Draw It Or Lose It

CS 230 Project Software Design - Michael Mathews

Version 3.0

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 5

Document Revision History

Version	Date	Author	Comments
3.0	02/19/2025	Michael Mathews	Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room.
2.0	02/06/2025	Michael Mathews	Evaluate platform characteristics, determine software development considerations, identify relevant tools.
1.0	01/25/2025	Michael Mathews	Information for setting up the software environment to facilitate the development of a web-based version of the (currently) Android only gaming app.

Executive Summary
CTS was tasked by The Gaming Room to create a web based version of their Android game: *Draw It or Lose It* with the goal of supporting multiple teams with multiple players per team. To ensure uniqueness, a singleton pattern is used for game instances, and an iterator pattern prevents conflicts among teams and players.

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.

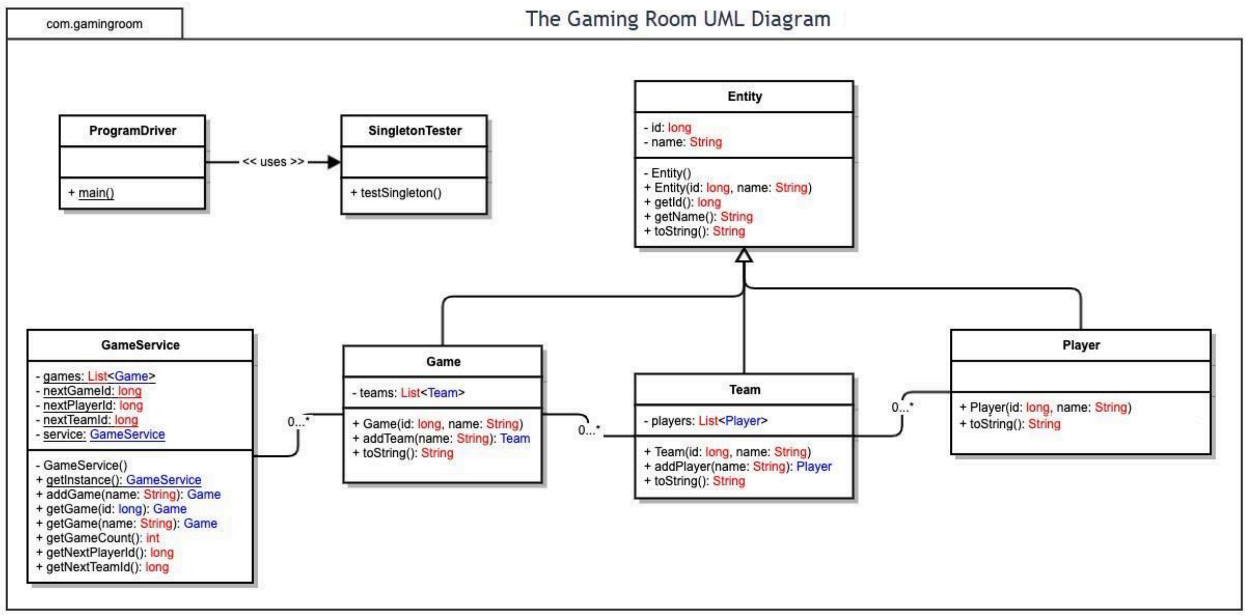
Design Constraints
The Gaming Room currently has an Android version of *Draw It or Lose It* and has asked us at CTS to make a web version. In order to do this the technology used needs to work well for the web, so Java was chosen. Since Java is the main language for Android it should make this project easier. Any APIs currently used for the Android app will need to be checked or updated for use with the web version.

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 application has a main driver class that starts the creation of all games, teams, and players. Game creation is handled by the GameService class, which uses the singleton to ensure only one instance exists. The GameService constructor is private and instances are created using the getInstance() method, which checks if it's already running.

Once GameService is running the driver class can call addGame() to create a new game. The method uses the iterator to prevent duplicate game names and adds the new game to the games list. The addTeam() and addPlayer() methods make sure no duplicate names are added for teams or players using the same pattern.

Game, Team, and Player inherit from Entity which has protected attributes for id and name and blocks null objects by requiring overloaded constructors. The design uses object-oriented techniques like polymorphism, inheritance, encapsulation, and abstraction. For example, users can add teams without needing to know how the team is created, as the constructor is private and accessed indirectly through addTeam().


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	Optimized for Apple hardware, but not commonly used as a server OS. User-friendly interface but lacks extensive server administration tools. Requires Apple hardware, making it expensive.	Highly efficient and lightweight, with minimal resource consumption.High security due to open-source transparency and strong user permissions. Highly customizable with different distributions.Free and open-source (except for some enterprise versions).	Requires more resources due to GUI and background services. Regularly updated security patches but often targeted by malware. Some customization through roles and features but limited compared to Linux. GUI-based, making it easier for administrators.Requires licensing, making it costly for businesses.	Not a good solution for maintaining a long term stable server.

Client Side	Requires Apple hardware (MacBooks, iMacs) for development, which is expensive. Xcode is free but macOS development can require paid developer accounts ($99/year for App Store distribution).- Streamlined development but limited to Apple's ecosystem. Requires knowledge of Swift or Objective-C.	Free and open-source tools GCC, Clang, and more.. Development can be done on low-cost hardware.Development time can be longer due to command-line-heavy environments and dependency management. Requires expertise in C, C++, Python, or Java. Development often requires package management and shell scripting knowledge.	Development tools like Visual Studio have free versions but enterprise editions are more expensive. Must pay licensing for Windows. -Some frameworks have licensing costs for enterprise features. Faster development with Visual Studio which has great debugging and UI design tools. .NET ecosystem provides many libraries for rapid development. Requires knowledge of C#, .NET, or C++.	iOS: Requires Mac hardware and a $99/year Apple Developer account. Android: Development is free with Android Studio. Cross-platform tools (Flutter, React Native) may require additional costs. iOS: Xcode's development and testing tools are optimized for fast iteration but App Store approval can take time. Android: Open development process, but different device specs increase testing time. Cross-platform (Flutter, React Native) speeds up development but may have performance trade-offs. iOS: Requires Swift/Objective-C expertise. Android: Requires Kotlin/Java expertise and Android SDK knowledge. Cross-platform: Requires Dart, JavaScript (React Native), or C# (Xamarin) expertise.

Development Tools	Swift - Primary language for macOS applications. Objective-C for Legacy macOS development. C++ for cross-platform applications. Python for scripting and automation. Xcode which is Apple's official IDE. JetBrains AppCode as an alternative to Xcode. Cocoa Framework for native macOS development. Homebrew as a Package manager. CMake a cross-platform build system. MacPorts for open-source macOS development.	C/C++ for core Linux applications. Python for scripting, automation, web services. Java for enterprise and server applications. Shell scripting bash for automation. Eclipse For Java development. JetBrains CLion for C++ development. Qt Creator for GUI-based Linux apps. GTK, Qt for GUI frameworks. Make, CMake, Autotools for build systems.APT (Debian/Ubuntu), RPM (RHEL/Fedora) all package managers. Snap, Flatpak for distributing Linux applications.	C# for .NET applications.C++ for Native Windows applications, games. Visual Basic for legacy business applications. Python for scripting and automation. Visual Studio is the primary IDE for Windows. JetBrainsRider for .NET development. Code::Blocks, JetBrains, Eclipse for C++ development. Eclipse, JetBrains for Java development. NuGet as a package manager for .NET. MSBuild for build automation.	For cross-Platform Mobile Development: Flutter - Dart. React Native -JavaScript/TypeScript. Xamarin - .NET/C#. Visual Studio for Xamarin. VS Code for Flutter and React Native. Expo for React Native development.

Recommendations

1. **Operating Platform**: A Linux-based server with a Windows or Linux-based front end would be the best option for cross platform development and distribution. This provides flexibility, scalability, and compatibility with web-based and native applications.

2. **Operating Systems Architectures**: Linux servers use a monolithic kernel architecture that provides high efficiency and direct hardware access for better performance. Windows front end systems operate on a hybrid kernel which can give a balance between stability and ease of use for client side interactions.

3. **Storage Management**: A cloud based storage system that uses MySQL, PostgreSQL, or MongoDB would provide a scalable and reliable data management solution. Which allows the game to efficiently store all of the game states, user profiles, and content across multiple platforms.

4. **Memory Management**: Linux uses demand paging and virtual memory techniques to allocate resources in an efficient way for server side processes. Windows uses a hybrid paging system with dynamic memory allocation which helps ensure smooth performance for game rendering and player input.

5. **Distributed Systems and Networks**: Use RESTful APIs and WebSockets to ensure real time communication between clients and servers, even across multiple platforms. With load balancing and cloud based redundancy to help maintain connectivity and minimize the impact of outages during the gameplay.

6. **Security**: TLS encryption, OAuth authentication, and secure API gateways will protect user data during any transmission. On the server side, firewall rules, intrusion detection systems, and database encryption will safeguard sensitive information from data breaches.