The Gaming Room System

CS 230 Project Software Design Template

Version 3.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	02/19/2020	Connor Brereton	Added verbosity to the 1-6 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

The company, The Gaming Room, needs to develop a web-app that is platform independent. Essentially it needs to have binaries built that can run in production on multiple platforms. The Gaming Room has a game called *Draw It or Lose It* which only works on Android. They want to be able to run it on MacOS, Linux, Windows, etc. The storyline of the game is that multiple teams made up of a handful of people each will go for four rounds of guessing what an image represents. It's a lot like charades. In the instance where the team cannot figure out what it is the other team across from them will get to answer until the 15 second mark runs out.

Design Constraints

- Each game has M-number of teams.
- Each game team has to have a unique ID in the form of a team name.
- Each team has to have N-number of people.
- In the gaming system there can only be one instance of the game (program or chained programs) running at a time.
- The game binary needs to be able to run on various platforms so that it can service a diverse set of users.

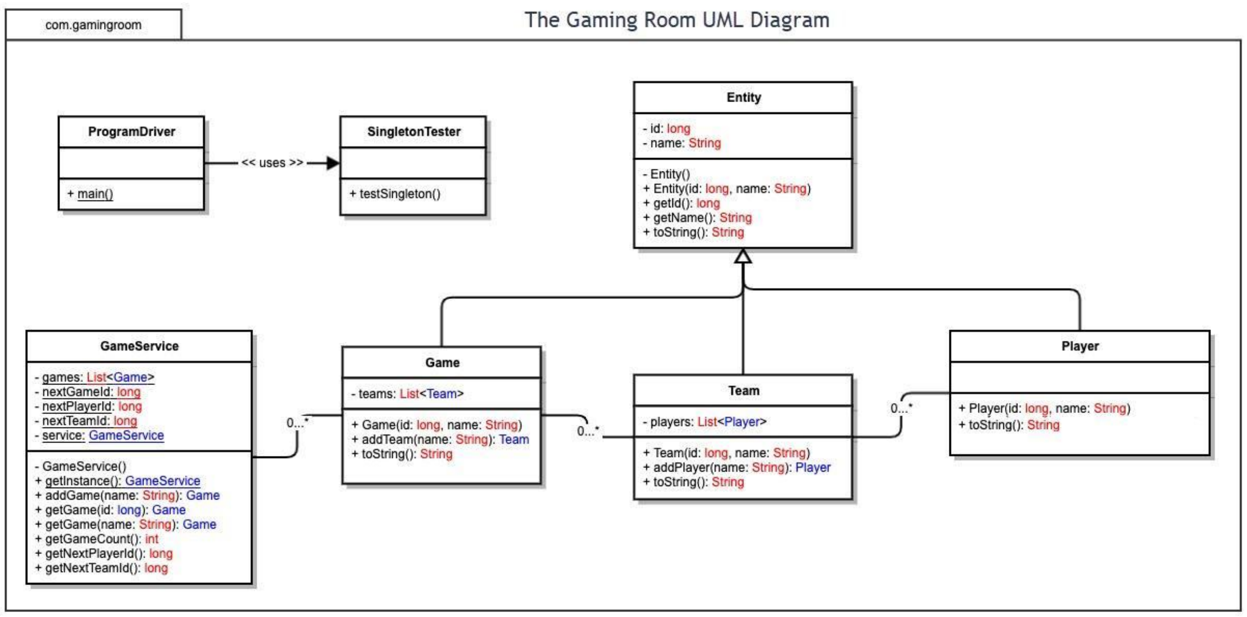
These are just the high level requirements for a developer to take a look at before they start writing all of their high level architecture documents and then write the code. The most important part of this spec is that the program runs across all platforms. In order to port the software from one operating to several others this is going to be a very complicated project. The most important thing to do in this case is use TDD to build a robust test case that will test each case of the original code in great detail. Then as you build each feature for the other operating system you can test in real time that it is working. This is a very arduous process but it's pretty much the sure fire way to make sure that you have a truly working solution for the operating system.

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.

Domain Model

We will step through the UML document from the top down to show how our program is formatted. The *entity* class creates a relationship between the *game*, *team*, and *player* classes in the sense that they all inherit from a super class. In the super class there gets defined a "name", "id", and a few other parameters in other cases. These are constructed and deconstructed using something called a getter and setter. Entity is a super class in this case. Team and player are what's called a "has a" type. Game has Team relationship. GameService has a Games relationship. These a child and parent class relationships. in UML there is something called aggregation which is what forms the *has a* relationship. All a *has a* relationship really is is one where a class calls another class. The diagram show us this like it did in the previous point about the parent child relationships.



Evaluation

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	UNIX based system that has flexibility and is developer friendly. Ease of use into the server to quickly make changes to the system.	Linux is better than MacOS because it has many features that are unrivaled. You don't have to use Homebrew to install a bunch of features.	More features and channel products to use that make it really easy to buy. They have simplified the buying process.	The server being immobile is a very important feature that cannot get overlooked. This is because they want to be able to track different devices in order to get more information about the whereabouts. It's also important to note that the specifications are better in other devices.

Client Side	The cost is roughly the same as doing a Windows system. It's a more time intensive application process that is niche.	Linux is going to take the longest and require the most experienced developer in order to complete. The cost is cheapest though since it's an entirely open source solution.	The least expertise and time required. Very fast to get up and into production.	Harder to get up and running but you can do updates on the fly in real time, you get amazing analytics on user behavior, and it's really just the future of where things are.
Development Tools	GitHub for the source code management. VSCode for project development. The programming stack would be MERN and use Docker for containers and Kubernetes for building and scaling our distributed system.	I would use all of the same tools that I mentioned to the very left along with some customer package management tools that are specific to Ubuntu Linux.	Much more complex to run than Linux since Windows is NOT UNIX based. UNIX is a great standard to learn for things such as application monitoring, high performance computing, etc.	For mobile I would just build this out in React Native since it's the least painful port over from pure Javascript. The React Native would run as a container in any environment because it can be built upon any operating system from how the Dockerfile is configured.

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**: I would advise The Gaming Platform to start off on Linux Ubuntu 14.04 LTS because it is the most stable release, highly documented online for troubleshooting, and so many other performance and monitoring reasons. Moreover, with Linux you have a ton of free packages and libraries to piggyback on top of. Also since Linux is UNIX based the port to MacOS would be pretty straightforward and simple.

2. **Operating Systems Architectures**: Architecturally going with Ubuntu 14.04 is a great call because it works extremely well with other tools needed for building high performance systems such as Docker, Kubernetes, etc. It's going to be pretty straightforward for building out a SOA from the start so that scaling comes way easier later on. The other great thing about an Ubuntu instance is that they are so easily spun up in AWS that it makes TTL (time to launch) much faster.

3. **Storage Management**: Storage management is a pretty broad topic. For managing IAM and compliance I would use a tool like Okta or Auth0 which is a cloud IDaaS tool. This is the solution for managing the storage of users which is a huge technical chunk of this project. For pretty much every other part of our infrastructure I would go with AWS all day. AWS is very affordable and it's the gold standard when it comes to cloud computing resources for infrastructure. S3 has many features built for managing objects, lifecycle configuration, cost allocation, etc.

4. **Memory Management**: I would use AWS SimpleDB since it is the easiest and cheapest solution right now. I would also look into using some kind of CDN in order to speed up the delivery of these image assets to the client. This will also boost Google SEO ranking. Because we are using AWS SimpleDB we can use the following features to monitor, scale, and manage our memory: high availability, auto-scaling, and zero administration.

5. **Distributed Systems and Networks**: When it comes to the distributed system I was thinking the best way to design this is as follows. You would have a *dev* and a *prod* cluster for each platform type. You can control the development and production on each cluster where each cluster is built on Docker images that are OS dependent. So you'd have a MacOS, Linux, and Windows cluster. Each of these clusters would be controlled by a load balancer that sits at the application layer and routes web requests to the appropriate operating system. You do this through reading which OS the request is coming from on the HTTPS request. On this request you read one of the bodies and it controls what goes where in your distributed system.

6. **Security**: Security is a really strong point of Ubuntu 14.04 LTS. I think the most important thing is going to be using some form of CIAM solution to control access to assets so that there is no compromised data. The IDaaS solution I would use is Auth0 since it is the most developer friendly. For data protection AWS has features that guard access to data, accounts, and the workload. For infrastructure protection AWS has features that allow you to set up rules for traffic filtering such as IP address blocking, HTTP headers, HTTP body (payload), and URI strings. Threat monitoring is also a big part of the monitoring inside of the environment. Compliance and data privacy are also managed by AWS in a dashboard that tells you what you're missing and what you're not up to date with. This is huge for a compliance tool.