

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 3.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc115077317)

[**Table of Contents 2**](#_Toc115077318)

[**Document Revision History 2**](#_Toc115077319)

[**Executive Summary 3**](#_Toc115077320)

[**Requirements 3**](#_Toc115077321)

[**Design Constraints 3**](#_Toc115077322)

[**System Architecture View 3**](#_Toc115077323)

[**Domain Model 3**](#_Toc115077324)

[**Evaluation 4**](#_Toc115077325)

[**Recommendations 5**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 3.0 | 02/19/2024 | Adam Vosburg | Revision including all elements for software development considerations |

## [Executive Summary](#_sbfa50wo7nsh)

At a high level, the GamingRoom application needs to provide a seamless user experience for activities like creating new games, connecting teams and players, and looking up existing data. All of these leverage centralized coordination within the program. For a use case such as adding a new game, the software should intuitively handle the complexity on the backend of ensuring it is saved properly without duplicate names, keeping the data integrity intact.

To satisfy these needs from both a technical and usability lens, I have adopted two industry-standard software design patterns in my Java application, the Singleton and Iterator. Applying the Singleton pattern to the central GameService class creates and uses a single instance of game data globally across the system. Leveraging the Iterator supports cleanly checking for "duplicate name" errors. Altogether, these set up extensible foundations catering to both the client’s end goals and the architecture for long-term agility as new features emerge.

## Requirements

The client for this game management application specified core functional needs centered on flexibility for users to create and organize game and team entities along with upholding data integrity. Specifically, the seamless addition of new games and teams must prevent duplicate names to ensure system consistency.

The centralized Singleton GameService design pattern aligns directly with satisfying the client's needs for controlled creation capabilities by serving as a coordination point validating all additions follow integrity rules. Using the external Iterator pattern for checking new names against records handles their requirement for uniqueness without exposing additional implementation complexity to clients.

Together with supporting relationships between Entity types, these patterns enable robust functionality for clients today and future-proof scalability. Upfront design considerations for access optimization and multi-user concurrency will promote meeting client expectations for performance and responsiveness even as system usage grows over time based on their product roadmap.

## [Design Constraints](#_2et92p0)

Elements inherent to distributed web architecture create two primary design constraints. First, multithreaded access poses performance and consistency risks for the singleton GameService at high load which must be addressed upfront through thread protections and safety checks.

Additionally, while external Iterators avoid inefficiencies encapsulating duplicate validation logic, searching ever-growing datasets linearly on inserts may cause eventual scalability bottlenecks without proactive optimization. Indexing and partitioning large-scale storage requires consideration.

The patterns help with simplicity currently but optimizations ensuring security, smooth concurrent access, and scalability require intentional design to satisfy production rollout across distributed infrastructure.

## [System Architecture View](#_ilbxbyevv6b6)

Please note: Nothing is 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](#_8h2ehzxfam4o)

Reviewing the UML class diagram for the game application reveals it applies core object-oriented principles to efficiently fulfill requirements stated previously around consolidated yet accessible entity management while ensuring uniqueness constraints.

Specifically, the GameService utilizes the Singleton design pattern to act as a centralized authoritative point for game data as operations like adding new games or teams are invoked by client code. This directly aligns with satisfying requirements around controlled creates and reads for core entities like games while lowering risks from the distributed state as GameService becomes the governor enforced through a single instance.

Additionally, leveraging the Iterator abstraction applied externally from main Game data storage demonstrates encapsulation principles keeping implementation details hidden within GameService. This upholds requirements for the locality of change as uniqueness-checking code can evolve independently without affecting clients. Using Iterator also inherently supports scalability requirements by avoiding entire storage searches on every read.

Finally, establishing inherited relationships between Entity, Game, and Team classes exhibits polymorphism principles allowing flexible future entity additions and grouping through hierarchical type relationships. This fulfills requirements for an extensible entity ecosystem as new domain classes can be linked.

Altogether, the UML diagram reflects multiple core OO principles mapping directly to efficiently fulfill functional and quality attribute requirements critical for a scalable application.

**"The Gaming Room UML diagram. The top of the diagram is labeled as com dot gamingroom. Test boxes are placed in two layers. The first layer has three text boxes and the second layer has four of them. In the first layer, the 'ProgramDriver' textbox points to 'SingletonTester' textbox. The 'ProgramDriver' textbox contains the text 'asterisk main round brackets.' The 'SingletonTester' textbox contains the text 'asterisk testSingleton round brackets.' The arrow between these two text boxes are labeled 'open two angle brackets uses close two angle brackets'. In the second layer, there are 'GameService', 'Game', 'Team', and 'Player' text boxes. The 'GameService' textbox has texts arranged in two layers. The first layer contains games colon List open angle bracket Game close angle bracket, nextGamesId colon long, nextPlayer Id colon long, nextTeamId colon long, and service colon GameService. The second layer contains GameService round brackets, getinstance round brackets colon GameService, addGame open parenthesis name colon String close parenthesis colon Game, getGame open parenthesis id colon long close open parenthesis colon Game, getGame open open parenthesis name colon String close open parenthesis colon Game, getGameCount round brackets colon int, getNextPlayerID round brackets colon long, and getNextTeamId round brackets colon long. The 'GameService' box is connected with the 'Game' textbox with a line labeled 'zero dot dt dot asterisk'.  The 'Game' textbox also contains text in two layers. The first layers contains the text teams colon List open angle bracket Team close angle bracket. The second layer has Game open round bracket id colon long comma name colon String close parenthesis, addTeam open parenthesis name colon String close parenthesis Team, toString round brackets colon String. The 'Game' textbox is connected with the 'Team' textbox with a line labeled 'zero dot dt dot asterisk'. The 'Team' textbox also contains text in two layers. The first layers contains the text players colon List open angle bracket Player close angle bracket. The second layer has Team open parenthesis id colon long comma name colon String close parenthesis, addPlayer open parenthesis name colon String close parenthesis colon Player, and toString round brackets colon String. The 'Team' textbox is connected with the 'Player' textbox with a line labeled 'zero dot dt dot asterisk'. It contains the text Player open parenthesis id colon long comma name colon String close parenthesis and toString round brackets colon String. The 'Game', the 'Team, and the 'Player' boxes point to the 'Entity' textbox in first layer. The 'Entity' textbox contains text in two layers. The first layer has the text id colon long and name colon String. The second layer has Entity round brackets, Entity open parenthesis id colon long comma name colon String close parenthesis, getId round brackets colon long, getName round brackets colon String, toString round brackets colon String.**

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Pros:   * File sharing, caching, and time machine servers bundled in macOS lower barriers to entry   Cons:   * macOS Server discontinued so no future updates or support * Migration off services like DNS and DHCP complex, requiring deep evaluation of alternatives upfront and IT effort to implement * Continued Profile Manager dependence risky if Apple retires it; replacing MDM capabilities would incur more costs | Pros:   * Highly scalable server distributions for cost-efficient large deployments * Flexible open-source options catering to diverse hosting needs * Mature technologies like LAMP stack for proven web app reliability   Cons:   * Fragmentation - supporting diverse distros adds client-side complexity * Admin overhead is higher lacking centralized management * More responsibility for in-house expertise for security and updates | Pros:   * Windows Server delivers reliable enterprise hosting capabilities * Integrated suite via IIS, .NET Framework, and ASP.NET for full-stack needs * Role hierarchy facilitates granular server permissions management   Cons:   * High licensing costs at scale requiring large budgets * Restrictive policies around third-party software and add-ons * Heavy resource demands mandating high-spec hardware | Pros:   * Managed backend solutions enable scalability without ops overhead * Integrated monitoring, logging, and alerting reduce dev workload * Services like push notifications included to boost engagement   Cons:   * Vendor dependence and cost at high usage volumes * Constraints around customization to proprietary platforms * Added latency over direct infrastructure access |
| **Client Side** | Pros:   * Mature frameworks like AppKit and SwiftUI provide rich capabilities for intuitive UIs * Native Apple guidelines compliance allows tapping huge iOS/macOS user base * Integrations like Handoff facilitate seamless experiences across Apple devices   Cons:   * Limitation to Apple ecosystem cuts out Windows, Android, and Linux users * Restrictive review policies and walled ecosystem reduce flexibility * Requirement to use the latest native SDKs for features poses app maintenance overhead | Pros:   * Open-source nature facilitates tailored solutions * Universal package formats ease distribution over varied distros * Lightweight and customizable desktop options   Cons:   * Desktop fragmentation challenges consistent UI development * Limited consumer desktop market share * Weak gaming and specialty application support | Pros:   * Dominant desktop market share ensures broad application support * Mature frameworks like WinForms and WPF for responsive client apps * Extensive backward compatibility minimizing upgrade issues   Cons:   * High system requirements exclude older or lightweight hardware * Increased security vulnerability surface area over rivals * Lagging pace of modern UI framework advancement | Pros:   * Platform standardization delivering polished UX benchmark * Access to device capabilities via extensive native APIs * Availability through dominant central app stores   Cons:   * Walled ecosystems mandating platform-specific dev skills * App store policies and fees reduce flexibility * Hardware and OS fragmentation within the Android ecosystem |
| **Development Tools** | Pros:   * Tight Xcode integration with macOS and iOS SDKs * Interface Builder streamlines visual UI development * Swift language offers modern capabilities and safety   Cons:   * Xcode is historically slower and prone to instability vs competitors * Strong Apple ecosystem lock-in with steep learning curves * Poor Windows interoperability for broader teams | Pros:   * Breadth of open-source programming languages available * Flexibility to use preferred text editors and command line tools * Avoid vendor lock-in   Cons:   * No unified centralized IDE experience * Varying dependencies and configurations across distros * Weaker design tooling for GUI development | Pros:   * Feature-packed Visual Studio IDE with seamless Microsoft stack integration * Abundance of documentation and dev community support * Platform versatility spanning web, client, and mobile apps   Cons:   * IDE instability and high memory usage requiring robust systems * Suboptimal web development support centered on the .NET ecosystem * Steep learning curve for full Visual Studio capabilities | Pros:   * Mature IDEs (Android Studio, Xcode) catered specifically for needs * Integrated SDKs and emulators accelerate build/test iterations * Robust component libraries simplify complex functionalities   Cons:   * COMPLEXITY in supporting two separate native mobile platforms * Poor code reuse and UI sharing between iOS and Android * Inferior emulation vs real devices for compatibility testing |

## 

## Recommendations

1. **Operating Platform:**

I recommend Amazon Web Services (AWS) specifically the Elastic Compute Cloud (EC2) platform for hosting Draw It or Lose It. AWS provides configurable cloud servers to host the web and application logic, along with a full suite of integrated networking, storage, database, and security services (Amazon Web Services [AWS], n.d.). EC2 offers options ranging from basic virtual machines to high-performance computing clusters if needed to scale up. Features like AWS Auto Scaling automatically add or remove EC2 instances in response to demand. ELB load balancing routes across zones. The AWS infrastructure is backed by an SLA with extensive reliability and security compliance. Leveraging this mature, fully managed IaaS platform removes the headaches of maintaining physical hardware. AWS offers a pay-as-you-go cost model allowing flexibility. Instance types size to needs and hourly billing apply for effective economics.

1. **Operating System Architectures:**

I recommend using Amazon Linux 2 on the provisioned EC2 computing resources. This Linux distribution from AWS optimizes integrating natively with associated services. Docker container images provide packaging and versioning for Draw It or Lose It's components like the web front-end, application logic backend, databases, etc. Containers enhance portability across environments and standardization regardless of the underlying host OS. Defining dependencies explicitly also simplifies deployment. Updates roll out through refreshed images. Modular containers assist availability; issues with one isolated service will not take down others. Orchestration platforms like Kubernetes handle container lifecycles reducing the administrative overhead of managing distributed, compartmentalized elements.

1. **Storage Management:**

I recommend using Amazon Simple Storage Service (S3) for game asset storage. This provides geo-redundant durable object storage at high scalability. Assets reside as objects in S3 "buckets" with unlimited capacity potential allowing growth. Buckets are arranged into a filesystem-like hierarchy as needed and these fine-grained access controls secure data exposure (Gartner Peer Insights, 2024). In terms of optimizing cost with frequent and infrequent asset access patterns, S3 Standard provides general availability while S3 Infrequent Access reduces costs for less utilized data through lower redundancy. Automated tiering achieved by S3 Lifecycle policies transitions objects meeting filters like age or access frequency to lower cost tiers. This will shift stale assets requiring preservation but rare retrieval to more economical storage while keeping actively used data in S3 Standard. Background data integrity checks to safeguard against corruption without administrative upkeep requirements.

1. **Memory Management:**

To improve backend throughput, I recommend leveraging Redis, an open-source in-memory data store, for fast data lookup caching as well as transient storage of session information and calculated query results. This keeps computed values in memory for low latency retrieval compared to secondary storage. The Redis in-memory database runs as a separate container but integrates with the application containers to provide this shared cache layer. For the language running the web application server whether JavaScript/Node.js or something like Python/Django, memory is automatically managed via garbage collection. This will free up engineers from complex allocation logic. Load testing provides profiling to fine-tune memory for peak efficiency across average and spike request volumes.

1. **Distributed Systems and Networks:**

Adopting a microservices architecture where backend components like user management, game data, and analytics ingestion compartmentalize into distinct services with REST APIs facilitating intercommunication promotes availability and scalability. If issues arise with one service, the others continue functioning rather than a monolith where problems cascade. Microservices also allow independent scaling. REST over HTTP with structured request data formats like JSON enables universal communication between other services, web apps, or mobile apps. Inside AWS, regional redundancy exists across data centers and services replicate across zones (data centers) by default. ELB load balancing routes requests across replicated app instances to prevent singular failures. RDS databases provision read replicas to offload analytical workloads too. If an outage occurs due to a zone failure, the autoscaling infrastructure automatically rebalances by firing up resources in alternate zones to mitigate disruption.

1. **Security:**

Applying security best practices ensures the protection of sensitive user data as well as systems against breach or abuse. Authentication requires integration with a service like Amazon Cognito; This lets users validate against an identity provider and assuming correctly scoped permissions receive temporary AWS credentials under the hood to sign requests. Audit logs monitor for suspicious activity with alarms triggered by CloudWatch for investigation if anomalies arise. Encryption of data both at rest and in transit protects assets like user personal data or application secrets; AWS Key Management Service (KMS) provides lifecycle management capabilities for encryption keys leveraged by storage and data transfer mechanisms. Meanwhile keeping infrastructure patched against newfound vulnerabilities occurs through AWS Config defining policies that track for known misconfigurations or run vulnerability scanners. Remediation steps can be automated upon deviations from best practices being detected. Role-based access controls restrict data and actions to essential needs.

**Reference:**

Amazon Web Services (AWS) (n.d.). What is Amazon EC2? Amazon Elastic Compute Cloud. Retrieved from <https://aws.amazon.com/ec2/>

Gartner Peer Insights. (2024). Amazon Simple Storage Service (Amazon S3) Reviews, Ratings & Features 2024. Gartner Peer Insights. <https://www.gartner.com/reviews/market/cloud-object-storage/vendor/amazon-web-services/product/amazon-s3/>