Design Project Design Summary
Group 22 – Andy Tran, Ethan Irimiciuc, Jacob Williams, Minh Luong

The Design section establishes the technical architecture for Poachers vs. Animals, showing how we transform the requirements into an actual system that can be built and maintained. This section bridges the gap between what the game needs to do and how it will actually work under the hood.

Our design goals focus on several key priorities. Educational impact remains central, with every technical decision supporting the mission of teaching players about conservation through debriefs that connect their actions to real-world consequences. Performance targets ensure the game runs smoothly at 30+ frames per second on minimum-spec hardware with AI responding in under two seconds. Reliability comes from auto-saving every two minutes and allowing full offline campaign play. Security protects players through age verification and anti-cheat systems. Usability makes the game accessible through intuitive controls, colorblind support, and quick tutorials. Finally, modularity allows us to add new regions and species later without rewriting core systems.

The system analysis identifies the main classes by examining our project description grammatically. We looked at nouns to find entities like Player, Mission, Campaign, Animal, Patrol, Environment, and EducationalContent. Verbs revealed relationships - players complete missions, patrols detect players, animals respond to threats. This process gave us a clear object-oriented structure that mirrors how the game actually works. Dynamic modeling then describes how these classes interact during gameplay. When starting a mission, the player selects from available campaigns, which triggers the MissionController to load configuration, spawn animals and patrols, set objectives, and start the auto-save timer. Detection events calculate visibility based on distance and cover, transition patrols into pursuit mode, and make animals flee or defend. Multiplayer sessions connect players through matchmaking with anti-cheat checks before syncing game state. After missions, the debrief system processes outcomes and generates educational content explaining conservation impact.

Our system architecture uses a three-tier client-server model. The client layer runs on the player's PC handling interface, graphics, input, and local simulation. The game engine layer manages mission flow, AI behavior, multiplayer coordination, and educational content. The data layer stores player accounts, species information, regional data, and mission configurations. Single-player runs entirely on the client without server connection, while multiplayer requires servers for matchmaking and synchronization. We organized this into nine subsystems: Game Control coordinates everything, Player Management handles authentication and progress, Campaign & Mission structures objectives, Animal AI and Patrol AI run behavior systems, Environment simulates terrain and weather, Educational Content delivers learning materials, Persistence manages saving, Security handles authentication and anti-cheat, and UI coordinates all interface elements.

The hardware mapping distributes components across client PCs running the game application, game servers hosting multiplayer sessions and validating actions, and database servers storing accounts and progress. Network communication uses TCP for mission results where reliability matters and UDP for position updates where speed is critical. HTTPS secures authentication. Minimum requirements are Windows 10, dual-core 2.5GHz CPU, 8GB RAM, DirectX 11 graphics, and broadband for multiplayer.

Persistent data management ensures nothing gets lost. Player profiles store progress and settings, mission saves capture everything needed to resume gameplay, and the system auto-saves every two minutes. Configuration files remember preferences while educational content caches locally for offline access. We use the Repository pattern so the game can save to local files or sync to cloud storage without changing game logic. Everything serializes to JSON for easy debugging.

Security protects players and game integrity through multiple layers. Age verification blocks users under 13 at account creation. Anti-cheat monitors client-side for hacks while server-side validates that actions are physically possible. TLS encrypts communication, bcrypt hashes passwords, and session tokens expire after two hours. Players control privacy settings and can opt out of analytics. Violations escalate from warnings to temporary bans to permanent bans with evidence logging.

Global software control centers on GameController managing game state and transitions between menus, missions, and debriefs. ResourceManager optimizes memory by loading only current assets. EventBus provides decoupled communication so systems can react to events without direct dependencies. ErrorHandler catches crashes and attempts auto-save before offering recovery. PerformanceMonitor watches frame rates and automatically reduces graphics quality when needed.

Boundary conditions define system behavior during critical moments. Startup loads configurations, initializes graphics, connects to authentication if online, and loads player profiles - defaulting to offline mode if servers are unreachable. Mission initialization loads environment assets, spawns AI entities, and begins objective tracking. Normal shutdown saves state, disconnects sessions gracefully, and releases resources with confirmation if exiting during missions. Crash handling does emergency auto-save, logs errors, and offers recovery on restart, with two-minute auto-save intervals minimizing typical progress loss.

We applied standard design patterns for organization and maintainability. Singleton ensures single instances of managers, Factory centralizes object creation, Strategy allows flexible AI behaviors, State manages complex transitions, Observer enables event-driven updates through EventBus, Command wraps player actions for input remapping, and Repository abstracts data storage. The final system integrates everything into packages: Core provides foundations, Player/Campaign/Gameplay implement domain logic, Multiplayer handles networking, Education manages content, UI coordinates interface, Data implements persistence, and Security enforces protection.

Object design details key classes. Player Management includes Player handling login and missions plus PlayerProfile storing progress. Campaign & Mission contains Campaign tracking advancement and Mission running objectives. Gameplay has Animal, Patrol, and Environment classes. Multiplayer provides Server and Session. Educational Content includes content display and DebriefGenerator. Data Persistence gives SaveManager and DataRepository for storage abstraction. This design transforms requirements into concrete implementation plans, establishing patterns and architectures that support educational goals while meeting professional standards for performance, reliability, and user experience.