Draw It or Lose It
CS 230 Project Software Design Template
Version 1.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	08/10/2024	Ifeoluwa Adewoyin	Initial draft of software design document

Executive Summary

The Gaming Room, a game development company, aims to expand their Android-based game "Draw It or Lose It" into a web-based, multi-platform application. This software design document outlines our proposed solution to meet their requirements and facilitate the development process.

"Draw It or Lose It" is a team-based game inspired by the classic TV show "Win, Lose or Draw." The game renders stock images as clues, with teams competing to guess the puzzle within time constraints. To successfully transition this game to a web-based environment, we propose implementing a distributed system architecture that ensures scalability, maintainability, and cross-platform compatibility.

Our solution will leverage cloud-based services for backend operations, a responsive web design for the client-side interface, and robust data management systems to handle game states, user information, and image libraries efficiently. By adopting industry-standard design patterns and best practices in software development, we aim to create a flexible and extensible system that can accommodate future growth and feature additions.

Requirements

- Support for multiple teams in a single game instance.
- Multiple players can be assigned to each team.
- Unique identifiers for games, teams, and players.
- Ensure game and team names are unique.
- Only one instance of the game can exist in memory at any time.
- Cross-platform compatibility (web-based, supporting various devices).
- Render images from a large library of stock drawings as clues.
- Support four rounds of play, each lasting one minute.
- Implement a 15-second guessing window for non-drawing teams.

Design Constraints

Developing "Draw It or Lose It" as a web-based, distributed application presents several design constraints:

1. Cross-Browser Compatibility: The application must function consistently across various web browsers, which may have different rendering engines and JavaScript implementations. This requires thorough testing and potentially the use of polyfills or transpilation for newer JavaScript features.

2. Network Latency: As a real-time, multi-player game, network latency can significantly impact user experience. The design must incorporate efficient data transfer methods and potentially implement predictive algorithms to minimize the perceived lag.

3. Scalability: The system must be designed to handle a variable number of concurrent users and games. This constraint necessitates a scalable architecture, possibly involving load balancing and distributed computing techniques.
4. Security: Protecting user data and preventing cheating are crucial. This requires implementing robust authentication, authorization, and data encryption mechanisms.

5. Stateless Architecture: To ensure scalability and reliability in a web environment, the application should follow a stateless architecture as much as possible, with game state managed carefully and efficiently.

6. Mobile Responsiveness: The web application must be responsive to various screen sizes and touch interfaces, which impacts the UI/UX design and implementation.

7. Data Persistence: Efficient storage and retrieval of game data, user information, and the image library is crucial. This may require a combination of database technologies and caching mechanisms.

8. Real-time Updates: The game requires real-time updates for all players, which may necessitate the use of WebSockets or similar technologies for push notifications.

System Architecture View

While not explicitly required for this project, it's worth noting that a comprehensive system architecture for "Draw It or Lose It" would likely involve:

1. A multi-tiered architecture with separate layers for presentation, application logic, and data management.
2. A cloud-based backend service to handle game logic, user management, and data persistence.
3. A responsive web frontend, possibly built with a modern JavaScript framework.
4. A WebSocket server for real-time communication between clients and the server.
5. A content delivery network (CDN) for efficient distribution of static assets like images.
6. Load balancers to distribute traffic and ensure high availability.
7. Caching layers to improve performance and reduce database load.

Domain Model

The UML class diagram provided illustrates the core classes of the "Draw It or Lose It" game application. Here's a description of the classes and their relationships:

1. Entity: This is the base class for Game, Team, and Player. It contains common attributes (id and name) and methods (getId(), getName(), toString()).
2. GameService: This is a singleton class that manages the creation and retrieval of Game instances. It uses the List<Game> to store games and provides methods to add and retrieve games.
3. Game: Inherits from Entity and contains a List<Team>. It provides methods to add teams and convert game information to a string.

4. Team: Inherits from Entity and contains a List<Player>. It provides methods to add players and convert team information to a string.
5. Player: Inherits from Entity and provides a method to convert player information to a string.

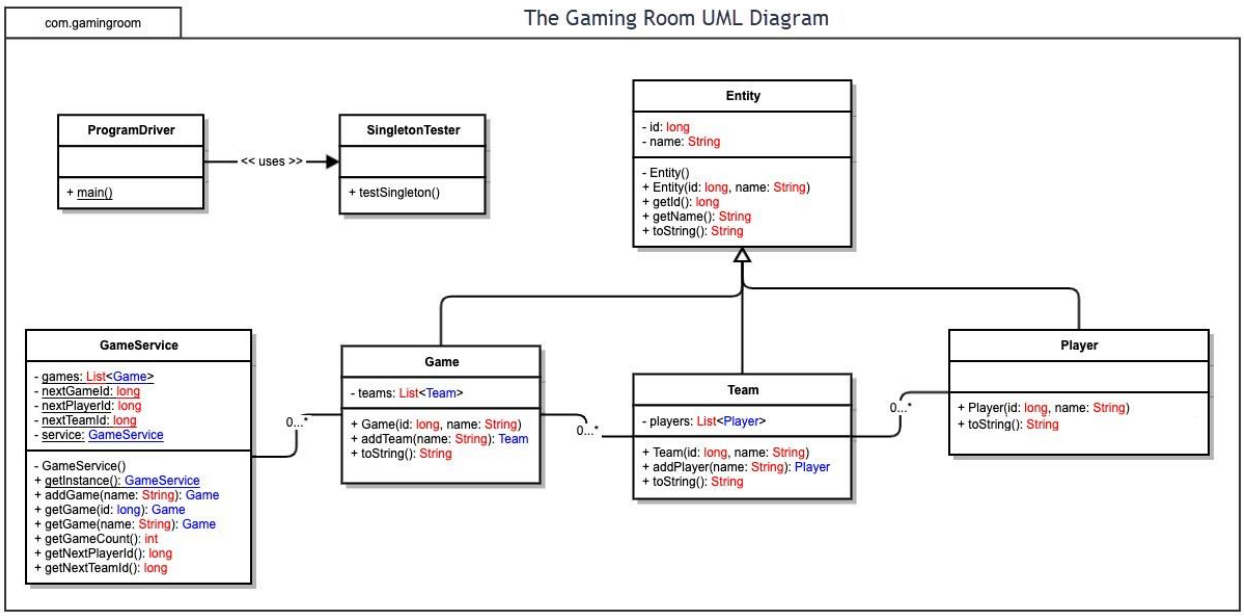
The relationships between these classes are as follows:

- GameService has a one-to-many relationship with Game (composition).
- Game has a one-to-many relationship with Team (composition).
- Team has a one-to-many relationship with Player (composition).
- Game, Team, and Player all inherit from Entity (generalization).

This design demonstrates several object-oriented principles:

1. Inheritance: Game, Team, and Player inherit from Entity, promoting code reuse.
2. Encapsulation: Each class encapsulates its data and provides methods to interact with it.
3. Composition: GameService, Game, and Team use composition to manage their respective collections.
4. Singleton: GameService is implemented as a singleton to ensure only one instance exists.

These principles help fulfill the software requirements efficiently by providing a clear structure for managing games, teams, and players, while ensuring unique identifiers and names as required.



Evaluation

Development Requirements	Mac	Linux	Windows	Mobile Devices

Server Side	Mac servers can run popular web servers like Apache or Nginx, and support various backend technologies. They offer good performance and stability but may have higher hardware costs. The Unix-based system provides robust security features.	Linux is a popular choice for servers due to its stability, security, and low cost. It supports a wide range of web technologies and can be highly customized. Its open-source nature allows for extensive community support and frequent updates.	Windows servers offer seamless integration with other Microsoft technologies. They provide a user-friendly interface for management but may have higher licensing costs. Windows servers are widely used and have good support for various web technologies.	Mobile devices are typically not used for hosting web applications due to power and performance limitations. They are more suited as clients for accessing the web application.
Client Side	Mac provides a consistent environment for testing and development. Safari, the default browser, may require specific considerations. Development for Mac ensures compatibility with a significant user base.	Linux offers a variety of desktop environments and browsers for testing. It's less common as a client OS for average users but important for ensuring compatibility with open-source browsers.	Windows is the most widely used desktop OS, making it crucial for client-side compatibility. It supports all major browsers and development tools, ensuring broad reach.	Mobile devices require specific considerations for touch interfaces, varied screen sizes, and potential bandwidth limitations. Testing on both iOS and Android devices is crucial for ensuring broad mobile compatibility.

Development Tools	Xcode (for native app development), Visual Studio Code, JetBrains IDEs (IntelliJ IDEA, WebStorm), and all major web browsers are available. Mac supports most programming languages relevant to web development.	Popular IDEs like Visual Studio Code, JetBrains IDEs, and Eclipse are available. Linux supports all major programming languages and provides powerful command-line tools for development.	Visual Studio, Visual Studio Code, JetBrains IDEs, and other major development tools are available. Windows supports all major programming languages and frameworks used in web development.	Xcode (for iOS) and Android Studio are the primary IDEs for mobile app development. For web applications, mobile devices are typically not used for development but are crucial for testing.

Recommendations

1. Operating Platform: We recommend a cloud-based platform like Amazon Web Services (AWS) or Microsoft Azure for hosting the "Draw It or Lose It" web application. These platforms offer scalability, reliability, and services that cater to various aspects of web application hosting and management.
2. Operating Systems Architecture: For the server-side, we recommend using Linux-based systems due to their stability, security, and cost-effectiveness. The architecture should be containerized using technologies like Docker to ensure consistency across development and production environments.
3. Storage Management: We recommend using a combination of relational and NoSQL databases. A relational database like PostgreSQL can handle structured data (user accounts, game records), while a NoSQL database like MongoDB can manage the more flexible data structures (game states, image metadata). For caching and session management, Redis would be an excellent choice.
4. Memory Management: The recommended cloud platforms provide robust memory management features. Implementing proper garbage collection in the backend services (e.g., using Java's or Node.js' built-in GC) will be crucial. For the "Draw It or Lose It" game, efficient management of game state in memory will be vital for performance.
5. Distributed Systems and Networks: To enable cross-platform communication, we recommend implementing a RESTful API for general data operations and WebSocket connections for real-time game updates. A microservices architecture could be beneficial, separating concerns like user management, game logic, and image serving. Load balancers should be used to distribute traffic and ensure high availability.
6. Security: To protect user information:
   Implement HTTPS for all communications.
   Use OAuth 2.0 for authentication and JWT for session management.
   Encrypt sensitive data at rest and in transit.
   Implement rate limiting and DDoS protection at the network level.
   Regularly update and patch all systems and dependencies.
   Conduct regular security audits and penetration testing.

References

1. Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley Professional. [Design patterns and enterprise architecture]
2. Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional. [Domain modeling and software design]
3. Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media. [Microservices architecture]
4. Amazon Web Services. (2024). AWS Architecture Center. https://aws.amazon.com/architecture/ [Cloud architecture patterns]
5. Microsoft Azure. (2024). Azure Architecture Center. https://docs.microsoft.com/en-us/azure/architecture/ [Cloud architecture patterns]
6. Mozilla Developer Network. (2024). Cross-browser compatibility. https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing [Web development best practices]
7. OWASP. (2024). OWASP Top Ten. https://owasp.org/www-project-top-ten/ [Web application security]
8. Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine. [RESTful architecture]
9. Fette, I., & Melnikov, A. (2011). The WebSocket Protocol. IETF. https://tools.ietf.org/html/rfc6455 [Real-time web communication]
10. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. [Object-oriented design patterns]
11. Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall. [Software architecture principles]
12. Google. (2024). Material Design. https://material.io/design [UI/UX design principles]
13. Richardson, L., & Ruby, S. (2007). RESTful Web Services. O'Reilly Media. [Web API design]
14. Nygard, M. T. (2007). Release It!: Design and Deploy Production-Ready Software. Pragmatic Bookshelf. [Scalability and reliability patterns]
15. Hunt, A., & Thomas, D. (1999). The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley Professional. [Software development best practices]