RAIInet

CS247 Project - Spring 2025

Aayush Patel (21073507) and Arjun Sehgal (21096728)

Overview

Game Board System

The board is represented as a 2D grid using `std::vector<std::vector<std::unique_ptr<Tile>>>`, which provides us with automatic memory management and efficient accesses.

The board features:

- An 8x8 grid with specialized tiles (basic, firewall, and server port)
- Link placement during initialization (randomised or through user-inputted file)
- Integration with Subject-Observer pattern for display updates

We used the observer pattern: Controller inherits from Subject to notify all attached views when the game state changes.

Link System and Enhancements

Abstract Link interface with BasicLink concrete implementation, which provides the fundamental link properties (strength, type, movement distance).

We implemented the decorator pattern as link modifications are implemented through LinkDecorator and concrete decorators:

- `BoostedLink` - increases travel distance
- `WeakenedLink` - decreases strength
- `KnightedLink` - enables knight-like movement

When abilities modify links, the Player class (src/player.cc methods `boostLink()`, `weakenLink()`, and `knightLink()`) wrap existing links with decorators while maintaining tile occupancy relationships, allowing multiple enhancements to stack.

Player Management

Each player manages multiple collections using smart pointers:

- `std::map<std::string, std::unique_ptr<Link>> links` - active links on board
- `std::map<std::string, std::unique_ptr<Link>> downloadedLinks` - captured opponent links
- `std::vector<std::unique_ptr<Ability>> chosenAbilities` - player's available abilities

- `std::map<Player*, std::map<std::string, Link*>> knownOpponentLinks` - revealed opponent links

The key features are:
- Automatic memory management
- Safe ownership transfer during download/upload operations using std::move
- Game state tracking (download counts, win/loss status)

Ability System Architecture

We used the Interface Segregation Principle: abilities are organized into specialized interfaces based on parameter requirements:
- `LinkAbility` - operates on single links (Knightify, Polarise)
- `LinkPlayerAbility` - operates on a link and a player (Download, Scan, Weakenify, LinkBoost)
- `TilePlayerAbility` - operates on tiles (Firewall)
- `LinkLinkAbility` - operates on two links (Tradeify)
- `LinkPlayerTileAbility` - more complicated interaction which operates on all three (Uploadify)

`Controller::useAbility()` methods use dynamic casting to safely execute abilities while maintaining type safety, and new abilities can be added without modifying existing controller logic.

Display System (View Architecture)

We implemented this using the Observer Pattern: multiple simultaneous displays (text and graphic displays) automatically update when game state changes.

Implementation Overview:
- Abstract Subject class - manages observer list and provides notification infrastructure
- Controller inherits from Subject - acts as the concrete subject that notifies observers when game state changes
- Abstract View interface - defines the observer contract with contextual notification types
- TextDisplay - ASCII-based game display that implements View
- GraphicDisplay - X11-based graphical interface that implements View

Key Features:
- Each Player has their own TextDisplay which tracks their perspective, appropriately hiding/showing Links based on what has been revealed to that Player
- Contextual Notifications: Controller sends specific notification types (MOVE_MADE, ABILITY_USED, TURN_SWITCHED, etc.) allowing Views to respond appropriately
- Smart Updates: GraphicDisplay can choose between partial updates (for moves) or full redraws based on notification type
- Views operate independently - graphics can be enabled/disabled without affecting text displays
- Loose Coupling: Controller doesn't own the displays; main.cc manages their lifetime while Controller notifies them through the observer pattern

This design promotes separation of concerns, where the Controller focuses on game logic and Views handle their own display responsibilities.

Game Control and Logic

We have a central Controller class, which manages all game interactions:

- Turn management and game state validation
- Move execution with rule enforcement (bounds checking, collision detection, battle resolution)
- Ability activation with parameter validation
- Win condition monitoring
- View coordination and updates

The important methods which Controller uses are**:**

- `makeMove()` - overloaded to handle both regular and double-direction movement (for knights)
- `battle()` - handle battles between links, determining the winner based on strength and initiator
- `useAbility()` - polymorphic ability execution (5 overloaded versions for different ability types)
- `executeCommand()` - command parsing, validation, and execution

Memory Management and RAII Implementation

We used smart pointers to eliminate manual memory management:

- `std::unique_ptr` for ownership (links, abilities, tiles, board)
- `std::move` semantics for safe ownership transfer
- Automatic cleanup on exceptions or normal termination
- No raw pointers for owned resources

We follow RAII principles, ensuring automatic cleanup and preventing memory leaks.

Game Initialisation and Configuration

We have implementations for the following key functionalities:

- Command-line argument parsing for graphics mode
- Support for predetermined link and ability arrangements
- Flexible player setup with configurable ability assignments

To initialize, we follow this flow:

1. Parse command-line arguments
2. Create players with the specified ability arrangements, or with the default selection
3. Initialize board with link placement (random or predetermined)
4. Set up displays (text + graphics, if enabled)
5. Begin the main game loop through the Controller

High Cohesion and Low Coupling

High Cohesion:

- Player class: all methods relate to player state management
- Individual ability classes: each focuses on single ability implementation
- Board class: all functionality relates to grid management

Low Coupling:

- The display system uses the observer pattern for loose coupling
- Abilities are independent of each other
- Components interact through Controller mediation rather than direct coupling

Design

Model-View-Controller Architecture

We separated class responsibilities using the MVC pattern with the Controller orchestrating game logic, the Board and related classes representing the model, and the View handling display.

This separation allows for multiple display modes (text and graphics) without affecting game logic. The Controller manages all game state transitions and rule enforcement, while the Views (TextDisplay and GraphicDisplay) handle the outputting concerns independently.

Observer Pattern

The Controller class inherits from Subject (include/subject.h), and View classes inherit from the abstract View interface (include/view.h), creating a publisher-subscriber relationship for display updates.

This design ensures all active displays automatically refresh when the game state changes, maintaining consistency across multiple views without tight coupling between the game logic and presentation layers. The contextual notification system enables optimized graphics updates where only changed elements are redrawn.

Decorator Pattern

Link modifications (boosting, weakening, and knighting) are implemented using the Decorator pattern through LinkDecorator and its concrete implementations (BoostedLink, WeakenedLink, and KnightedLink).

This allows dynamic modification of link properties without changing the original link objects. The pattern allows us to stack multiple effects on a Link, and maintain the original link's interface while extending functionality.

Interface Segregation Principle (Abilities)

Abilities are implemented using the Interface Segregation Principle through multiple interfaces (in include/ability.h): `LinkAbility`, `LinkPlayerAbility`, `TilePlayerAbility`, `LinkLinkAbility`, and `LinkPlayerTileAbility`, grouped by the parameter requirements that the abilities need.

This design follows the Interface Segregation Principle by ensuring that clients (Controller methods) only depend on the interfaces they actually need. Rather than having one large Ability interface with many unused methods with empty implementations, abilities are segregated into focused interfaces based on their parameter signatures.

Smart Pointer Resource Management

We use smart pointers throughout the system for automatic memory management, which we'll discuss further in the Extra Credit Features section.

SOLID Design Principles

Single Responsibility Principle: Each class has a clearly defined, single responsibility. For example, the Board class handles only grid management, Player class manages only player state, and individual ability classes each handle one specific ability type.

Open/Closed Principle: The system is open for extension but closed for modification. New link types can be added via the Link interface, new abilities via the ability interfaces, and new views via the View interface, all without modifying existing code.

Liskov Substitution Principle: All derived classes can be substituted for their base classes. Any LinkDecorator can substitute for a Link, any concrete ability can substitute for its interface, and any View implementation works wherever View is expected.

Interface Segregation Principle: As discussed above, the ability system exemplifies this principle by providing multiple focused interfaces rather than one large interface.

Dependency Inversion Principle: High-level modules (Controller) depend on abstractions (View, Link, Ability interfaces) rather than concrete implementations, allowing for flexible substitution of implementations.

How the final design differs from the plan

Once we began writing code and tests, several practical issues forced us to change course.

Change of publisher: During our coding process, we saw that only the Controller knows when a turn ends, when a battle is complete, and when an ability is performed. Letting the board call notifyObservers in the middle of those sequences resulted in out-of-order turns and, in graphics mode, flickering while battles were still being resolved. We therefore moved the Subject from the Board to the Controller. The board now stays focused on state, while the controller decides the exact moment to broadcast one of five NotificationType values (`MOVE_MADE`, `BATTLE_OVER`, `ABILITY_USED`, `TURN_SWITCHED`, `FULL_REDRAW`).

Multiple ability interfaces: In our initial plan, all abilities would override two methods (`isValid()`, `apply()`). When we began to type the signatures a single interface forced many abilities to accept parameters they never use, which broke type safety. We split the contract into the five small interfaces mentioned above. We modified `Controller::useAbility()` to be overloaded instead, and use dynamic_cast to pick the right interface.

Extra link data: We added id, owner, and back-pointers to Tile so that displays could label pieces without searching the whole board and so that the controller could resolve battles quickly.

Smart pointers: While coding, we used `unique_ptr` for every owning relationship and raw pointers only for non-owning references, as specified in the extra credit challenge. This choice caused us to add more move-aware helpers (`Player::download()`, `Player::upload()`) that were not in the initial plan.

Display callbacks: We originally planned for both TextDisplay and GraphicDisplay to inherit from View and accept `notify(row, col, change)`. However, after moving notification responsibilities to Controller, we needed two kinds of updates: fast cell-level updates as well as full board refreshes. Therefore, View now keeps both `notify(row, col, change)` as well as a `notifyFull()` function for full board refreshes.

Resilience to Change

Views and Observer Integration

Because the Controller communicates only through the abstract View interface, any new display technology can subscribe to game events without altering game logic. Adding, for example, a web-based view or a VR overlay only requires implementing the View interface and registering it as an observer. No changes to controller.cc are required. The contextual NotificationType system allows new Views to respond dynamically to different game events (moves, abilities, turn changes) for optimized rendering. For example, we implemented the graphics display with minimal changes to existing game logic using this pattern.

Link Enhancements via Decorator

Link modifications are layered through LinkDecorator, allowing multiple effects to stack in arbitrary order through decorator chaining. New mechanics, such as a PoisonedLink that drains strength over time, can easily be implemented as another decorator class without having to modify the existing Link and LinkDecorator code.

Ability System with Interface Segregation

Each ability type conforms to a focused interface that matches its parameter requirements. This allows new abilities to be added by simply implementing the appropriate interface. For example, creating a `TileTileAbility` would be very simple and wouldn't modify existing ability classes or the controller's logic. The `Controller::useAbility()` methods use dynamic casting to safely and flexibly dispatch to the correct implementation, maintaining type safety while also supporting extension.

Domain Expansion under Open/Closed

New link or tile varieties inherit from their respective base classes, and the Controller enforces rules that apply to the abstractions, not the concrete types. Introducing a PortalTile that teleports links therefore affects only its own class and the specific rule logic added to the controller.

Summary

Collectively, these choices keep the codebase open for extension and closed for modification: most future work happens behind existing abstractions, preserving stability in the central modules while letting the game be grown quickly.

Answers to Questions

In this project, we ask you to implement a single display that maintains the same point of view as turns switch between player 1 and player 2. How would you change your code to instead have two displays, one of which is from player 1's point of view, and the other of which is player 2's point of view?

What Actually Happened: Our original plan to create two displays with constant viewerSide fields worked as expected, but we discovered a significant issue that caused us to rethink the observer pattern entirely. We initially planned to have Board inherit from Subject and notify all displays when state changed, but we realized this violated separation of concerns since Board was managing game state while Controller was orchestrating game logic and deciding when notifications should occur.

The solution required making Controller inherit from Subject instead of Board, which fundamentally changed our observer architecture. Rather than Board automatically notifying displays whenever its state changed, Controller now explicitly sends contextual notifications at appropriate moments in the game flow. This meant Controller became the central source of both game logic and display updates, which actually improved the design by giving us more control over when and how displays refresh.

We also discovered that our TextDisplay instances needed better perspective handling than anticipated. Each TextDisplay was constructed with a different player perspective as planned, but the notification logic became more complex since displays now only update when their perspective matches the current turn. Additionally, we found that GraphicDisplay could leverage contextual notifications to optimize rendering, using partial updates for most game events rather than full screen redraws, which significantly improved graphics performance beyond what our original plan would have achieved.

How can you design a general framework that makes adding abilities easy? To demonstrate your thought process for this question, you are required to add in three new abilities to your final game. One of these may be similar to one of the already-present abilities, and the other two should introduce some novel, challenging mechanic. You are required to actually implement these.

Our plan to approach implementing abilities by creating individual subclasses proved successful and maintainable. Each ability does live in its own subclass implementing `isValid()` and `apply()` methods. However, we didn't anticipate the amount of complexity that would arise from needing multiple ability interfaces based on parameter signatures. We ended up implementing five different ability interface types (`LinkAbility`, `LinkPlayerAbility`, `TilePlayerAbility`, `LinkLinkAbility`, and `LinkPlayerTileAbility`) rather than just one base Ability class, because we realized that type safety required segregating abilities by their parameter requirements.

The `Controller::useAbility()` method ended up requiring five different overloaded versions (src/controller.cc, lines 300-380) to handle the different ability signatures safely, which was more complex than we had anticipated while planning.

For our implemented abilities, Knightify worked exactly as planned, but we actually implemented WeakenedLink through the decorator pattern rather than direct strength reduction, which proved more flexible for handling multiple effects. Uploadify required more complex tile validation logic than expected, as we had to ensure uploaded links were placed on valid, unoccupied tiles. We also renamed Swapify to Tradeify, as we forgot to account for the fact that the first character of each ability should be unique.

One could conceivably extend the game of RAIlnet to be a four-player game by making the board a "plus" (+) shape (formed by the union of two 10x8 rectangles) and allowing links to escape off the edge belonging to the opponent directly adjacent to them. Upon being eliminated by downloading four viruses, all links and firewall controlled by that player would be removed, and their server ports would become normal squares. What changes could you make in your code to handle this change to the game? Would it be possible to have this mode in addition to the two-player mode with minimal additional work?

While we designed our system with extensibility in mind, we realized during implementation that a four-player extension would require more changes than originally anticipated. Our current Player class does track opponents, but we actually store knownOpponentLinks as a map keyed by Player pointers, which works well for multiple opponents. However, we didn't initially consider how the turn management would need to handle player elimination mid-game. We implemented the end of a game as "all other players have lost", rather than the game ending when the first player loses (ie, we have one winner, instead of just one loser).

The `Board::grid` structure would indeed support a configurable plus shape with inactive tiles, but we realized we'd need more sophisticated edge detection logic in our move validation methods. The current movement validation in `Controller::isValidMove()` checks bounds against fixed height and width values, which would need to become more flexible to handle irregular board shapes.

We also discovered that the display formatting would require significant changes to handle non-rectangular grids properly. `The TextDisplay::print()` method assumes an 8x8 rectangular layout, and adapting it to render a plus-shaped board with proper spacing and alignment would be more challenging than we initially expected.

Extra Credit Features

Smart Pointers Memory Management Challenge

We successfully completed this challenge throughout our entire RAIInet implementation. Our codebase demonstrates RAII principles without explicitly managing memory (no delete statements), without leaking any memory.

Examples of Smart Pointer Usage:

Board Class:

```
std::vector<std::vector<std::unique_ptr<Tile>>> grid;
```

The entire game board uses smart pointers for automatic tile management. When the board is destroyed (ie, when the game is over), all tiles are automatically freed and cleaned up without any explicit delete calls.

Player Class:

```
std::map<std::string, std::unique_ptr<Link>> links;
std::map<std::string, std::unique_ptr<Link>> downloadedLinks;
std::vector<std::unique_ptr<Ability>> chosenAbilities;
```

All player-owned resources use unique_ptr for automatic cleanup. Link ownership transfers are handled safely by using `std::move`.

Controller Class:

```
std::unique_ptr<Board> board;
```

Main Class:

```
auto p1 = make_unique<Player>(1);
auto p2 = make_unique<Player>(2);
unique_ptr<Board> board(new Board());
auto textDisplay1 = make_unique<TextDisplay>(...);
auto textDisplay2 = make_unique<TextDisplay>(...);
```

Safe Ownership Transfer:

Download/Upload Operations: our link download and upload system demonstrates safe ownership transfer without manual memory management:

```
// From download.cc
auto downloadedLink = currentOwner->download(&l);
p.storeDownloadedLink(move(downloadedLink));
```

```
// From uploadify.cc
auto linkPtr = move(it->second);
downloadedLinks.erase(it);
p.upload(move(linkPtr), &location);
```

These operations transfer ownership safely using `std::move`, which ensures that there aren't memory leaks or double-deletions.

Raw Pointers (Non-Owning Only):

Our limited raw pointer usage follows the challenge's requirements: they express relationships, not ownership.

- **Observer Pattern:** `std::vector<View*>` observers in Subject class - non-owning references for notification
- `Player* owner` in Link classes: non-owning back-references
- `Link* occupant` in Tile class: non-owning reference to the current occupant (a Link)
- `Board* board`, `Player* perspective` in display classes: non-owning references

Knightify Ability Implementation

We built the Knightify ability, which was our fourth additional ability. Knightify presented unique challenges in input parsing and movement calculation, as knights require two-direction inputs for their movement pattern.

We implemented this ability by adding another decorator to a Link, KnightedLink, which indicates if that Link is a knight. We then had overloaded functions to validate, calculate, and execute a move for knights:

```
std::pair<int, int> calculateMove(Link* l, std::string direction1, std::string
direction2);

bool makeMove(Link& l, const std::string directionFirst, const std::string
directionSecond, Player& p);

bool isValidMove(Link* l, const std::string& directionFirst, const std::string&
directionSecond);
```

These overloaded methods specifically handle the knight's two-step movement pattern: two steps in one cardinal direction, followed by one step in a perpendicular direction.

We also decided that a knightedLink that has also been decorated with the boost link ability, can move 4 steps in one direction, followed by 2 steps in a perpendicular direction. In essence, we just double the step distance of a link for the first direction. Note that a link may be boosted twice as well as knighted.

Final Questions

What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Working together on this project taught us several crucial lessons about collaborative software development. Communication is absolutely critical - we learned the hard way that assumptions about how components should interact often differed between us. For example, when one person implemented the ability system while the other worked on the controller, we discovered our interfaces didn't match our mutual expectations, leading to some refactoring. We realized that frequent check-ins and explicit discussion of design decisions would prevent significant integration problems later, which is even more important in a large-scale project such as this one.

Version control and code organization became essential skills we had to develop quickly. Initially, we struggled with merge conflicts and inconsistent coding styles, but we eventually established clear conventions for file organization, naming, and commit messages. We learned to work in smaller, focused commits and communicate about who was working on which components to avoid stepping on each other's code.

The importance of modular design became apparent when working in parallel. Our initial attempts to work on the same files simultaneously created chaos, but once we established clear module boundaries (one person handling game logic while the other focused on the displays) our productivity increased dramatically. This taught us that good architecture isn't just about clean code, but about enabling effective teamwork.

What would you have done differently if you had the chance to start over?

We'd spend much more time on interface design before implementation. While working on DD2, we often immediately settled on the first design decision that came to mind. However, as we implemented the project, there were a lot of instances where we realised we had made a choice which would cause a lot of extra complexity, simply because we hadn't fully thought through the option.

Additionally, we ideally should have adopted test-driven development from the start. We implemented large chunks of the program while assuming that the logic would hold, but as we began testing later on, we realised we had missed a lot of edge cases.

Another major aspect would be implementing incremental development with regular integration testing. We often worked on separate components in isolation, such as one person handling abilities while another worked on the display system, only to discover integration issues when combining our code. This led to significant refactoring sessions that could have been avoided with more frequent integration checkpoints.

We also underestimated the complexity of memory management and ownership semantics with smart pointers, which caused several issues which were difficult to debug late in development. Starting with simpler ownership models and gradually introducing complexity would have been more sustainable.