

<Name-of-Software-Application>

**CS 230 Project Software Design Template**

Version 1.0

**Table of Contents**

**Document Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comments |
| 1.0 | <19/03/23> | Fernando Balen | Initial Revision |
| 2.0 | <03/04/23> | Fernando Balen | Evalution of software and platforms |
| 3.0 | <16.04.23> | Fernando Balen | Recommendation changes |

**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**

This document describes the software design for a game application that runs in a web-based distributed environment. The game application includes several classes such as ProgramDriver, SingletonTester, Entity, GameService, Game, Team, and Player. These classes are connected to each other, and the UML class diagram provides an understanding of how they relate to each other. The document also describes the requirements and constraints for developing the game application in a web-based distributed environment, including identifying the client's business and technical requirements. The proposed solution fulfills these requirements efficiently.

**Requirements**

*The client's requirements for the game application include the ability to support multiple types of clients such as Mac, Linux, Windows, and mobile devices. The application should be user-friendly, easy to use, and offer high-performance.*

**Design Constraints**

The design constraints for developing the game application in a web-based distributed environment include ensuring that the application is scalable and fault-tolerant. The application should also have a high level of security to protect user data and ensure that unauthorized users cannot access the system. Furthermore, the application should be designed in such a way that it can be easily maintained and updated.

**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 UML class diagram provided below consists of seven classes that are connected to each other. The ProgramDriver class contains a main method that uses the SingletonTester class to test the Singleton design pattern. The Entity class contains fields for storing the entity's ID and name and provides methods for retrieving the ID and name values. The GameService class provides methods for adding, retrieving, and counting games, as well as generating IDs for players and teams. The Game class contains a list of teams and methods for adding teams and generating game IDs. The Team class contains a list of players and methods for adding players and generating team IDs. The Player class contains fields for storing player ID and name and provides a toString method that returns the player's name.

The UML class diagram demonstrates several object-oriented programming principles, including encapsulation, inheritance, and polymorphism. For example, the Entity class uses encapsulation to hide the implementation details of its fields and methods. The GameService class uses inheritance by extending the Entity class, and the Team and Player classes use polymorphism to allow for the addition of new players and teams dynamically.



**Evaluation**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| **Server Side** | The high licensing cost of the platform can make it expensive for small businesses (Gleeson, 2021). However, the platform offers an intuitive graphical user interface (GUI) similar to that of Windows. Its architecture is a combination of layered and modular, allowing for easy scaling of functionality with minimal performance impact (Mell and Grance, 2011). The platform supports a wide range of implementation technologies, offering more flexibility and fewer technical constraints compared to Windows (Cachia, 2008). However, scalability can be expensive as the platform carries many features that may not be necessary for hosting specific applications, and licensing costs can increase with additional servers (Santos, 2020). | Open source with numerous libraries and utilities pre-packaged (Horne, 2019). Least privilege access enforced out of the box (Taylor, 2018) File structure is susceptible to trojan horses and access control is more easily circumvented (Silberschatz, 2009). Certain distributions carry a GUI, but administrative functions are best executed from the command line. Modular architecture exposes only core functionality through the kernel with scalability supported through modules (Silberschatz, 2009). Best OS processing power and stability (Battle of Web…, 2013). Open source application availability yields lowest of technical constraint for needed solutions and tools. | Expensive platform with both OS and product licensing costs (Horne, 2019). Targets Microsoft web framework technologies, imposing technical constraints (Heng, 2020). Intuitive GUI for administrative functions (Battle of Web…, 2013). Most restrictive user access controls and lock screen sequences that restrict malicious scope. Least privilege access tends to require additional configuration vs. Linux (Silberschatz, 2009). Process execution is more granular vs. other OSs and tends to get blocked less. Supports a robust service API that integrates easily with Java (Silberschatz, 2009). Layered architecture that is heavyweight and subject to performance and availability issues. Costly scalability. | Web hosting frameworks like I-Jetty emphasize scalability and performance" (Vangie Beal, 2022). "The process of DNS conversion needs connectivity to an internet-enabled computer, which adds extra relay steps" (Vangie Beal, 2022). "Some popular solutions are no longer actively maintained" (Vangie Beal, 2022). "Deployment tools like Jenkins and Maven are supported on all major OSs" (GeeksforGeeks, 2022). "Deployment solutions that are independent of the OS can be used" (GeeksforGeeks, 2022). |
| **Client Side** | Regardless of OS, the software development process should ensure optimized, modular approaches that are capable of handling numerous requests and delivering quick, lightweight responses to the client. RESTful API architecture supports these needs and promotes scalability (Roy Fielding, 2000). The software should also support modern browsers and use secure authentication and transmission protocols to prevent session hijacking (Kaur & Gupta, 2017). Mac OS supports a limited number of file structures, which may restrict the types of media that can be used without additional programming overhead (Bartlett, n.d.). | Due to the limitations in file structures, additional programming is needed to handle media types that are not supported by the out-of-the-box file type support (Sharma, 2021). This requires extra effort in terms of programming and can be a potential challenge for developers (Sharma, 2021). | Although optimized to run with Microsoft proprietary web browsers, the software supports all major web browsers and the Microsoft web browsers provide built-in support for the RESTful API architecture, which is ideal for delivering lightweight responses (Kaur & Gupta, 2017). | The software lacks processing capability and does not support plugin architecture to handle heavyweight or executable media types (such as Java applets). All the logic must be server-side, making it better suited for lightweight RESTful API with concise message exchange (such as JSON) (Monus, 2020). Developers must possess expertise in developing for optimal user experience across various form factors (Zmora, 2015). |
| **Development Tools** | "Multi-threaded one-to-one architecture is a fundamental design pattern used in many operating systems, including Windows (Singhal & Shrivastava, 2019). Language factors such as object-oriented programming (OOP) and use of compiled languages are important for scalability, concurrency, and rapid development on different OSs (Chen, 2018). Many open source frameworks like React Native and Xamarin, enable cross-platform development and support deployment to multiple OSs (Johnson, 2021). | The multi-threaded one-to-one model used in many operating systems can introduce extra overhead (Kilinc & Ileri, 2019). Interpreted languages like Ruby and PHP can limit scalability and design capabilities, while compiled languages such as C++ can provide better performance and scalability (Chen, 2018). Frameworks like Node.js, which use an event-driven model, are also designed to improve scalability and performance (Srinivasan, 2015). | The one-to-one thread model used in Windows and other operating systems can introduce performance overhead, especially in highly parallelized applications (Gupta, 2021). Compiled languages like C# can help offset this overhead and promote scalability (Chen, 2018). Windows is optimized for .NET framework, which uses C# for OOP, but also supports other programming languages like C++ and Visual Basic (Microsoft, n.d.). | Android favors Java as its primary programming language, while iOS primarily uses Swift and Objective-C (Kucherenko, 2018). Frameworks like React Native and Flutter, which use JavaScript and Dart, respectively, can be used to develop cross-platform mobile applications for both Android and iOS (Bhandari, 2020). However, cross-platform development requires knowledge and expertise in particular frameworks or polyglot programming, and can introduce technical challenges like device fragmentation (Lamanna, 2020). |

**Recommendations**

Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room. Specifically, address the following:

**Operating Platform**:   
The Gaming Room should consider a cloud-based architecture with a Linux OS distribution, as it can reduce complexity and inefficiencies while promoting scalability and efficiency (Smith, 2021). This architecture allows for accessing services through various methods, such as HTTP and APIs, enabling better support for distributed operations across different computing environments (Brown, 2020).

The pay-by-use serverless pricing model promotes efficient design with fewer dependencies and reduced processing time (Jones, 2022). Middle layer service solutions can simplify the app's logic layer, allowing for direct access to functionalities from the client and promoting a simpler app architecture (Smith, 2021). This can result in a faster-running application with improved user experience and quicker implementation of product improvements (Brown, 2020).

* **Operating Systems Architectures**

Traditional three-tier solutions, as described in the Three-Tier Architecture (2018), typically involve a middle layer that handles logic and communication between the user view and the data store. In contrast, serverless architectures take this concept further by modularizing the middle layer, allowing for the consumption of various services, such as security services, content delivery, and business logic functions (Smith, 2021). This modularization is similar to the microservice architecture, but on a larger scale, and provides The Gaming Room with the flexibility to only use and pay for the services it needs.

In a similar manner, the Linux OS is designed in a modular fashion, with a kernel that contains only core components and separate modules for dynamic expansion of services (Johnson, 2020). This results in a lightweight solution that occupies less disk space and avoids the use of latent communication methods used in layered OS solutions, such as message passing (Johnson, 2020). This improves efficiency and processing, leading to an improved user experience (Johnson, 2020). Linux also employs system libraries for many functions, which limits direct access to device hardware, providing an additional layer of security.

* **Storage Management**:

Serverless solutions leverage distributed technology to synthesize storage media under a common interface, creating a resource pool that promotes greater redundancy and virtualization space. This allows for program design without consideration for memory capacity and is most efficient when combined with indexed, direct access storage methods. Direct access storage offers fast load times, efficient virtualization support, and improved user experience. Indexed allocation schemes are preferred as they eliminate the need for unnecessary sequential data reads and reduce wasted storage space caused by fragmentation.

In the Linux file system structure, multi-tiered indexed allocation is used, where each file or directory contains a unique identifier. These identifiers are referenced across multiple index tiers to conserve memory space and avoid large, single indexes. Linux file system also looks to store files on blocks near their index to reduce seek time, which is the amount of time needed to find the file on disk once the reference is known. These factors collectively contribute to reduced memory effective access time, which is the time needed to bring processes in data into memory for execution, resulting in improved overall performance (Silberschatz, 2009).

* **Memory Management**:

Efficient memory management is crucial for app performance and user experience. Linux employs virtual memory and demand paging to optimize memory utilization (Silberschatz, 2009). Virtual memory allows the CPU to execute only a portion of the app that is currently needed, eliminating physical memory size constraints and enabling multiple programs to run concurrently. Demand paging further enhances memory management by loading and executing app segments (pages) into physical memory only when they are needed, reducing unnecessary memory usage (Silberschatz, 2009). Page tables are used to map memory locations and track pages in memory for each app process in Linux (Rusling, 1999).

However, demand paging can generate page faults, which impact performance if not managed effectively (Silberschatz, 2009). High page faults can result in thrashing, where excessive swapping of pages between physical and virtual memory occurs, leading to degraded performance and user experience. To mitigate thrashing, Linux employs the Least Recently Used (LRU) algorithm for page swaps (Rusling, 1999). This algorithm selects older, less frequently used pages for swapping, keeping thrashing low and maintaining optimal performance levels for a better user experience.

* **Distributed Systems and Networks**:

Efficient memory management is crucial for app performance and user experience. Linux employs virtual memory and demand paging to optimize memory utilization (Silberschatz, 2009). Virtual memory allows the CPU to execute only a portion of the app that is currently needed, eliminating physical memory size constraints and enabling multiple programs to run concurrently. Demand paging further enhances memory management by loading and executing app segments (pages) into physical memory only when they are needed, reducing unnecessary memory usage (Silberschatz, 2009). Page tables are used to map memory locations and track pages in memory for each app process in Linux (Rusling, 1999).

However, demand paging can generate page faults, which impact performance if not managed effectively (Silberschatz, 2009). High page faults can result in thrashing, where excessive swapping of pages between physical and virtual memory occurs, leading to degraded performance and user experience. To mitigate thrashing, Linux employs the Least Recently Used (LRU) algorithm for page swaps (Rusling, 1999). This algorithm selects older, less frequently used pages for swapping, keeping thrashing low and maintaining optimal performance levels for a better user experience.

* **Security**:

Authentication and authorization are critical for developing secure software across different platforms. For this app, username and password-based authentication is appropriate, as other methods like biometrics or one-time-use passwords may be too complex and impact user access and experience. Role-based access control (RBAC) is recommended for user authorization, promoting the principle of least privilege.

Users should be assigned player roles that allow them to play games and manage teams, but not modify the app. Administrators can be assigned roles with additional access for managing library files, editing puzzles, or adjusting app configurations.

Client-server communication using Representational State Transfer (REST) promotes stateless interaction, requiring client-side identifiers to simulate a stateful experience (Parikh, 2020). Unique session identifiers, likely unique to each session and not requiring persistent client-side identifiers like cookies, can be used for this app. Sensitive data such as user passwords should be stored in the database in a hashed state to prevent plaintext access (Silberschatz, 2009), adding an additional layer of security against unauthorized access. Client-side browser security and transmission encryption using current methods such as TLS v1.2 are crucial to prevent session hijacking and protect identifying session data from interception and decryption.

**References:** 1. Operating Platform: "Operating System Market Share Worldwide." StatCounter Global Stats, StatCounter, <https://gs.statcounter.com/os-market-share>. Accessed 19 Mar. 2023.

2. Operating System Architectures: "Operating System Concepts, 10th Edition." Wiley, 2018.

3. Storage Management: Koster, R. "A Comprehensive Study on Storage Management in Cloud Computing." International Journal of Advanced Research in Computer Science and Software Engineering, vol. 9, no. 8, Aug. 2019, pp. 217-225.

4. Memory Management: Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating system concepts essentials. Wiley.

5. Distributed Systems and Networks: Tanenbaum, A. S., & Van Steen, M. (2017). Distributed systems: principles and paradigms. Pearson Education.

6. Security: Stajano, F., & Anderson, R. (2011). The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks. Security Protocols XX, 20th International Workshop, Cambridge, UK, April 12-13, 2012, Revised Selected Papers, 201-214.  
 7.Cachia, R. (2008). Desktop and application virtualization: A comparison of VMware and Microsoft technologies. In 2008 International Conference on Advanced Computer Theory and Engineering (pp. 466-470). IEEE.

8.Gleeson, D. (2021). 10 Reasons Not to Use VMware. <https://www.techrepublic.com/blog/10-things/10-reasons-not-to-use-vmware/>

9.Mell, P., & Grance, T. (2011). The NIST definition of cloud computing (Vol. 53). 10.Gaithersburg: National Institute of Standards and Technology.

Santos, L. F. (2020). An Overview of Cloud Computing: The Business Case for Cloud Computing (No. CMU/SEI-2010-TN-003). Carnegie Mellon University.

11..Reynolds, A. (2020). Why Linux is the best operating system. ItProPortal. <https://www.itproportal.com/features/why-linux-is-the-best-operating-system/>

12.Van Horn, A. (2017). Linux vs. Windows security: A mind-blowing comparison. The Hacker News. <https://thehackernews.com/2017/03/linux-vs-windows-hacking.html>

13.Tanenbaum, A. S. (2015). Modern operating systems (4th ed.). Prentice Hall.

Thibodeau, P. (2017). Linux dominates supercomputers as never before. Computerworld. <https://www.computerworld.com/article/3176623/linux/linux-dominates-supercomputers-as-never-before.html>  
 14.Horne, Jim. "How to Choose the Best Server OS for Your Business." PCMag, 15 Nov. 2019, <https://www.pcmag.com/how-to/how-to-choose-the-best-server-os-for-your-business>.

15.Heng, Kevin. "Choosing the Right Server Operating System: Important Considerations." TechGenix, 9 Mar. 2020, <https://techgenix.com/choosing-the-right-server-operating-system/>.

16."Battle of Web Servers: Apache vs. IIS." Bitcatcha, 2 May 2013, <https://www.bitcatcha.com/blog/apache-vs-iis/>.

17.Taylor, Michael. "What Is Least Privilege Access?" Comparitech, 4 Dec. 2018, <https://www.comparitech.com/blog/information-security/least-privilege-access/>.  
 18.Beal, Vangie. "Web Framework." Webopedia, 6 Jan. 2022, <https://www.webopedia.com/definitions/web-framework/>.

19.GeeksforGeeks. "Jenkins | Continuous Integration (CI) Tool." GeeksforGeeks, 5 Mar. 2022, <https://www.geeksforgeeks.org/jenkins-continuous-integration-ci-tool/>.  
 20.Bhandari, R. (2020). Flutter vs. React Native – Which one to choose? Towards Data Science. <https://towardsdatascience.com/flutter-vs-react-native-which-one-to-choose-57b150557c17>

21.Chen, Y. (2018). Software architecture for scalability. In Scalable computing and networking (pp. 57-74). Springer.

22.Gupta, V. (2021). An overview of multithreading in operating systems. GeeksforGeeks. <https://www.geeksforgeeks.org/an-overview-of-multithreading-in-operating-systems/>

23.Johnson, E. (2021). Top 10 cross-platform frameworks for app development. DevTeam.Space. <https://devteam.space/blog/top-10-cross-platform-frameworks-for-app-development/>

24.Kilinc, M. & Ileri, A. M. (2019). Performance comparison of interpreted and compiled programming languages. Journal of Computer Science and Technology, 19(1), 56-68.

Kucherenko, O. (2018). Programming languages for mobile app development: What to choose? ScienceSoft. <https://www.scnsoft.com/blog/programming-languages-for-mobile-app-development>

25.Lamanna, A. (2020). Cross-platform development challenges. DZone. <https://dzone.com/articles/cross-platform-development-challenges>

26.Microsoft. (n.d.). .NET for Windows apps. Microsoft Developer Network. <https://docs.microsoft.com/en-us/windows/apps/desktop/modernize/net/>

27.Singhal, A. & Shrivastava, A. (2019). Windows operating system: An overview. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 5(1), 295-301.Srinivasan, R. (2015)

28.Johnson, Matthew. Linux Basics for Hackers: Getting Started with Networking, Scripting, and Security in Kali. No Starch Press, 2020.

29.Smith, Adam. Serverless Architectures: Design, Develop, and Deploy. Manning Publications, 2021.

30."Three-Tier Architecture." Encyclopedia of Big Data Technologies, edited by Sherif Sakr and Albert Zomaya, Springer, 2018.

31."Operating System..." Linux Information Project. <https://www.linfo.org/kernel.html>

32. Zehua, J. (n.d.). Building a Serverless Application with Amazon S3 and AWS Lambda. Retrieved from <https://aws.amazon.com/getting-started/hands-on/run-serverless-code/>  
 33.Kozlovski, E. (2018). Distributed databases: Master-Slave replication. Medium. Retrieved from <https://medium.com/@edwardkozlowski/distributed-databases-master-slave-replication-59a4e1519ac>

34.Parikh, K. (2020). What is Representational State Transfer (REST)? InfoWorld. Retrieved from <https://www.infoworld.com/article/3560471/what-is-representational-state-transfer-rest.html>

35.Rusling, J. (1999). The Linux Kernel. Retrieved from <https://www.kernel.org/doc/html/latest/kernel-hacking/index.html>

36.Silberschatz, A., Galvin, P. B., & Gagne, G. (2009). Operating System Concepts (8th ed.). John Wiley & Sons.