CS 247 Project: RAIInet - Project Plan & Task Breakdown

Balaji Leninrajan, Kayla Chang, Tyler Chen

Implementation Phases

We will build the project in logical, testable phases. This ensures we always have a working program, as recommended by the spec.

- **Phase 1: Core Foundation & Build System**
    - **Goal**: Create all class files (`.h`/`.cpp`) with basic definitions and method stubs.
    - **Tasks**:
        1. Set up the Makefile (Tyler)
        2. Create all header files with class declarations. (Balaji)
        3. Create all `.cpp` files with empty method implementations. (Tyler)
        4. Ensure the project compiles into a runnable program that does nothing. (Kayla)
- **Phase 2: Text-Based Game Logic (The MVP)**
    - **Goal**: Implement a fully playable game with only the text interface.
    - **Tasks**:
        1. Implement the `Board` and `Cell` classes. (Kayla)
        2. Implement the `TextView` to draw an empty board.
        3. Implement the `Factories`. (Balaji)
        4. Implement the `Player` and `Game` classes to manage state. (Tyler)
        5. Implement the `Controller` to parse `move`, `board`, and `quit` commands.

6. Implement movement and battle logic (strength comparison, capturing). (Tyler)
7. Implement win/loss conditions. (Balaji)

- **Phase 3: Adding Abilities**
  - **Goal**: Integrate the ability system.
  - **Tasks**:
    1. Implement the `Ability` Strategy interface and `AbilityFactory`. (Balaji)
    2. Implement each of the five required abilities as a concrete strategy class. (Combined efforts)
    3. Implement the `LinkBoostDecorator`. (Tyler)
    4. Update the `Controller` to handle the `abilities` and `ability <N> <params>` commands. (Kayla)
    5. Design and implement the three new custom abilities. (Combined efforts)
- **Phase 4: The Graphical View**
  - **Goal**: Add the graphical user interface.
  - **Tasks**:
    1. Choose a simple graphics library (e.g., X11).
    2. Implement the `GraphicalView` class as another Observer. (Tyler)
    3. Implement drawing logic for the board, links (with colors for type), and game status info. (Kayla)
    4. Handle the `-graphics` command-line flag. (Balaji)
- **Phase 5: Finalization & Polish**
  - **Goal**: Final bug fixes, documentation, and ensuring all spec requirements are met.
  - **Tasks**:

1. Thoroughly test all commands and interactions. (Combined efforts)
2. Implement robust error handling for bad commands. (Tyler)
3. Implement the `sequence <file>` command. (Balaji)
4. Write internal code documentation and review the project structure. (Combined efforts)

Risk Management & Priorities

Following the spec's advice, our priority is to always have a working, submittable program.

- **Top Priority (MVP)**: Completing **Phase 2**. A program that allows two players to move pieces and battle in a text-only interface is a passing submission.
- **Secondary Priority**: Completing **Phase 3**. Adding the full ability system is the next most important feature set.
- **Final Priority**: Completing **Phase 4**. The graphical interface will be the last major component we add.

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?

We chose to maintain a design where the Controller has a vector of vector of views, and it notifies every single view with a player-dependent update. This means that every single view can handle the update independently, which allows us to either create a view class for each player, or create a view class that stores multiple sets of data to display for each player. When display is called, we will call the Game getPlayerIndex() to determine what player's view to display, and call the display functions on all the related views.

This approach allows us to create any number of views; We can create both a text view and a graphics view for each player, or a view that can handle more than one player; In this case, the view pointer would be passed into the vector that maps player indices to views multiple times.

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.***

To design a general framework that makes adding abilities easy, we can use the factory method and have an AbilityFactory class to handle adding abilities for players. We have brainstormed three unique abilities to our final game. First, we have the lag ability. When a player activates it, opponents will see one move behind the player for three rounds. Next, we have the quantum entanglement ability, which allows two of the player's links to be entangled together. This means if one link moves in a certain direction, so does the other. This also applies to downloading, so if a player downloads one link, they download both links. Most importantly, we have the Papple ability. This is a strategic move in which a player creates a battle formation to attack and defend at the same time (there must be one link on each corner of the board, all owned by the player). On the next round, when calling the Papple ability, the player wins the game.

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 10×8 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?

The game class will create and maintain a vector of player pointers, switching between them between each turn. Thus the turn-taking mechanisms should be extensible for 4 players.

Interactables that are owned by a player do not depend on anything except their owner and the player they're interacting with for their logic. This means any game piece interactions should not rely on the existence of exactly two players and should be extensible to 4.

Displays for 4 players are also handleable as previously stated.