Draw it or Lose it

CS 230 Project Software Design Template

Version 1.2

Table of Contents

CS 230 Project Software Design Template 1

Table of Contents 2
Document Revision History 2
Executive Summary 3
Requirements 3
Design Constraints 3
System Architecture View 3
Domain Model 3
Evaluation 4
Recommendations 6

Document Revision History

Version	Date	Author	Comments
1.0	07/16/23	Oved AYDIN	Find solutions by examining the needs of the client and comparing solutions.
1.1	07/30/23	Oved AYDIN	Evaluated various platforms for the project.
1.2	08/13/23	Oved AYDIN	Explored and defined the recommendations for the project.

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

Our client, The Gaming Room, has a game called “Draw it or Lose it”. They currently have a single version of the game for Android devices. They want a web-based version of the game to expand their user base.

Requirements

- A game could have one or more teams involved.
- Each team will have multiple players.
- Game names and team names must be unique. Users can check the names.
- Only one instance of the game can exist in memory at any given time.
- The gaming app should be web-based.

Design Constraints

The web-based version of the game should have only one instance of the game in memory. Teams should be able to have multiple players. And we need unique names for games and teams. The game must be deployed on a server to handle clients with different operating systems.

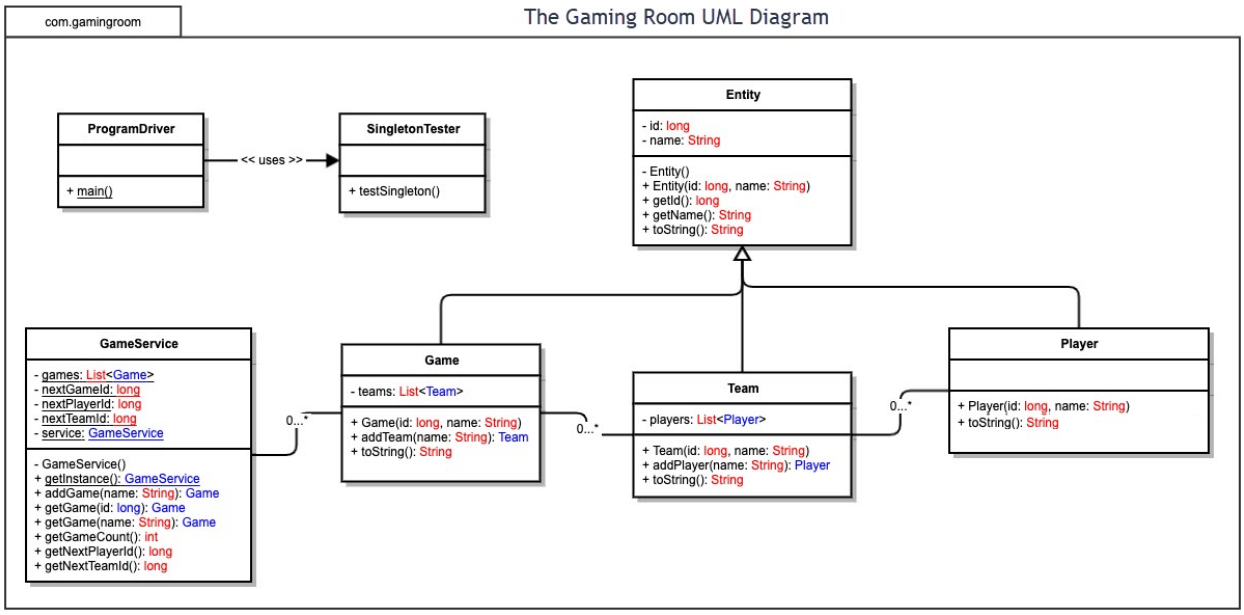
System Architecture View

1. Presentation Layer:
    a. Web Browser Interface: This layer handles interactions with the game as a web page.
    b. Native Interface: This layer handles interactions with the game as a native app
2. Application Layer:
    a. Game Management: This component handles the lifecycle of game instances (creation, assigning teams and sessions)
    b. Team Management: This component handles assigning team players and team names.
    c. Player Management: This manages player information.
3. Data Layer:
    a. Database: This component handles any persistent data (scores, previous games)
    b. Data Access Layer: This component handles operations to manipulate or retrieve the data in the database.
4. Infrastructure Layer:
    a. Web Server: This server hosts the game application and serves the game to the users.
    b. Application Server: This server runs the game server application, handling the business logic, and processing player requests.
    c. Database Server: This server hosts and manages the database.

Domain Model

We see that Game Service could hold multiple instances of games since we want a single Game Service instance in memory at any time. And the Game could hold multiple instances of teams since we want a single instance of a game in memory. One of the requirements is for a team to have multiple players and

we see that relationship too. Game, Team, and Player inherit a class called Entity for simplicity since all their instances require id, name, etc. We also have a Program Driver and Singleton Tester to test the classes.



Evaluation

Development Requirements	Mac	Linux	Windows	Mobile Devices
Server Side	• Possible • Not common • No licensing cost • High hardware cost • Build time is long • Less documentation • Supports LDAP • Possible to join and interact with Active Directory domains • Supports cloud (macincloud) but discontinued by Apple	• Possible • The most common • No licensing cost • Low hardware cost • Build time is short • More documentation • Supports LDAP • Possible to integrate with Active Directory using tools like Samba • Supports cloud (AWS, Digital Ocean etc.)	• Possible • Not so common • High Licensing cost • Low hardware cost • Build time is long • Enough documentation • Supports LDAP • Natively supports Active Directory • Supports cloud (Azure)	• Not feasible • Not preferred • Low licensing cost • Low hardware cost • Build time is long • Not enough documentation • Supports LDAP • Possible to integrate with Active Directory • Supports cloud to sync data
Client Side	• There is a native way. • Cross-platform possible	• There are various ways. • Cross-platform possible	• There are various ways. • Cross-platform possible	• There are various native ways. • Cross-platform possible

Development Tools	• Python, JavaScript, Java, Go, Rust, Docker for server • XCode, VScode, etc • Swift, Objective-C for native • Tauri or Electron, JavaScript, HTML, CSS for cross-platform	• Python, JavaScript, Java, Go, Rust, Docker for server • Any IDE • No native way • Tauri or Electron, JavaScript, HTML, CSS for cross-platform	• Python, JavaScript, Java, Go, Rust, Docker for server • Any IDE • No native way • Tauri or Electron, JavaScript, HTML, CSS for cross-platform.	• Python, JavaScript, Java, Go, Rust and Termux for server • Any IDE • Swift, Kotlin etc for native • Xcode, Android Studio, Vscode etc. • React Native, Capacitor, NativeScript, Ionic, JavaScript, HTML, CSS for cross-platform.

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**: Linux is the ideal operating system for expanding ”Draw it or Lose it”. Linux offers various distributions to choose from, and server costs could be as low as one dollar initially. It’s flexible and easy to customize and it can run for years without failing.

2. **Operating Systems Architectures**: A multi-tier architecture is ideal for the client's system. It allows for better scalability as the user base and system complexity grow, while also promoting modularity, making it easier to manage and update individual layers. Enhanced security measures can be implemented at different levels, ensuring the protection of both game logic and player data. System maintainability is improved, with bug fixes and updates applied to specific layers without disrupting the entire system. Additionally, performance is optimized by distributing processing load to dedicated application servers, and the flexibility to use different technologies for each layer allows for tailored solutions based on specific requirements.

3. **Storage Management**: A SQL database like Postgres is ideal to store data like scores and previous games since most of the objects have relationships with each other. It would be good practice to keep frequently accessed data on SSDs like application files. A cloud-based object storage service like Amazon S3 is a great way to store images and offload the storage burden from the application servers.

4. **Memory Management**: We use the singleton pattern to avoid multiple instances of an object in memory, and modern programming languages like Python and Java have garbage collectors that constantly free memory as much as possible. We can deploy our application servers on cloud-based virtual machines, so that we can reconfigure them to scale up the memory. We can utilize in-memory caching solutions like Redis, Memcached to handle frequently accessed data such as

leaderboards or game session statuses. Later, we can use distributed memory caching to share cached data between nodes.

5. **Distributed Systems and Networks**: We need to use technologies such as RESTful APIs, message queues, or web sockets for inter-component communication. We need RESTful APIs for handling user requests. We can send game-related data in-game with low latency and continuously using web sockets. We can utilize a service like sentry.io to detect errors and server crashes early. We can use load balancer to distribute incoming request, so that our servers can handle large number of users. We need to make sure that if one server or component fails, the system should automatically switch to a backup or alternate server to minimize the downtime and outages. We need strategies to maintain connectivity with retry mechanisms, degrading and offline modes.

6. **Security**: We should use HTTPS to protect data in transit. We need to maintain logs of user activities and system events to monitor and detect suspicious activities. We can use Sentry for that as well. A cloud service like AWS or Azure is helpful for security, utilizing virtual private servers. Instead of building the system to handle authentication and sessions, we should go with AWS Cognito or Oauth. They can save us time, and we won't have to deal with user security extensively. Finally, we will allow users to access only the necessary data.