, and cross-platform ones like Flutter. While mobile devices would be cost-effective compared to the other options, they lack many tools the other OS have like native server hosting and IDE's. For a streamlined mobile experience, APIs should be used to ensure the game works on many platforms and mobile devices. Mobile devices, while not ideal for webhosting, would be cheaper than Linux.
Development Tools	MacOS provides a lot of tools right out of the box which will help speed up development. MacOS is written in Swift, and the OS can support a wide range of IDEs such as VS code, Xcode, and	Linux can support most IDEs, languages, and tools. For web development, some common ones are Node.js, Python, HTML/CSS, java, and more. Linux IDEs include Eclipse, VS code, Git, Docker,	Windows utilizes languages such as C#, Python, HTML/CSS, and Node.Js. It can use most IDE Linux and MacOS can. These include VS code, eclipse, Docker, and IIS. IIS is important because it	Mobile requires a distinct set of skills from developers. The platform uses many different languages than other OS. Some of the most common are Kotlin, Swift, dart, and other languages such as HTML/CSS

	Eclipse. VS code would be ideal because it would support web languages such as JavaScript and CSS. Most of these tools are free on MacOS however, some have a cost. Publishing apps on IOS costs $99 a year. To build an app that can be supported on multiple platforms many development teams may be needed. One for the front and back end and one specifically for mobile.	and Nginx/Apache. Apache is specifically important because it can host web apps and manage HTTP requests. Allowing the game to run on any platform using methods such as RESTful APIs. Most of these tools are free to use and have no licensing costs. If the game is to be on many platforms, then a few teams of developers may be needed. A front, back, and quality team.	is a native web server hosting application for Windows. It uses .NET and PHP. If the team decides to use IIS, then developers will need knowledge of these languages and IIS. IIS itself is free to use with Windows, but it costs $500/server. Having developers handle the server as well as the front end and a team to handle testing will ensure the game works on all platforms.	and Python. Mobile requires the use of different tools and IDEs than other OS. For Android, the team can use Android Studio, and for IOS, Xcode. The team should consider Unity which is used to make games often in C# or Java. Unity can cost around $400/year, but it would drastically improve the mobile experience of the game and other platforms.

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**: <Recommend an appropriate operating platform that will allow The Gaming Room to expand Draw It or Lose It to other computing environments.>

   Linux will allow the Gaming Room to expand to many computing environments. It is the most common operating platform for servers for a reason. It offers the most secure, flexible, and scalable option. Due to Linux's flexibility, it supports deployment to other operating systems which is ideal for the Gaming Rooms goals.

2. **Operating Systems Architectures**: <Describe the details of the chosen operating platform architectures.>

   Linux is a layered open-source UNIX-style operating system. Each layer performs a separate task. The kernel controls the activity of the hardware of the system. While there are many distinct kinds of kernels Linux can use, they share the same goal. To provide a low-cost or free and flexible operating system. Linux is most commonly a monolithic kernel, which is where all the system's core functions are managed by the kernel. After the kernel is the shell which connects the kernel to the user. In Linux servers, unlike other OS servers, the shell is a command line interface, not a GUI. The main benefit of the layered nature of Linux is that features can be added without downtime. One layer can be changed or updated at a time.

3. **Storage Management**: <Identify an appropriate storage management system to be used with the recommended operating platform.>

   On the server side, the game and user/team data will need to be stored. This data is not static and will change often. For example, players may leave teams or pictures for the game will be added or removed. Linux offers many kinds of storage management systems and methods. To store or allocate the data an indexed allocation would be ideal. This will reduce external fragmentation and allow data to be stored in a different order. For example, a team can continue to grow even if a block of the disk is filled. For managing storage, this is a very scalable method. To look for data and reduce seek time LOOK scheduling should be used. A space map could be used to help reuse space in the database. This would manage unused space using something called metaslabs to divide the space.

   Another ideal option is a cloud-based model. A cloud model combined with an access method such as direct access would allow for quick reading and writing. Also, the storage methods are abstracted and only what is used is paid for. Common providers are AWS, Google Cloud or Azure. All of which are compatible with Linux. Cloud models are not only ideal for more efficient storage management, but they also provide backups of data.

4. **Memory Management**: <Explain how the recommended operating platform uses memory management techniques for the Draw It or Lose It software.>

Linux has a low overhead allowing more resources for the game. It also has many tools to handle memory we can utilize for Draw it or lose it. First, to prevent the memory from being overloaded when multiple games are in play and allow the images for the game to load quickly, they should be cached. Caching works by loading the images only for the current round into memory. This is much more efficient than loading the entire library or images that are not needed. Linux uses paging methods and virtual memory to increase the amount of usable RAM. Demand paging is like caching but for pages, only the memory pages that are accessed are loaded. To scale to many plays and games Linux provides dynamic memory allocation. This is one of many reasons Linux is a popular server operating system. By using low-level libraries and functions memory can be allocated as the games progress. For example, memory can be added or freed when a player leaves or joins a team.

5. **Distributed Systems and Networks**: <Knowing that the client would like Draw It or Lose It to communicate between various platforms, explain how this may be accomplished with distributed software and the network that connects the devices. Consider the dependencies between the components within the distributed systems and networks (connectivity, outages, and so on).>

For draw it or lose it to communicate on various platforms we can use a distributed system or network. We have already discussed RESTful APIs as a cross-platform communication method for the game. RESTful APIs provide a standard method of communication between all devices using HTTP, and it separates the client side from the server. This means the server only handles the game's data while the client would handle anything they need based on their operating system. Allowing the game to work on any platform or browser so long as it uses the same method of communication.

The client-server model does have some downsides. If the server goes down then the game cannot be played, it has a single point of failure. Also, if a user is far away or in a remote location and has poor connectivity to the server, the game's performance will suffer. A solution to this is to have many servers which with cloud computing is feasible even for small organizations. This expands the area of connectivity and increases the points of failure, and with most providers extensive data backups and security. By using a cloud provider, the game can handle varying loads of players on a wide network range.

6. **Security**: <Security is a must-have for the client. Explain how to protect user information on and between various platforms. Consider the user protection and security capabilities of the recommended operating platform.>

Security is an important consideration for any application. Linux is one of the most secure operating systems. Because the game will use a client-server model, protecting the data transfer between the two is particularly important. To do this, we can use a method of encryption like TLS/SSL or end-end. To protect the Linux server there are many tools we can use. Linux offers access control which limits who and what can read, write, or execute files. This makes it difficult for a malicious application to fully control the system. It also helps to implement two significant security features; RBAC and the least privilege principle. RBAC or role-

based access control limits what certain user logins can do. For example, there are user and admin logins. The least privilege principle states that the user should only have access to what they need. So, in the case of our game, a player should not be able to access sensitive data. This reduces the impact of human error which is often the largest security risk. Methods like MFA, biometrics, and other layers of security are beneficial. The more layers the better. We previously discussed using RESTful APIs for cross-platform communication. Securing these APIs with methods like OAuth2 is also important.

We can also increase data security by collecting and storing the data safely. One of the best ways to do this is only to collect the data needed for the game. Which likely will not be sensitive. Another strategy to reduce a data breach's impact would be removing any identifying information from it. As we have discussed, there are many ways to implement security to protect the user. If a cloud model is used, then the security of the server or hardware is the provider's reasonability.