Draw It Online

CS 230 Project Software Design Template

Version 3.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
3.0	02/24/2024	Zainab Lowe	Updates in the “Recommendations” section following project 3 guidelines.

Executive Summary

This comprehensive software design document presents the roadmap for developing "Draw It Online," an innovative web-based adaptation of the existing Android application, "Draw It or Lose It." Our goal is to enhance and transform the current gaming experience into a versatile, cross-platform web application. The game's concept, drawing inspiration from the classic 1980s television game show "Win, Lose or Draw," revolves around teams engaging in a competitive environment to decipher puzzles based on a sequence of visual clues.

"Draw It Online" will retain the fundamental appeal of the original game while leveraging the advantages of web technology to introduce significant improvements. These enhancements include broadening accessibility across various devices and platforms, enriching user interaction, and providing a more dynamic and scalable gaming environment.

Key Functionalities and Innovations:

Multi-Team Dynamics: Unlike its predecessor, "Draw It Online" will support multiple teams in a single game, each comprising several players. This feature aims to encourage collaborative play and expand the game's social aspects.

Unique Identification System: To streamline user experience, the game will implement a real-time verification system for the uniqueness of game and team names. This mechanism is crucial to avoid confusion and conflicts among players and teams.

Single Instance Integrity: A vital aspect of the design is to ensure that only one instance of the game is operational in memory at any given moment. This will be achieved through the implementation of unique identifiers for each game, team, and player, thereby enhancing the game's performance and stability.

Web-Based Architecture: The transition to a web-based platform is pivotal. It not only broadens the game's reach but also offers seamless updates, maintenance, and cross-platform compatibility, ensuring a consistent and high-quality gaming experience.

This document will delve into the detailed software requirements, elucidating the technical specifications, and development strategies. Additionally, it will address the design constraints inherent in creating a distributed web-based application and their implications on the development process. The document also includes a review of the existing UML diagram, providing insights into the domain model and highlighting the object-oriented programming principles employed. These details are aimed at ensuring a robust, scalable, and engaging gaming experience that aligns with the client's vision and market expectations.

Requirements

The development of "Draw It Online" requires meticulous attention to both business and technical aspects. A clear and concise understanding of these requirements is vital to steer the project towards success while aligning with the client's vision and objectives.

Business Requirements:

Market Accessibility: The game must be accessible across various platforms including desktop, tablets, and smartphones to capture a wide user base.

Engagement and Retention: Strategies to enhance user engagement and retention, such as incorporating leaderboard systems, regular updates with new features, and community-building tools.

Scalability and Flexibility: The application should be scalable to support a growing user base and flexible to adapt to changing market trends and user preferences.
Revenue Generation: Integration of monetization mechanisms such as in-app purchases, subscriptions, or targeted advertising while ensuring a balance between revenue generation and user experience.
Brand Integrity: Ensuring that the transition from the original Android app to a web-based platform maintains the game's core essence and aligns with the existing brand identity.

Technical Requirements:
Multi-Team and Multi-Player Support: The game must support multiple teams with multiple players in each, ensuring smooth and synchronized gameplay.
Unique Identity Verification: Implementation of a robust system to ensure unique names for games and teams, avoiding conflicts and enhancing user experience.
Memory Management: Designing the application to allow only one active instance in memory at a time, efficiently managing resources through the use of unique identifiers.
Responsive and Adaptive Design: The game interface must be responsive to different devices and screen sizes, providing a consistent and optimal user experience.
High-Performance Backend: Developing a backend infrastructure that is capable of managing real-time data exchange, scalability, and ensuring game integrity and security.

Design Constraints

The development of "Draw It Online," a web-based gaming application, is subject to a range of design constraints that significantly influence its architecture and functionality. These constraints shape the technological choices, development methodologies, and ultimately the effectiveness and appeal of the final product.

Cross-Platform and Device Compatibility: The application must provide a consistent and seamless experience across various platforms and devices (e.g., PCs, smartphones, tablets). This necessitates a responsive and adaptive design, which implies a focus on HTML5, CSS3, and JavaScript frameworks that are known for their cross-compatibility. The trade-off might be in limiting the use of advanced platform-specific features that are not uniformly supported across all devices and operating systems.
Network Performance and Reliability: Given the game's reliance on internet connectivity, it must be optimized for varying network conditions. This includes efficient data handling to reduce latency and bandwidth usage, and robust error-handling mechanisms for network interruptions. This constraint impacts server choice and necessitates cloud-based solutions with global distribution capabilities to minimize latency.
Browser Compatibility and Standards Compliance: The application must function optimally across all major web browsers, requiring adherence to web standards and extensive cross-browser testing. This multi-browser compatibility often complicates the frontend development process and can limit the use of cutting-edge browser features that are not yet universally supported.
Scalability and Resource Management: The ability to scale and manage server resources efficiently in response to fluctuating user numbers is crucial. This constraint influences the backend architecture, leading to the adoption of scalable cloud infrastructure, microservices architecture, and efficient database management systems to handle dynamic loads.
Security and Data Protection: As a web-based application, it faces heightened security risks. Stringent security measures, including secure coding practices, data encryption, regular security audits, and compliance with data protection regulations, are imperative. This constraint mandates a multi-layered security approach encompassing both the application and the infrastructure.

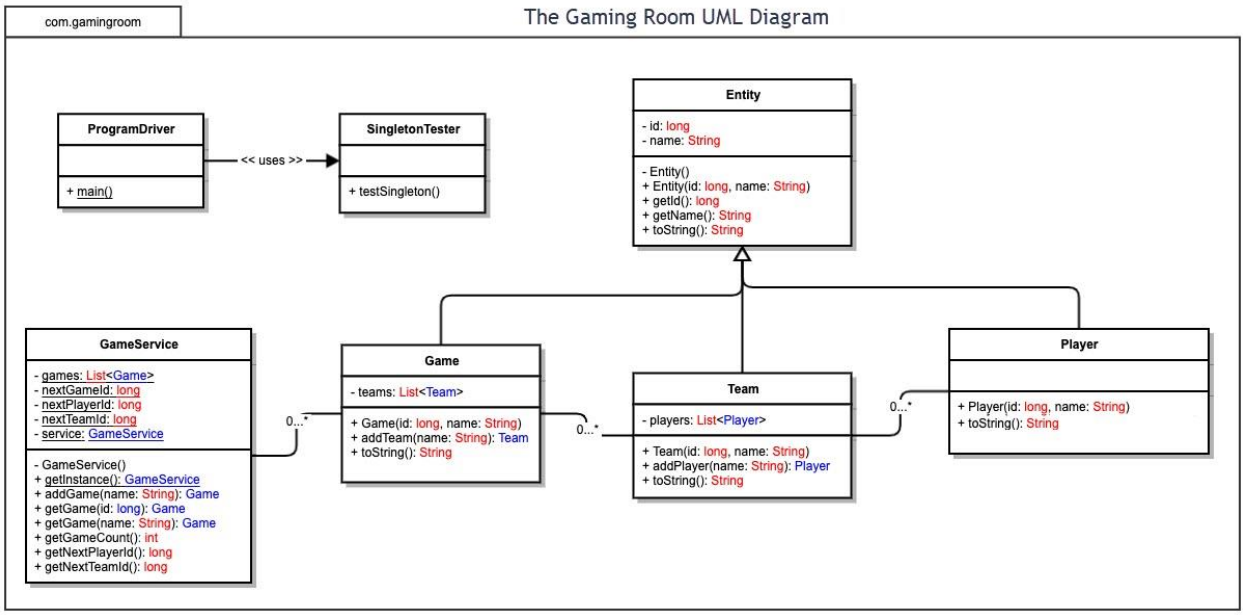
User Interface (UI) and User Experience (UX) Design: Creating an engaging and intuitive UI/UX that is consistent across different devices and screen sizes poses a significant challenge. This requires an innovative and adaptive design approach, potentially increasing the complexity and time involved in the UI/UX development process.
Integration of Monetization Strategies: Incorporating monetization features such as advertisements or in-app purchases without disrupting the gaming experience requires a delicate balance. Designing these elements to be non-intrusive yet effective is a complex task that necessitates creative solutions and can influence the overall game design.

System Architecture View

In the case of "Draw It Online," a thorough understanding of the system architecture is vital, though not a requirement for this project. A detailed view of this architecture would encompass:

Frontend Architecture: This layer includes the user interface elements and client-side logic. It is built using responsive design principles with technologies such as HTML5, CSS3, and JavaScript frameworks, ensuring compatibility across different devices and browsers.
Backend Architecture: The server-side component handles game logic, data processing, and network communications. It is likely to be implemented using scalable cloud services and might employ a microservices architecture for greater modularity and ease of maintenance.
Database Systems: A robust and scalable database is essential for storing user profiles, game data, and transaction records. The choice of database (SQL vs. NoSQL) will depend on the data structure and requirements for scalability and speed.
Network Infrastructure: This includes the necessary hardware and software for internet connectivity and internal networking. It involves the use of secure transmission protocols and efficient data handling strategies to ensure smooth communication between the client and server.
Cloud Computing Services: Utilizing cloud platforms for hosting the application offers scalability and reliability. Services like AWS or Azure could be used for their extensive network and capabilities in handling large-scale web applications.
Security Framework: Incorporating comprehensive security measures at every level of the architecture is essential. This includes secure coding practices, encryption, firewalls, and regular updates to safeguard against emerging threats.
Physical Architecture: Though primarily a web-based application, understanding the physical deployment in terms of server locations, CDN (Content Delivery Network) usage, and backup facilities is important for performance optimization and disaster recovery planning.

Domain Model



The provided UML (Unified Modeling Language) class diagram represents the structure of a system by showing the system's classes, their attributes, methods, and the relationships between the classes. Here's a breakdown of the diagram:

Entity Class: This is an abstract class indicated by the italicized class name. It has two attributes: id (of type long) and name (of type String). It also has a constructor that takes id and name, along with getters for both and a toString method. Other classes in the diagram inherit from this class, as shown by the lines with hollow arrowheads pointing to it. This suggests that Game, Team, and Player are all types of Entity and inherit its properties and methods.
Game Class: Inherits from Entity. It has a teams attribute which is a list of Team objects. The Game class has methods for adding a team and overriding the toString method. The "0..*" notation near the Team class indicates that a Game can have zero or more Team instances.
Team Class: Also inherits from Entity. It has a players attribute which is a list of Player objects. The class has methods to add a player to the team and an overridden toString method. The "0..*" notation near the Player class indicates that a Team can have zero or more Player instances.
Player Class: Inherits from Entity and appears to be a leaf in the class hierarchy, with no subclasses. It does not add any new attributes or methods beyond what it inherits from Entity, except for the overridden toString method.
GameService Class: This class does not inherit from Entity and seems to be a service or controller class. It contains attributes to keep track of games (games as a list of Game objects), the next game ID (nextGameId), the next player ID (nextPlayerId), and the next team ID (nextTeamId). It has a constructor and methods for game-related operations, such as adding a game, getting a game by name or ID, and getting counts of games, players, and teams. It also has methods to generate the next IDs for games, players, and teams, ensuring uniqueness. The "0..*" notation near the Game class indicates that the GameService can manage zero or more Game instances.

ProgramDriver Class: This seems to be the main entry point of the application, with a main method. It uses the GameService.
SingletonTester Class: It has a method testSingleton() which presumably tests the singleton nature of the GameService. This implies that GameService is likely implemented as a singleton, ensuring that there is only one instance of this service throughout the application.

Object-Oriented Principles Demonstrated:

Inheritance: Game, Team, and Player classes inherit from the Entity class, promoting code reuse and polymorphism.
Encapsulation: Each class encapsulates its data and operations, exposing only what is necessary through methods.
Abstraction: The Entity class provides an abstraction for the shared attributes and methods of Game, Team, and Player.
Polymorphism: This is inferred by the overridden toString methods in the subclasses of Entity, allowing them to provide specific implementations.
Singleton Pattern: It's suggested that GameService is a singleton, which ensures that there is only one instance of this object in the application, providing a global point of access to it.
These principles are used to create a well-structured and efficient software design that facilitates maintenance and scalability. The use of inheritance allows shared attributes and methods to be defined once in the Entity class, rather than in each class separately, reducing redundancy. Encapsulation and abstraction make the system more modular and easier to manage, as changes in one part of the system are less likely to impact other parts. The singleton pattern ensures that the state and behavior related to game management are centralized in a single instance, preventing conflicts and duplication.

Evaluation

Development Requirements	Mac	Linux	Windows	Mobile Devices
Server Side	**Deployment Method:** Mac OS offers server capabilities, suitable for web-based applications but less common due to niche market presence. **Licensing Costs:** Higher, reflecting Apple's premium strategy. **Characteristics:** Offers robust security and stability, but faces limitations in software diversity and support community.	**Deployment Method:** Linux is highly preferred for its scalability, flexibility, and open-source nature, ideal for large-scale web applications. **Licensing Costs:** Mostly free, with no licensing fees for the majority of distributions. **Characteristics:** Unparalleled stability, security, and customization options with extensive community support.	**Deployment Method:** Windows Server is user-friendly and integrates well with enterprise applications, suitable for web-based applications. **Licensing Costs:** Higher due to significant licensing fees. **Characteristics:** Less secure than Linux, requiring additional security measures.	**Deployment Method:** Not applicable for server-side deployment due to limited processing power and storage. **Licensing Costs:** N/A. **Characteristics:** Focused on client-side use.

Client Side	**Compatibility:** Development with Apple-specific tools needed for responsive HTML design. **Development Considerations:** Higher costs for hardware and software, justified by targeting specific audience preferences.	**Compatibility:** Requires accommodating diverse Linux distributions, increasing testing efforts. **Development Considerations:** Cost-effective due to open-source nature but demands expertise for varied environments.	**Compatibility:** Ensuring compatibility across different Windows versions is crucial for wide market reach. **Development Considerations:** Availability of diverse tools makes development cost-effective, though compatibility testing is extensive.	**Compatibility:** Development for both iOS and Android requires responsive design and extensive device testing. **Development Considerations:** Involves expertise in mobile frameworks and languages; the fragmented ecosystem increases complexity.
Development Tools	**Tools:** Xcode for native, Visual Studio Code for web. **Impact on Team:** Necessitates expertise in Apple's ecosystem. **Licensing Costs:** Includes costs for some development software and Apple Developer Program membership.	**Tools:** Wide range including GCC for C/C++, Python, Java, and IDEs like Eclipse. **Impact on Team:** Broad tool support minimizes need for multiple specialized teams. **Licensing Costs:** Mostly free, benefiting from the open-source ecosystem.	**Tools:** Microsoft Visual Studio for .NET and other languages. **Impact on Team:** May require separate teams for Microsoft technologies and web development. **Licensing Costs:** Licensing fees for the OS and Visual Studio, though free versions exist for smaller teams.	**Tools:** Android Studio and Xcode for native development, with cross-platform frameworks like React Native. **Impact on Team:** Might necessitate multiple teams for iOS, Android, and cross-platform development. **Licensing Costs:** Mainly associated with iOS development, including Apple Developer Program fees.

This detailed evaluation addresses the client's questions by examining each platform's capabilities for server-side hosting, client-side support, and the necessary development tools. For server-side, Linux emerges as the most cost-effective and flexible option, while Mac and Windows present higher licensing costs and specific use cases. Client-side development requires a focus on compatibility and responsive design, with mobile development emphasizing the need for expertise across different devices and operating systems. The choice of development tools impacts the team structure and budget, considering the potential need for multiple teams and the licensing costs associated with certain platforms and tools.

Recommendations for "Draw It or Lose It" Expansion

1. Operating Platform

Recommendation Overview: To accommodate "Draw It or Lose It" in a multi-platform environment, a strategic selection of operating platforms is paramount. We recommend a dual-approach architecture: Linux for server-side operations due to its robust performance and security features, coupled with a universally accessible web-based client platform. This configuration ensures wide accessibility while minimizing development and operational costs.

Server-Side Platform: Linux is advocated for the server environment, leveraging its stability, extensive community support, and open-source cost-efficiency. Linux's compatibility with various web technologies makes it an ideal candidate for backend operations, providing a solid foundation for the game's server requirements.

Client-Side Platform: A web-based client platform is recommended for cross-platform compatibility. Utilizing modern web technologies (HTML5, CSS3, JavaScript) alongside frameworks such as React.js or Angular ensures a responsive, device-agnostic user experience. This approach eliminates the need for multiple native applications, streamlining development and maintenance.

2. Operating Systems Architectures

Linux Architecture: We propose adopting a LAMP (Linux, Apache, MySQL, PHP) or MEAN (MongoDB, Express.js, Angular, Node.js) stack, depending on the game's specific backend requirements. These stacks offer a scalable, efficient solution for web application hosting, with a proven track record in managing dynamic content and real-time interactions.

Client Architecture: The client-side architecture will be built on responsive web design principles, ensuring seamless operation across diverse devices and screen sizes. The use of single-page application (SPA) architecture will enhance user engagement through fast, dynamic updates without reloading the entire page.

3. Storage Management

Cloud-Based Storage Solution: For optimal scalability and reliability, cloud-based storage solutions such as Amazon S3 for object storage and Amazon RDS or Google Cloud SQL for database services are recommended. These platforms offer automated scaling, backup, and recovery processes, ensuring the game's data remains secure and accessible.

Performance Optimization: Incorporating caching mechanisms like Redis or Memcached will significantly improve response times and reduce the load on the database by storing frequently accessed data in memory.

4. Memory Management

Linux Memory Management: The Linux operating system employs advanced memory management techniques, including virtual memory, demand paging, and swap management, to maximize efficiency. For "Draw It or Lose It," utilizing containerization technologies (e.g., Docker) will provide isolated environments for each instance of the game, allowing for precise resource allocation and scalability.

5. Distributed Systems and Networks

Architecture Design: A microservices architecture is recommended to facilitate the game's operation across various platforms. This architecture enhances modularity, allowing individual game components to be developed, deployed, and scaled independently.

Communication Protocol: RESTful APIs are recommended for facilitating communication between the frontend and backend, ensuring lightweight, stateless interactions. For real-time features, WebSocket technology will provide full-duplex communication channels over a single TCP connection, crucial for live game updates.

6. Security

Comprehensive Security Strategy: Security is a multi-layered endeavor, requiring attention at every level of application development and deployment. On the Linux server, implementing firewalls, intrusion detection systems, and regular patch management will fortify the game against unauthorized access and attacks.

Data Transmission Security: All client-server communication must be encrypted using TLS, with HTTP Strict Transport Security (HSTS) enforced to prevent intercepts. Additionally, employing modern authentication protocols such as OAuth 2.0, alongside JSON Web Tokens (JWT) for authorization, will secure user sessions and data access.

Ongoing Security Practices: Regular security audits, compliance with data protection regulations, and adherence to security best practices (OWASP guidelines) will ensure the game's infrastructure remains impervious to emerging threats.