RAIInet Design Document

Introduction

RAIInet is a two-player strategy game implemented in C++ demonstrating OOP design principles. The game features an 8×8 board where players control eight links (viruses and data with strengths 1-4) initially placed face-down. Each turn, players may use one ability and move a link one space in a cardinal direction. The game incorporates information discovery, battles between links, special abilities, multiple victory conditions, a text display, and a graphics display.

The project showcases multiple design patterns including MVC, Observer, and Strategy patterns (polymorphic abilities). The architecture supports both text-based and graphical displays simultaneously, with each player having their own perspective on revealed information about opposing links. The system is designed for extensibility, supporting easy addition of new abilities, display modes, and game variations.

Overview

Overall System Architecture The RAIInet system uses Model-View-Controller architecture that cleanly separates concerns and provides excellent maintainability and extensibility:

Core Game Engine:

- **GameModel:** Central hub maintaining authoritative game state including board, players, current turn, and game rules. Inherits from Subject for Observer pattern and manages all game logic including movement validation, battle resolution, and win condition checking.
- **Board:** Manages an 8×8 grid of cells, with each cell capable of holding links, having a special firewall type, or serving as a server port. Handles coordinate validation and provides controlled access to individual cells.
- **Cell:** Represents individual board positions supporting multiple types (Normal, ServerPort, Firewall). Each cell can simultaneously hold a link pointer, firewall ownership information, and type-specific metadata.
- **Player:** Manages individual player state including owned links (std::unique_ptr<Link>), available abilities, download counts, and knowledge about opponent links discovered through gameplay.

Game Components:

- **Link:** Represents game pieces with properties like type (Data/Virus), strength (1-4), boosted status, and reveal state. Links maintain references to their owners and support type switching through Polarize ability.
- **Ability system:** Polymorphic Ability base class with eight concrete implementations (LinkBoost, Firewall, Download, Scan, Polarize, Exchange, Hijack, GoLater). Each ability encapsulates its own argument parsing and execution logic.
- **GameSetup:** Handles command-line parsing, configuration validation, and game initialization. Supports flexible configuration through ability strings, link files, and graphics mode selection.

Display System:

- **TextDisplay:** Character-based output with player-specific perspectives. Implements selective information display based on current player's knowledge and ownership.
- **GraphicsDisplay:** X11-based graphical interface with added double-buffering for buttery smooth updates. Maintains separate pixmaps for each player's perspective and supports real-time perspective switching.
- Both displays implement the Observer interface for automatic updates after being notified of game events (e.g. ability use, link movement).

Control System:

- **GameController:** Manages user input, command parsing, gameplay flow, and maintains command history for replay functionality. Supports interactive play, batch command execution via sequence files, and comprehensive error handling.

Key Design Patterns Implementation

Model-View-Controller (MVC) System strictly separates game logic (Model), display concerns (View), and user interaction (Controller). GameModel contains all game rules without presentation logic. Multiple view classes (TextDisplay and GraphicsDisplay) handle different display modes independently. GameController processes commands and orchestrates game flow while remaining decoupled from model internals and view implementations.

Observer Pattern GameModel inherits from Subject and maintains a list of Observer objects (displays). When game events occur (link movement, ability use, game over, etc.), all registered observers are notified automatically through specific ChangeEvent types, ensuring synchronized updates across multiple display modes without tight coupling.

Strategy Pattern Ability system uses polymorphism where each ability implements the Ability interface with a common execute() method. This allows runtime selection and execution of different abilities while maintaining a consistent interface. New abilities can be added without modifying existing code.

Design

Ability System Design: The ability system represents a strategy design pattern implementation balancing flexibility with type safety and performance.

Core Architecture: Each ability inherits from abstract Ability base class defining:

```
    virtual void execute(GameModel& model, std::vector<std::string> args) = 0;
```

This design allows each ability to parse and verify its own arguments with tailored validation, implement unique game mechanics, maintain its own state (used/unused), and integrate seamlessly with the game's command system.

Argument Handling: Rather than using a generic parameter system, each ability receives string arguments and performs its own parsing. This provides type-safe parameter validation at ability level (e.g. Firewall validates coordinates, Exchange validates link ownership), flexible parameter schemes supporting different argument counts and types, clear error messages specific to each ability's requirements, and easy extensibility for new parameter types without changing the base interface.

State Management: Each ability tracks usage state through markUsed() method and integrates with the player's turn management system. GoLater ability demonstrates sophisticated state coordination by modifying player's abilitiesAllowedThisTurn for subsequent turn.

Graphics System and Double Buffering The graphics system implements sophisticated double-buffering for smooth, flicker-free updates while supporting multiple player perspectives.

Buffer Management: Two off-screen pixmaps (buffer1, buffer2) maintain complete board states for each player perspective. Each buffer contains the game from one player's viewpoint, showing their links fully while hiding opponent information appropriately. Only changed cells are redrawn using changedCells tracking system, minimizing computational overhead.

Perspective Management: Graphics system dynamically switches between player perspectives by maintaining separate buffers with different information visibility based on player knowledge, using viewerId to determine what information to display for each observer, and seamlessly switching active buffers during turn transitions through efficient pixmap copying.

Efficient Updates: Changed cells are tracked in GameModel::changedCells and only modified regions are redrawn. The system uses updateBuffersWithChangedCells() to refresh both buffers from their respective perspectives, ensuring consistency across view switches.

Command Processing and History System The command processing system balances user-friendliness with robust error handling and provides advanced features like command history and replay.

Multi-layered Command Handling: Primary command parsing in GameController::play() handles basic syntax and command routing. Specialized handlers for each command type (handleMove, handleAbility, handleBoard, handleSequence) provide focused processing. Nested command processing through handleSequence() enables batch execution and recursive sequence file support.

Command History and Replay: System maintains complete history of valid commands in std::vectorstd::string commandHistory, enabling automatic generation of replay files with unique timestamps (test1234.txt), sequence file execution for testing and automated gameplay, and debugging support through complete command tracing and reproducible game states.

Error Handling Strategy: Commands are validated at multiple levels: syntax validation at controller level (correct number of arguments, valid command names), semantic validation in model (link ownership, valid board positions, ability availability), and game rule enforcement (turn restrictions, ability usage limits, movement constraints).

Observer Pattern Implementation The Observer pattern implementation goes beyond basic notification to provide efficient, targeted updates.

Selective Notification: Different events (ChangeEvent enum) allow observers to respond appropriately: GameStart triggers full board redraw and initial display setup, LinkMoved updates specific cells and player information panels, AbilityUsed refreshes ability counts and affected board areas, DownloadOccurred updates download counts and removes links from display, and TurnEnded switches player perspectives and updates active player indicators.

Performance Optimization: Text display implements player-specific filtering through viewerId, only displaying full information during relevant player's turn. This reduces unnecessary output and improves user experience by showing appropriate information hiding.

Board and Cell Architecture The board system implements clean separation between spatial organization and cell contents.

Cell Type System: Cells support multiple types through CellType enum (Normal, ServerPort, Firewall) with type-specific behavior. ServerPorts trigger downloads when opponent links enter, owned by specific players. Firewalls affect virus links from opposing players, with ownership tracking. Normal cells provide standard movement and battle resolution.

Flexible Content Management: Each cell can simultaneously hold a link pointer (owned by either player but referenced here), firewall ownership information for effect resolution, and type-specific metadata for server port and firewall behavior. This design allows complex interactions like links moving over firewalls while maintaining clear separation of concerns between board structure and game piece management.

Game Setup and Configuration The game setup system demonstrates robust input validation and flexible configuration management.

Multi-source Configuration: System supports configuration from command-line arguments for abilities (-ability1 LFDSP) and link files (-link1 sample_links.txt), file-based link configurations with comprehensive validation, random generation as fallback when files are not provided, and graphics mode selection through -graphics flag.

Validation Strategy: Configuration validation occurs at multiple levels: syntax validation for ability strings (exactly 5 characters, valid ability codes), semantic validation ensuring exactly one of each link type (V1-V4, D1-D4), and constraint validation (maximum 2 of each ability type, proper file formats).

Memory Management and Smart Pointers: The system uses smart pointers throughout to eliminate manual memory management. All ownership relationships use std::unique_ptr (e.g. Player::links, Player::abilities, GameModel::players), while raw pointers serve only as non-owning references (e.g. Cell::link, Cell::firewallOwner). This provides automatic cleanup, clear ownership semantics, and exception safety during complex operations like link transfers.

Design Evolution from Original Plan

One of the first deviations we had to make from our initial plan was our game logic distribution. Initially, we planned for the Board class to manage all link movement and firewall placement. However, we found it more effective for GameModel to directly handle link movement and enforce game rules, since it has full context of the game state and turn order. For firewalls, instead of centralizing their management in Board, we shifted responsibility to individual Cell objects, allowing each cell to manage its own firewall state and ownership. This change improved cohesion, simplified logic, and made it easier to handle abilities and interactions involving firewalls.

We also introduced a GameSetup class, which was not part of our original plan. This allowed us to decouple game initialization logic from the GameController, which now focuses solely on gameplay mechanics. As a result, modifying or extending the setup process can be done cleanly within the GameSetup class without affecting core gameplay logic.

Another change involved our display system. While we retained two instances of TextDisplay (one per player), we realized that having two GraphicsDisplay instances would be redundant and confusing from a UI perspective. Instead, we opted for a single shared GraphicsDisplay and introduced a setViewerId method, enabling the display to render the board from different player perspectives as needed.

Lastly, to support our enhanced feature of command history tracking we added a std::vector<std::string> commandHistory field to GameController. This allows us to record all commands issued during a game, which is then saved to a file when the game ends. This addition supports debugging, replay functionality, and potential extensions like undo/redo.

Cohesion and Coupling Analysis

High Cohesion Examples: The GameModel class exhibits excellent cohesion by containing all game logic, rules, and state management. Every method relates directly to maintaining and manipulating game state, from moveLink() handling movement mechanics to isGameOver() checking victory conditions. Individual ability classes each demonstrate perfect cohesion by containing only logic specific to that ability. Methods focus exclusively on argument parsing, validation, and execution of specific ability's effects without unrelated functionality. Display classes (both TextDisplay and GraphicsDisplay) maintain high cohesion by focusing exclusively on presentation concerns. All methods relate to formatting and displaying game information without containing game logic or state management.

Low Coupling Achievements: The Observer pattern implementation achieves excellent decoupling between GameModel and display classes. Display classes depend only on Observer interface and ChangeEvent notifications, allowing complete independence of display logic from game logic implementation details. Our ability system isolation means individual abilities are completely decoupled from each other and from game models beyond standard interface. New abilities can be added without modifying existing code, and ability implementation details are fully encapsulated. Command processing separation ensures the GameController is loosely coupled to GameModel through a clean interface. Command processing logic is separated from game logic, allowing either to be modified independently without affecting the other.

Resilience to Change

Architectural Flexibility Our RAIInet architecture demonstrates exceptional resilience to change through careful abstraction and loose coupling.

Adding New Abilities: Polymorphic ability system makes adding new abilities straightforward.

1. create new class inheriting from Ability
2. implement execute() method with specific logic of the new ability
3. add ability character to Player::setAbilities() switch statement
4. update ability name mapping in GameController::abilityFullName()

This process requires no changes to core game logic, display systems, or command processing. We added three custom abilities (Exchange, Hijack, GoLater), and adding more is relatively simple.

Adding New Displays: Our observer pattern allows for unlimited display types without changing the model. Adding new displays requires implementing the Observer interface with appropriate notify() handling, registering with GameModel through addObserver(), and handling relevant ChangeEvent notifications for that specific display type.

Game Rule Modifications: We can easily change the rules of the game because our game logic is centralized in GameModel. If we'd like to change the win conditions, all win conditions are isolated in getWinnerId() and isGameOver() methods. If we want to change movement or battle mechanics, all such rules are contained in moveLink(), and download mechanics are also centralized and easily modifiable.

Command System Extensibility: Adding new commands is also simple as all commands are managed in GameController::play(). Add command parsing in the function for the new command, and implement handler method following the established handle*() pattern. Also update help text and error messages accordingly.

Performance Scalability The design includes several features supporting performance scalability:

Efficient Graphics Updates: Only changed cells are redrawn rather than full board refreshes using changedCells tracking. Double buffering eliminates flickering during updates and provides smooth transitions. Pixmap caching reduces redundant drawing operations for static elements.

Memory Efficiency: Smart pointers eliminate memory leaks and reduce memory fragmentation. STL containers provide optimized memory management with minimal overhead. Minimal object copying through move semantics and reference parameters.

Command Processing Optimization: String parsing is performed once per command with results cached in local variables. Error checking occurs early to avoid expensive operations on invalid input. Command validation is layered to fail fast on obvious errors.

Multi-Player Extensions While current implementation supports two players, architecture naturally extends to additional players. Player Management uses std::vector<std::unique_ptr<Player>> container that can hold any number of players, and turn management system (currentTurn % players.size()) automatically handles rotation for any number of participants. Display Scaling can extend a perspective-based display system to support multiple simultaneous views through the existing viewerId mechanism. Board Layout Adaptations can accommodate different board layouts through configuration changes rather than architectural modifications.

Answers to Questions

Q1: Multiple Display Perspectives

In our original plan, we proposed using the Observer pattern with multiple TextDisplay and GraphicsDisplay instances, each configured with a specific player ID to show the game from that player's point of view. This setup allows each display to render the board and player information according to the perspective of the assigned player, filtering information based on ownership, revealed status, and player knowledge.

To adapt our code for two displays, one for player 1 and one for player 2, we would continue to use this Observer pattern by instantiating one TextDisplay and/or GraphicsDisplay for each player and register them as observers. Each display would be attached to the GameModel and receive ChangeEvent notifications whenever the game state changes. This ensures both displays stay synchronized automatically. All necessary player information is contained within the Player class and the display logic remains unchanged.

The board itself would remain stored in a global compass orientation. To present the board from each player's perspective, we would implement a coordinate mapping function that translates global board coordinates to the local perspective of each player. For example, player 1 would see the default orientation, while player 2’s display would flip the board horizontally and vertically so their pieces always appear closest to them.

For movement commands, we would introduce a direction mapping layer. This layer would translate player-relative directions, such as "up" as seen by player 2, into the corresponding global directions used by the game logic. This ensures that commands always feel natural from each player's perspective, regardless of the underlying board orientation. Our existing moveLink function has a pointer to the Player responsible for moving their link (default is the current player, but differs when the hijack ability is used). When a move or ability is performed, the game logic operates on the global board, and all displays are notified. Each display then updates itself by converting the changed cells’ coordinates to its own perspective. This approach ensures minimal redraw and consistent synchronization.

This design is flexible and can easily be extended to more players or additional player information fields. For example, with four players on a plus-shaped board, each display would have its own orientation and direction mapping, but the core logic and observer notification system would remain unchanged.

In summary, our original plan already supports multiple displays with different player perspectives. By implementing coordinate and direction mapping functions, we can provide each player with an intuitive and personalized view of the game while maintaining a single, consistent game state internally.

Q2: Ability Framework Design

Original Plan: We proposed an Ability interface with an execute() method that takes a vector<string> args parameter, allowing each ability to parse its own arguments and implement its unique mechanics. By using polymorphism and following the Strategy design pattern, each ability is encapsulated as a separate class that inherits from the Ability interface. This makes it easy to add new abilities in the future, since new mechanics can be introduced simply by creating a new subclass and implementing its execute() method.

Final Implementation: We closely followed the planned design with a few enhancements, like adding a protected markUsed() method to manage usage.

With this framework, we also implemented three new abilities:

1. Exchange: Swaps positions of two player-owned links. Arguments: Two link IDs (e.g., "a b"). Complexity: Validates both links exist, belong to the current player, are on board, and handles firewall interactions after swap. Innovation: Includes comprehensive firewall effect checking through checkFirewallAfterPlacement() helper
2. Hijack: Moves opponent's link in specified direction. Arguments: Opponent link ID and direction (e.g., "D up"). Complexity: Validates target is opponent's link, respects boosted movement (2 spaces), handles all movement edge cases. Innovation: Demonstrates opponent control mechanics and link boost interaction while maintaining movement rule consistency.
3. GoLater: Allows using two abilities on the next turn instead of one. Arguments: None required. Complexity: Integrates with the turn management system through activateGoLater() and player state tracking. Innovation: Shows deferred effect abilities and state coordination across turns.

Framework Enhancements Beyond Original Plan: Added argument formatting system through formatAbilityArgs() for user-friendly feedback. Implemented comprehensive turn-based ability usage limits with GoLater integration affecting abilitiesAllowedThisTurn. Created helper methods for common operations (firewall checking, board searches, link validation). Added comprehensive error messages specific to each ability's requirements and failure modes. After an ability is used, a notifyObservers(ChangeEvent::AbilityUsed) is called to update graphics and textdisplay on any changes. GameModel::changedCells tells graphics which cells are updated for minimal redraw. Player information is updated after an ability use to show the decrease in remaining abilities and any new knowledge (ex. reveals).

Q3: Extend to 4 players

To support a four-player mode with a plus-shaped board, our current codebase is already well-positioned for this extension due to its modular design. The use of a std::vector<std::unique_ptr<Player>> for storing players in GameModel means we can easily accommodate any number of players. Our existing turn management increments currentTurn and uses modulo with the vector's size, which would naturally cycle through any number of players, not just two. To implement four-player mode, we would need to abstract the board into a base class and create a PlusBoard subclass to handle the unique geometry and boundary logic of the plus shape. Player elimination (removing their links/firewalls and converting their server ports to normal cells) would require some additional logic, but the core player and turn management would remain unchanged. A boolean isEliminated can be added to Player to check if their turn is skipped. Display classes would need to scale for the larger board and support four perspectives, but the Observer pattern and MVC separation make this feasible. Victory conditions can be left unchanged, as we already check that a winner must have 4 downloaded data, or have every opponent download 4 viruses. Overall, while some new code would be needed for board geometry and elimination, our vector-based player management and turn cycling logic already supports 2+ players, so the extension would not require major changes to the core game loop or player handling.

Extra Credit Features

1. Complete Memory Management via Smart Pointers

We aimed to eliminate all explicit memory management and handle them via vectors and smart pointers. To do this, we handled all owns-a relationships through smart pointers:

1. GameModel owns Players, so we used std::vector<std::unique_ptr<Player>>
2. Player owns links, so we used std::map<char, std::unique_ptr<Link>>
3. Player also owns Abilities, so we used std::vector<std::unique_ptr<Ability>>
4. Graphics Display owns XWindow, so we used std::unique_ptr<Xwindow>

This way we no longer had to manually delete in our destructors, and we could use std::move() for ownership transfer. Non-owning references still used raw pointers, and the codebase contains zero delete statements and no memory leaks which we verified through valgrind testing.

2. Game Replay System (Command History)

Challenge: Create a comprehensive command recording and replay system that captures only valid commands and enables complete game recreation.

Implementation Architecture: Command filtering records only successful commands in commandHistory vector after validation and execution. Nested sequence support through handleSequence() method supports recursive sequence file execution with proper error handling. Error resilience ensures invalid commands in sequence files are logged but don't terminate execution.

Advanced Features: Quit integration automatically saves command history when user types quit or game ends naturally. Both interactive and sequence file commands are traced for complete coverage. Validating the replay system allows generated files to be used immediately with sequence commands in future games for verification and testing.

User Benefits: Game sessions can be completely recreated for debugging complex interactions. Interesting games can be shared and replayed for analysis or demonstration. Automated testing through recorded game scenarios ensures regression testing capability. Educational value for analyzing game strategies and ability interactions.

3. Enhanced Graphics with Perspective Management

Challenge: Create a pretty graphics system supporting real-time perspective switching with double-buffering for smooth updates.

Technical Implementation: Double Buffering uses two complete pixmaps (buffer1, buffer2) maintaining full game state for each player perspective. Selective Updates redraws only changed cells using changedCells tracking, reducing computational overhead significantly. Dynamic Perspective allows graphics to automatically switch between player views during turn transitions through setViewerId(). Information Filtering ensures each buffer shows appropriate information based on player knowledge, ownership, and revealed status.

Performance Optimizations: Minimal Redraws through changed cell tracking reduces unnecessary graphics operations from O(64) to O(changed_cells). Buffer caching draws grid and static elements once during initialization and reuses them. Efficient Blitting provides complete buffer swaps for instant perspective changes without redraw delays.

Final Questions

1. “What lessons did this project teach you about developing software in teams?”

Here’s what RAIInet taught us about developing software as a team:

Planning stage is goated: Spending time on creating an effective UML and debating architectural decisions paid off in the development stage. The choice to use MVC with an Observer pattern created a solid foundation that supported all subsequent feature additions without major refactoring. This reinforced that upfront design time is well worth it in large projects.

Interface Designs are goated: Well-designed interfaces (like Ability base class) made the system highly extensible. The ability to add three new abilities without modifying existing code demonstrated the power of good abstraction design and proper separation of concerns.

Observer Pattern is goated: The observer pattern exceeded expectations in providing clean separation between game logic and presentation. Supporting both text and graphics displays simultaneously with no code duplication demonstrated the pattern's effectiveness for complex systems.

Git is goated: Using git for version control helped us work simultaneously on the project, keeping a log of our changes and git blame told us who we needed to consult if we didn’t understand a function someone else wrote, and it helped us debug when certain commits would break things!

2. What Would Be Done Differently:

Earlier Graphics Planning: Graphics implementation became more complex than initially anticipated due to perspective management requirements. Starting with a more detailed graphics architecture plan would have streamlined development.

More Comprehensive Testing Framework: Our sequence files supported integration testing well, but a formal unit testing framework would’ve caught bugs earlier and improved class-level reliability.

Documentation: More consistent inline comments and recording down our design decisions during development would’ve made the final documentation process smoother and more comprehensive.