REQ5: Design Rationale (A.P.I Integration: Dynamic Quest System with Gemini AI)

A. Quest generation strategy selection

Design A1: Hardcoded Gemini calls throughout the codebase. Scatter API calls directly in the Questmaster class or action handlers, constructing HTTP requests inline wherever a quest is needed.

Pros	Cons
Minimal abstraction; direct API usage.	Violates SRP: quest generation logic mixed with NPC/action logic.
Fewer classes initially.	Tight coupling: changing API provider requires editing multiple classes.
	DRY violation: HTTP request construction, JSON parsing, and error handling duplicated across call sites.
	Hard to test: cannot mock API calls or provide fallback behavior.
	No extensibility: adding local generation or alternative APIs requires refactoring.

Design A2: Strategy Pattern with QuestService interface. Introduce a QuestService interface defining generateQuest(). Provide two implementations: GeminiQuestGenerator (uses API) and LocalQuestGenerator (fallback templates). A factory selects the appropriate strategy at runtime based on configuration.

Pros	Cons
High cohesion: generation logic isolated in dedicated service classes.	Adds several classes (interface, implementations, factory).
OCP: new generation strategies can be added without modifying consumers.	Requires indirection through the service interface.
SRP: Questmaster focuses on interaction; generators focus on quest creation.	
Testability: can inject mock service or test each generator independently.	
DIP: depends on QuestService abstraction, not concrete API client.	
Easy fallback: gracefully degrades to local generation when API unavailable.	

We choose Design A2 because it provides modularity, extensibility, and separation of concerns. The Strategy Pattern allows runtime selection between AI-generated and template-based quests without modifying the Questmaster or QuestAction classes, adhering to the Open-Closed Principle and enabling graceful degradation.

B. API client abstraction and HTTP communication

Design B1: Direct HttpClient usage in generator. GeminiQuestGenerator directly instantiates java.net.http.HttpClient and constructs requests inline, parsing JSON responses with string manipulation.

Pros	Cons
Uses built-in JDK libraries; no external dependencies.	Tight coupling: generator tied to specific HTTP implementation.
Straightforward implementation.	Hard to test: cannot mock network calls without complex setup.
	Violates SRP: generator handles both quest logic and HTTP communication.
	Closed to extension: switching to a different HTTP library requires rewriting generator.

Design B2: GeminiClient interface with HttpGeminiClient implementation. Introduce a GeminiClient interface defining generateText(). HttpGeminiClient implements HTTP communication, URL encoding, JSON serialization, and response parsing. GeminiQuestGenerator depends on the abstraction.

Pros	Cons
DIP: generator depends on abstraction, not concrete HTTP implementation.	Adds interface and implementation class.
SRP: HTTP concerns separated from quest generation logic.	Slight increase in indirection.
Testability: can inject mock client for unit testing.	

OCP: can swap HTTP implementations without changing generator.	
Clear boundary: API communication isolated from domain logic.	
Future extensibility: easy to add retry logic, rate limiting, or caching.	

We choose Design B2 because it decouples API communication from quest generation logic, enabling independent testing and future enhancements. The abstraction allows mocking during tests and provides a clean extension point for adding features like request caching or alternative HTTP libraries.

C. Configuration management and API credentials

Design C1: Configuration scattered across classes. Each class that needs API credentials reads environment variables directly using System.getenv() or hardcodes default values.

Pros	Cons
No configuration class needed.	DRY violation: configuration reading logic duplicated across classes.
Direct access to values.	Tight coupling: every class knows about environment variable names.
	Hard to change: updating configuration sources requires editing multiple classes.

	No single source of truth for default values.
	Violates SRP: classes mix business logic with configuration reading.

Design C2: Centralized ApiConfig utility class. Create ApiConfig with static methods (apiKey(), model(), apiVersion(), isConfigured()) that encapsulate all configuration reading logic, checking environment variables, system properties, and providing sensible defaults.

Pros	Cons
SRP: single class responsible for configuration management.	One additional utility class.
DRY: configuration logic centralized, not duplicated.	Static methods (acceptable for configuration utilities).
Clear contract: well-defined methods for each configuration value.	
Easy to extend: adding new configuration options only requires editing one class.	
Consistent fallback behavior: defaults defined in one place.	
Simplifies testing: can override system properties for test configurations.	

We choose Design C2 because it provides a single source of truth for configuration, eliminates duplication, and makes it trivial to modify configuration sources (e.g., adding file-based config) without affecting consumers. The isConfigured() method enables clean fallback logic in the factory.

D. Reward distribution and item mapping

Design D1: Reward parsing hardcoded in QuestAction. QuestAction's execute() method contains switch statements or if-else chains that parse reward names and instantiate items directly.

Pros	Cons
All reward logic visible in one method.	Violates SRP: action class handles both UI flow and item instantiation.
Fewer classes.	Tight coupling: action depends on concrete item classes.
	DRY violation: reward parsing logic cannot be reused elsewhere.
	Closed to extension: adding new reward types requires modifying action.
	Hard to test: cannot test reward distribution independently of action execution.

Design D2: RewardDistributor interface with fuzzy matching strategy. Introduce RewardDistributor interface defining distribute(). SimpleRewardDistributor implements fuzzy matching (case-insensitive substring matching) to map AI-generated reward names to concrete items, with random fallback for unknown rewards.

Pros	Cons
SRP: reward distribution logic separated from action flow.	Additional interface and implementation.
OCP: can add new distributors (e.g., economy-based, rarity-based) without modifying actions.	Fuzzy matching may be imprecise for some edge cases.
Testability: can test distribution logic independently.	
Flexibility: handles both predefined and AI-generated reward names gracefully.	
Safety: unknown rewards still grant something valuable (random fallback).	
Clear extension point: future distributors can implement different strategies.	

We choose Design D2 because it decouples reward interpretation from game flow, handles the unpredictability of AI-generated content gracefully, and provides a clean abstraction for future reward systems. The fuzzy matching bridges the gap between creative AI output and fixed game items

E. Quest participant registration and decoupling

Design E1: Direct instanceof checks in quest system. Quest-related code uses instanceof checks to determine if an Actor can participate in quests, casting as needed.

Pros	Cons
No additional interfaces or registry.	Tight coupling: quest system depends on concrete actor types.
Direct type checking.	Violates OCP: adding new quest participants requires modifying quest code.
	Hard to extend: cannot make arbitrary actors quest-capable without inheritance.
	Poor type safety: requires casting after instanceof checks.
	Violates DIP: depends on concrete classes, not abstractions.

Design E2: QuestParticipant interface with registry pattern. Introduce QuestParticipant marker interface exposing getQuestTracker(). QuestParticipantRegistry maintains a WeakHashMap mapping Actor instances to QuestParticipant implementations, avoiding memory leaks.

Pros	Cons
Decouples quest system from actor hierarchy.	Additional interface and registry class.
OCP: any actor can become quest-capable by registering.	Requires explicit registration.

Type-safe: no casting required.	
DIP: quest code depends on QuestParticipant abstraction.	
Flexible: actors can participate without inheritance changes.	
Memory-safe: WeakHashMap prevents memory leaks when actors are removed.	

We choose Design E2 because it completely decouples the quest system from the engine's Actor hierarchy, allowing quest participation to be added to any actor without modifying class hierarchies. The registry pattern with WeakHashMap provides flexibility without memory management concerns.

F. Quest state management and progress tracking

Design F1: Quest progress tracked in QuestAction. Store active/completed quests in QuestAction as instance variables, checking progress during action execution.

Pros	Cons
All quest data co-located with action logic.	Violates SRP: action handles both interaction and state management.
Simple implementation.	Quest data lost when action completes (no persistence).
	Tight coupling: cannot access quest data outside action context.

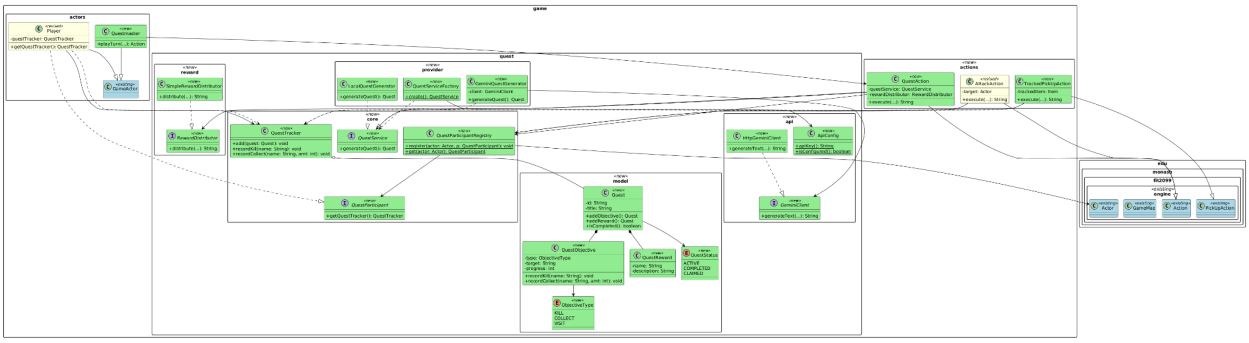
	Cannot share quest state across multiple actions or systems.
	Hard to implement quest triggers (kills, collections) from other parts of codebase.

Design F2: QuestTracker component with event recording methods. Create QuestTracker class maintaining active/completed quest maps. Provide recordKill(), recordCollect(), recordVisit() methods that update all relevant quest objectives and automatically move completed quests.

Pros	Cons
SRP: tracker focused solely on quest state management.	Additional tracker class.
High cohesion: all progress tracking logic centralized.	Requires explicit event recording from gameplay code.
Reusable: any game system can record quest progress.	
Extensible: easy to add new objective types (e.g., craft, trade).	
Clear ownership: each participant has their own tracker instance.	

Testable: can test progress logic independently of actions.	
Future persistence: easy to serialize/deserialize tracker state.	

We choose Design F2 because it separates quest state from interaction logic, enabling quest progress to be triggered from anywhere in the codebase (combat, item pickup, movement). The centralized tracker provides a foundation for future persistence and cross-system quest integration.



The diagram (see UML) shows the final design of requirement 5, which utilizes each Design 2 of parts A through F stated above to implement a dynamic quest system that integrates the Gemini AI API for generating contextual quests with predetermined fallback to local generation. The Questmaster NPC serves as the interaction point, providing QuestAction to adjacent actors, which acts as a Facade coordinating quest generation, state management, and reward distribution.

The quest generation strategy follows the Strategy Pattern, where QuestServiceFactory selects between GeminiQuestGenerator (when API credentials are configured) and LocalQuestGenerator (fallback templates) at runtime. This ensures the game remains functional regardless of API availability or configuration status. GeminiQuestGenerator depends on the GeminiClient abstraction rather than concrete HTTP implementations, with HttpGeminiClient handling all network communication, JSON parsing, and response extraction using only JDK built-in libraries to satisfy the assignment's constraint on external packages. ApiConfig centralizes all configuration management, eliminating the need to scatter environment variable reads throughout the codebase.

The quest system decouples from the engine's Actor hierarchy through QuestParticipantRegistry, which uses WeakHashMap to prevent memory leaks while allowing any actor to participate in quests without modifying inheritance hierarchies. Each participant's QuestTracker manages their quest portfolio, automatically progressing objectives and transitioning completed quests when recordKill(), recordCollect(), or recordVisit() events occur anywhere in the game. The quest model follows a three-tier structure: Quest objects contain QuestObjectives (supporting KILL, COLLECT, and sequential VISIT types) and QuestRewards, with lifecycle status tracking from ACTIVE through COMPLETED to CLAIMED.

The SimpleRewardDistributor addresses the challenge of mapping unpredictable AI-generated reward names to concrete in-game items through fuzzy substring matching. This allows the system to interpret creative outputs such as "Mystical Flame Sword" and match them to existing items, while unknown rewards trigger a random fallback mechanism that ensures consistent reward distribution regardless of AI output variability. The design demonstrates strong adherence to SOLID principles: each class maintains a single responsibility, depends on abstractions rather than concrete implementations, and remains open for extension without modification.

Since quest generation, API communication, state tracking, and reward distribution logic are encapsulated entirely within dedicated classes, the system avoids duplicating HTTP request construction, quest parsing, progress checking, or reward mapping code (DRY). The architecture minimizes problematic connascence through explicit enums, named parameters, and localized coupling. Extension points are provided for future features including new objective types (CRAFT, TRADE, ESCORT), alternative reward distribution strategies (economy-based, rarity systems), quest persistence through serialization, and context-aware AI generation that incorporates player statistics, world state, or quest history into the generation prompt. This modular architecture ensures maintainability and extensibility while adhering to object-oriented design principles throughout the implementation.