Draw It or Lose It

CS 230 Project Software Design Documentation

Version 1.2

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	03/26/2024	Zane Deso	-Summarized software design problem with presented solution. -Identified design constraints for web-based distributed environment. -Explained implications of design constraints.
1.1	04/03/2024	Zane Deso	-Evaluated various systems, platforms, developer tools and security measures.
1.2	04/23/2024	Zane Deso	-Provided recommendations for system architecture that should be used in developing this software.

Executive Summary

Creative Technology Solutions (CTS) has embarked on a journey to create a web-based game for their new client, The Gaming Room. The Gaming Rooms want to develop a web-based game that serves multiple platforms based on their current game, Draw It or Lose It, only available through Android. The staff at The Gaming Room have limited knowledge of how to set up the environment, get the development started and their overall requirements and/or constraints.

Requirements

The software requirements laid out by The Gaming Room are as follows:

1. A game will have the ability to have one or more teams involved.
2. Each team will have multiple players assigned to it.
3. Game and team names must be unique to allow users to check whether a name is in use when choosing a team name.
4. Only one instance of the game can exist in memory at any given time. This can be accomplished by creating unique identifiers for each instance of a game, team, or player.

Design Constraints

The design constraints as identified by CTS are as follows:

1. The game must run on a web-based platform: This implies that the game will need to be responsive and robust utilizing technology such as HTML5, CSS3, and JavaScript.
2. The game must be responsive facilitating real-time interaction: This implies that the game's architecture may need to use web-sockets to foster an interactive environment seamlessly between the game services, within the teams, and for the individual as well.
3. Uniqueness must be maintained for game instances, team names, and player names to avoid conflicts and ensure data integrity: This implies that a robust system to generate unique identifiers and check for uniqueness before creating new instances, involving backend logic and a database to track these identifiers.
4. The application must be capable of scaling to support a potentially substantial number of concurrent users and games without significant performance issues: This necessitates a scalable architecture such as cloud-based services, and an optimized back-end environment for efficient DB queries
5. Security and data protection is also a constraint due to the nature of the multiplayer platform functioning on the web. Common web vulnerabilities should be considered, and these vulnerabilities should be upended using security and data protection best practices.
6. The codebase ideally should be built in a manner that is easily digestible, maintainable, and scalable for prospects and for The Gaming Room to seamlessly pick up where CTS personnel left off

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.

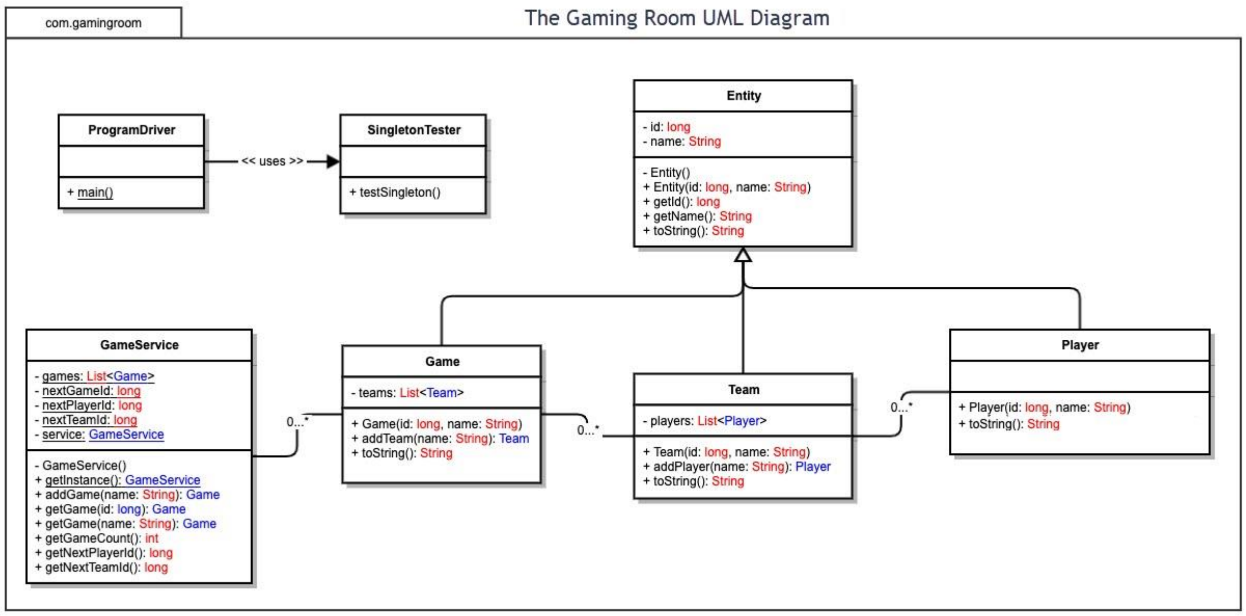
Domain Model

The Gaming Room UML Diagram depicts many rectangular classes within the greater com.gamingroom service. The classes and relationships are as follows:

- **Entity Class**: This is the base class with common attributes like id (of type long) and name (of type String). It also has a constructor Entity(id: long, name: String) and common methods getId(), getName(), and toString(). It acts as a superclass or parent for other classes, providing common data and behavior that they inherit.
- **GameService Class**: This class is responsible for the game logic and maintaining the list of games (List<Game>). It has a singleton pattern implemented, indicated by the private constructor GameService() and a static method getInstance() that returns the single instance of GameService. It also has methods to manage games and IDs for the game, players, and teams, like addGame(), getGame(), getNextPlayerId(), and getNextTeamId().
- **Game Class**: It inherits from the Entity class and has a list of Team objects (List<Team>). It also provides methods to add a team (addTeam()) and override the toString() method.
- **Team Class**: This class inherits from Entity and contains a list of Player objects (List<Player>). It allows the addition of players with the addPlayer() method and overrides the toString() method.
- **Player Class**: It also inherits from the Entity class and overrides the toString() method. This indicates that each Player is an Entity with a unique ID and name.

The Object-Oriented Programming Principles that are exemplified in the UML are as follows:

- **Inheritance**: Inheritance is used here as the Game, Team, and Player classes inherit from the Entity class. This promotes code reuse and relationship representation.
- **Encapsulation**: Encapsulation is applied by keeping some data private within the classes and providing access through public methods. This is seen in the GameService class, which encapsulates the logic for game management and maintains control over how game, player, and team IDs are assigned and accessed.
- **Abstraction**: The Entity class provides an abstraction for the shared characteristics of all entities in the game. The concrete details of how each Entity operates are defined in the subclasses.
- **Polymorphism**: This is achieved by the ability to override the toString() method in each subclass, providing a specific representation for a Game, Team, or Player.
- **Singleton Pattern**: The GameService class uses the singleton pattern to ensure that only one instance can exist in memory, fulfilling the requirement that only one game service is active at any given time.
- **Modularity**: The system is modular, with separate classes for different entities and services, which makes the system more manageable and allows individual parts to be developed and tested independently.



com.gamingroom
The Gaming Room UML Diagram
ProgramDriver
+ main()
<< uses >>
SingletonTester
+ testSingleton()
Entity
- id: long
- name: String
- Entity()
+ Entity(id: long, name: String)
+ getId(): long
+ getName(): String
+ toString(): String
GameService
- games: List<Game>
- nextGameId: long
- nextPlayerId: long
- nextTeamId: long
- service: GameService
- GameService()
+ getInstance(): GameService
+ addGame(name: String): Game
+ getGame(id: long): Game
+ getGame(name: String): Game
+ getGameCount(): int
+ getNextPlayerId(): long
+ getNextTeamId(): long
0...*
Game
- teams: List<Team>
+ Game(id: long, name: String)
+ addTeam(name: String): Team
+ toString(): String
0...*
Team
- players: List<Player>
+ Team(id: long, name: String)
+ addPlayer(name: String): Player
+ toString(): String
0...*
Player
+ Player(id: long, name: String)
+ toString(): String

Evaluation

Development Requirements	Mac	Linux	Windows	Mobile Devices
Server Side	Server-side Mac operating platforms offer stability and security through its UNIX based system. Supports many languages such as Python, PHP, Ruby, and Java. Weakness: Higher cost due to requiring apple hardware.	Server-side Linux architecture offers security as well as customizability due to its open-source nature. Linux supports an ecosystem of software such as Apache for web servers and is a popular choice for backend technology using Python, PHP, and Node.js. Linux has a primary advantage of having a lower entry cost threshold. Weakness: CLI steep learning curve.	Server-side Windows operating platforms offer a friendly use environment, support for .NET and Microsoft SQL servers. There is support for various languages and frameworks which makes it a versatile choice for web application development. The server side has graphical based management tools, which are easier to use from the get-go. One of the main costs is the licensing fees associated with a copy of windows. Weakness: .NET and Microsoft SQL servers are more vulnerable to malicious attacks.	The use of mobile devices as a server-side operating platform is not suited for production but may be used locally or for prototyping. Cost might not be an issue, but the limited nature of scalability, storage and reliability will be in constant question. Weakness: Not well suited to a production environment.

Client Side	For client side, Mac platforms offer consistent environments between any device, and support for any type of development. Native support for Xcode and is compatible with every web-based development tool and web browsers. Weakness: Higher Cost incurred from requiring Apple hardware.	For the client side, Linux is again customizable due to being open source which allows for free tools for development. Linux supports many browsers for testing web apps and allows for tools like Docker for containerization. Since Linux does not require specific hardware the cost to get it running on any device is only the time spent installing and setting up the environment. Weakness: Steep Learning Curve	Client-side Windows offers a versatile set of tools that offers a tool for any skill level, with varying pricing. Microsoft has native support for tools like Visual Studio IDE which can code, debug, and test. Windows supports every browser enabling web development to go off without a hitch. The great part about windows client environments is there are many different tiered versions of their licenses. This model will enable developers to only pay for what they need. Weakness: Limited cross platform tools available	Client side, not exactly the best environment for development but for the user it can be accessible. There are some advantages of mobile devices during the development of a program. Tools like React Native or Flutter allow developers to build mobile applications using web technologies that can run natively on mobile devices, facilitating cross-platform compatibility. Testing on actual devices is crucial for understanding user experience and performance. Weakness: Development Subscriptions, and large variation in screen sizes.

Development Tools	Development tools on Mac are geared to developing with Objective-C or Swift but Mac offers support and tools for any type of development. Beyond mobile, Mac supports web development with tools for JavaScript, Ruby, Python, PHP, and more, making it versatile. Weakness: Abstraction from complex processes and tools may oversimplify concepts for new developers.	Linux's development tools cater to Python, Java, C/C++, PHP, and JavaScript. Many choices of IDEs are available. Such as Eclipse, NetBeans, and JetBrains' suite (e.g., PyCharm, IntelliJ IDEA), along with lightweight editors like Vim and Emacs. Many distinct types of compilers are available. The platform is also ideal for development involving Docker and Kubernetes for containerization. Weakness: Steep platform learning curve. Management of dependencies by developer.	Windows offers its developers tools to use C# and .NET frameworks. These tools lend themselves to a desktop application focus although many tools are available for any kind of development. Visual Studio is one of the main IDEs associated with Windows. Weakness: Security. Windows relies heavily on its own services and tools which limits the customizability of using it for a development environment.	Tools available for development of and on mobile devices include languages such as Kotlin, and Swift and are helpful to create programs that run great on IOS and Android systems. Dependency management is a major consideration for this type of development. Weakness: Limited tools, set in stone guidelines for production level applications and even cost of initial distribution via subscriptions. Not to mention platform specific guidelines must be met to roll out a mobile application on each app store.

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 operating platform that I recommend is Mac OS given the versatility of the type of applications it can support, and the well documented nature of the environment would shorten potential development time if solutions had to be otherwise created rather than uncovered. MacOS will lend itself to web-based development as well as mobile development. The ease of integration with external APIs that the application may need is a huge plus. One major hiccup to lookout for is any compatibility issues during the rollout of the app to many distinct types of browser environments

2. **Operating Systems Architectures**: The operating system to be used should also be MacOS for the same reasons stated above. For the development of the application, the MacOS-based architecture will ensure that resources are easily accessible, relevant, has ample community support and there are many third-party companies specializing in developing in this architecture. This will aid itself to the possibility of outsourcing some of the work while also remaining compatible on the in-house side of the development.

3. **Storage Management**: For storage, I recommend a cloud-based system for automatic scalability, pay for what you use model, and maximum UX with low latency responses and general playability from anywhere. Data congruency will also be a major factor in choosing this option. Let us use Azure Cosmos DB for this. Compared to other models which may require more development to get functionality in place or require proprietary specific knowledge to navigate Azure Cosmos DB will be a low hanging fruit that the whole team can get behind.

4. **Memory Management**: Memory management will not be as much of an issue given our use of the Mac platform, operating system, and Azure Cosmos DB. To clarify this, memory management in the macOS ecosystem as well as with Azure Cosmos DB, is done automatically without any interaction from the developers required. The less time fixing issues with memory management means more time to create a responsive product that everyone will love. MacOS touts automatic reference counting (ARC) which handles the automation of memory management of objects. ARC reduces memory leaks by deallocating objects from memory when not in use. MacOS also has a system level memory pressure relief functionality. What this does is compresses the memory of less-active applications on the system to free up memory for more actively used applications. Azure Cosmos DB is designed for global distribution which relies on automated replication of data across many geographical locations to minimize latency thereby increasing a responsive application.

5. **Distributed Systems and Networks**: By relying on a serverless architecture we minimize the need to worry about connectivity, outages and so forth given that all this technical measure will be taken upon by Azure as it happens. In the backend we will ensure to write protective functions in the case of one of these events and ensure to call the correct Azure functions to shift if needed.

6. **Security**: To ensure we are covering all bases, I recommend allocating as much time and resources to security as possible. We will need to ensure we are serializing and deserializing to

ensure data in transit is not attained or accessed erroneously or maliciously. We may also conduct regular and random security audits of each part of the distributed system to ensure 100% secure coverage. We can task a team of developers to act as the penetration testers to gauge our level of security and find any vulnerabilities before any harmful actor may find it first. Moreover, developers shall adopt secure coding practices, and this shall be outlined in the style and security guide to set the standard across the board for in-house and third-party developers. Our authentication can be handled by Azure B2C, which will take care of the authorization security protocols in an encapsulated environment away from the rest of the system. As an added benefit much of the set-up process for Azure B2C is straight forward and requires minimal coding to achieve maximum security standards during authorizations by all users. Beyond that, security is not needed as we recommend not collecting any data about the users effectively nulling out the chances of any bad outcome from the event that any part of the program is penetrated with malicious intent. The protection of the user at this point falls onto the user itself, as well as the platform and operating system that the user has chosen to use. As far as protecting the asset of the game itself from being accessed erroneously or maliciously, we will ensure that the development process follows the industry best practices for security and data protection. OOP principles will abstract and encapsulate protecting the software, no data collection means nothing to worry about as far as the user and the rest falls onto the systems that the program runs on to protect the user furthermore.