The Gaming Room

CS 230 Project Software Design

Version 3.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	03-23-2025	Melissa Chessa	Initial Creation of Document
2.0	04-06-2025	Melissa Chessa	Expanded Evaluation Section for Project Two
3.0	04-20-2025	Melissa Chessa	Final version for Project Three

Executive Summary

- Extend the game to function on multiple operating systems and platforms.
- Ensure the system is scalable and maintainable for long-term use.
- Support high availability and performance for multiple simultaneous users.

Requirements

- Develop a distributed web-based system.
- Implement security and data protection features.
- Maintain consistent data across all components.
- Support client-server communication across networks.
- Support one instance of GameService in memory using the Singleton pattern.
- Implement unique identifiers for each game, team, and player.
- Use the iterator pattern to enforce uniqueness of team and game names.
- Create a distributed web application that can function across Windows, Linux, Mac, and mobile devices.
- Apply proper software security practices to protect user data.

Design Constraints

- The software must run in a web-based distributed environment.
- All logic must support multi-platform compatibility.
- A single instance of the GameService class must exist at runtime.

- Game and team names must be unique and searchable before creation.
- Software must be modular, scalable, and easy to deploy.

Implications:

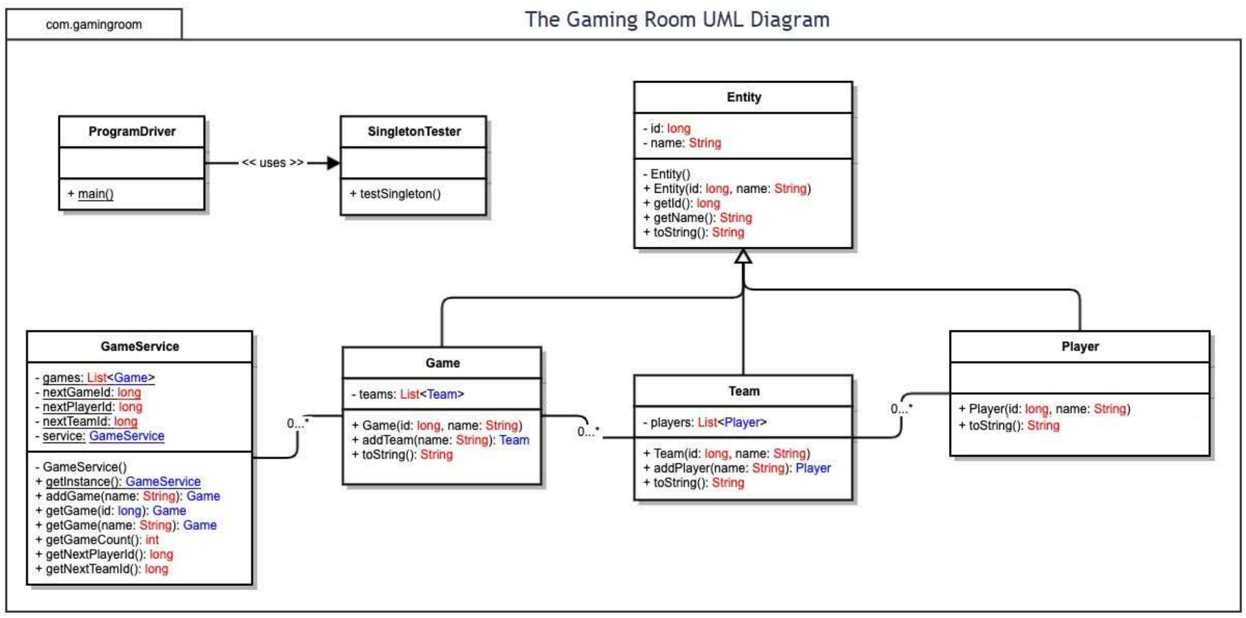
- Requires use of Java (for back-end), along with HTML/CSS and JavaScript for front-end.
- Frameworks such as React (front-end) and Spring Boot (back-end) are recommended.
- GameService must be implemented using the Singleton design pattern.
- The iterator pattern will be necessary for functions such as addGame() and addTeam().
- Security measures must be integrated to protect communication and data persistence.

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

<Describe the UML class diagram provided below. Explain how the classes relate to each other. Identify any object-oriented programming principles that are demonstrated in the diagram and how they are used to fulfill the software requirements efficiently.>



Evaluation

The UML class diagram defines the relationships and responsibilities between key classes:

- Entity: A base class with id and name shared by Game, Team, and Player.
- GameService: Implements the Singleton pattern to manage game creation and access. Maintains IDs and a list of games.
- Game: Composed of multiple Teams; supports adding teams.
- Team: Composed of multiple Players; supports adding players.
- Player: Individual players belong to a Team.
- ProgramDriver: Contains the main() method to initiate the application.
- SingletonTester: Verifies the Singleton pattern in GameService.

Object-Oriented Principles Demonstrated:

- Inheritance: Game, Team, and Player inherit from Entity, promoting code reuse.
- Encapsulation: Fields are private; access is controlled via getters/setters.
- Abstraction: The diagram abstracts the core entities of the game system.
- Singleton Pattern: GameService ensures that only one instance manages the game state.

Composition: Game contains Teams; Teams contain Players. These show strong HAS-A relationships.

Development Requirements	**Mac**	**Linux**	**Windows**	**Mobile Devices**
Server Side	Mac offers a Unix-based environment but is less commonly used in production. Less ideal due to limited hosting support and higher hardware cost.	Preferred for server-side development due to its stability, flexibility, open-source nature, and community support. Ideal for web server hosting due to scalability and low cost. Easily integrated with Java, Spring Boot, MySQL/PostgreSQL. No licensing costs.	Supports server-side deployment through IIS, Apache, or Tomcat. Licensing fees apply for Windows Server. Less stable than Linux for high traffic.	Not ideal for hosting. Limited capacity and dependency on cloud synchronization. Useful only for client-side interaction.
Client Side	Developing for Mac requires testing on Apple hardware and complying with specific UI guidelines. Cost and expertise required are high.	Linux clients are less common among end-users. Useful when targeting developers or power users. Compatible with all browsers.	Windows dominates desktop environments. Requires broad driver and browser support testing.	Mobile clients must support both Android and iOS, which adds cost and complexity. Cross-platform frameworks (e.g., React Native) help reduce effort.
Development Tools	Xcode (macOS/iOS development), IntelliJ. Git, Terminal, and Homebrew supported. Xcode is free, but hardware is expensive.	Eclipse, IntelliJ, VS Code. Git, Docker, SSH supported natively. Full control of CLI tools. No licensing costs for open-source tools.	Visual Studio, IntelliJ IDEA, NetBeans. Easy for Windows developers. Some tools may have licensing costs.	Android Studio, Xcode, React Native, Flutter. Need emulators and physical devices for full QA testing.

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 recommended for server-side deployment due to its stability, security, cost-effectiveness, and wide adoption in cloud environments.

2. **Operating Systems Architectures**: A multi-tier architecture should be implemented. The front-end will operate through a browser interface. The back-end will run on a Linux server using Java for logic and MySQL or similar for storage.

3. **Storage Management**: A relational database like MySQL or PostgreSQL is suitable for storing structured game data such as player IDs, game sessions, and team memberships. These databases are well-supported and integrate easily with Java applications.

4. **Memory Management**: The JVM (Java Virtual Machine) used on the Linux server provides robust memory management features such as garbage collection, stack/heap separation, and performance tuning capabilities that help maintain game responsiveness and stability.

5. **Distributed Systems and Networks**: The system should use RESTful APIs for communication between clients and server. Load balancers and redundancy must be used to manage high traffic and ensure uptime. Server nodes must sync state consistently, possibly using caching layers or real-time databases.

6. **Security**: Secure communication should be enforced via HTTPS. Passwords and sensitive data must be encrypted at rest and in transit. Use role-based access control (RBAC) and input validation to prevent injection attacks and unauthorized access.