Students of Watan

Michael Liu, Owen Tan

Introduction

Inspired by Settlers of Catan, this project brings the board game **Students of Watan** to life. It supports four players as they compete to collect resources and advance their academic standing through assignments, midterms, and exams. The game features fair and loaded dice mechanics, resource trading, and real-time tracking of victory conditions as players race to achieve 10 course criteria.

Overview

High-Level Structure

The design separates the application into distinct modules that follow a Model-View-Controller (MVC) principle:

- Model: The core state is held by the Board, Tile, Criterion, Goal, and Player modules. These classes manage data integrity, adjacency maps, player inventory, and resource generation logic.
- Controller: The Game module orchestrates the entire session, managing turns, checking victory conditions, and handling rule enforcement. The CommandExecutor module serves as the primary input controller, dispatching user commands to the Game state.
- View: Output is generated through operator overloading (operator<<) defined on the Board, decoupling the display (the ASCII art) from the underlying game state.

Design

Techniques Used to Mitigate Implementation Risks

In our plan of attack document, we identified several risks that we may encounter during the project. These are the strategies we used to mitigate them.

- Complexity of rules can lead to inconsistent or buggy game state
    - Technique: Template Method Pattern. The Game class defines the strict, invariant sequence of a turn (e.g., must roll before acting, must check win condition before advancing player). This rigid structure prevents skipping critical state checks or executing commands out of sequence.
- Dynamic conditions and status detection may be tricky to maintain

    - Technique: Observer Pattern Intent (Model-View Decoupling). The Board (Model) holds all state data (owners, resources). State checks, like updating the ASCII display, are performed by the observer (operator<< and Board::display) which reads the data directly, ensuring the visual output always reflects the current, accurate state of the Board model.
- Incorrect board parsing or failing to read from file
    - Technique: Factory Method Pattern & Strategy Pattern. The IBoardSetupStrategy interface is used to define how the board is initialized . Subclasses like FileBoardSetupStrategy strictly encapsulate the file-reading logic, keeping parsing errors contained and separate from the core Board structure. The Game logic only relies on the final, valid Board object provided by the chosen strategy.
- Event cards may conflict with turn order or normal resolution
    - Technique: Polymorphism & Strategy Pattern. Event cards inherit from a base EventCard class, defining a common apply(Player &player, Game &game) interface. This polymorphism allows the Game loop to simply call card->apply(...) without knowing the specific card's complex effects (like stealing resources or moving geese), ensuring that the general turn resolution process is not affected by individual card logic.
- Coordination issues if multiple people implement different components
    - Technique: C++20 Modules (Low Coupling) and Shared Type Modules. Splitting the code into .ixx (Interface) and .cc (Implementation) files minimized dependencies. The WatanTypes module centralized common definitions (ResourceType, PlayerColour) , eliminating the risk of circular dependencies between highly interconnected components (like Player and Criterion).

Design Patterns

We leveraged established design patterns to ensure our architecture is scalable, modular, and allows for runtime variation

- Factory Method
    - This pattern is clearly demonstrated by the dice component. We defined the generic IDiceStrategy interface, allowing the Game class to ask for a dice roll without knowing its specific implementation (e.g., Fair vs Loaded).
- Template Method
    - The Game::nextTurn() method acts as the template method, enforcing the invariant steps: checking the win condition, managing turn state flags (mustRoll, hasRolled), and advancing the currentPlayer. Similarly, the handleInitialCriterionPlacement method enforces the precise "snake order" (Blue

-> Red -> Orange -> Yellow -> Blue, etc.) and rules specific to that phase, defining the strict skeleton of the initial setup.

Updated UML

See uml-final.pdf
The original UML provided a high-level model, but the final implementation evolved significantly to handle complex state transitions and rule validation required by the game. Key differences involved expanding simple UML methods like void run() into a sequence of state-driven functions (e.g., rollDice, nextTurn) and adding comprehensive state management fields (gamePhase, mustRoll, hasRolled, waitingForGeesePlacement) which were not present in the original design but are crucial for enforcing the turn sequence and player commands. Methods like getCurPlayer() were replaced by more robust, const-correct overloads (getPlayer() const, getPlayer(int index) const) to ensure read/write safety, and numerous private methods (handleGeesePlacement, distributeResources) were added to encapsulate complex, multi-step game logic that was only implied in the original UML.

Resilience to Change

Our design philosophy centered on **low coupling** and **high cohesion**. This approach allows the codebase to adapt to significant specification changes with minimal refactoring.

Low Coupling

- Module-based architecture
    - Each subsystem (Board, Player, Event Cards, etc.) exposes only what is necessary through exported interfaces. Internal details remain hidden, preventing unintended dependencies
- Shared core types centralized in WatanTypes
    - Resource types and player colours are defined once, allowing for easy adding of new resources or player colours compared to more tightly coupled systems (e.g., changing only one file compared to three)
- Factory Method for dice mechanics
    - Game logic depends on an IDiceStrategy interface, not on specific dice implementations, i.e., the system does not need to know how dice work internally
- Event card polymorphism

    - Playing and storing a new card (e.g., LazeezCard inheriting from EventCard) would automatically be supported without requiring changes to the Player class itself

High Cohesion

- Subsystems manage only their own responsibilities
    - Player handles only player state and actions
    - Board handles tile layout, ownership, and resource allocation
    - Dice classes handle randomness only
    - Event cards encapsulate their own behaviour

Combined Impact

By keeping classes focused and reducing interdependence, our code supports the possibility of various changes to the program specification. Various ways our game can change or improve include:

- New gameplay mechanics
    - Adding additional dice types (e.g., 12-face dice) is as easy as creating a new class inheriting from IDiceStrategy without modifying any of the game logic
    - Event cards are separate polymorphic modules, meaning the creation of new ones require no changes to Player or Board
    - Building more tiers above exams (e.g., Internships)
- Gameplay variants
    - Since board generation is isolated, new board classes for alternate boards can be swapped in
    - Changing the theme of the game is as simple as updating the resource names in WatanTypes (Study, Lecture, Lab -> Wheat, Wood, Sheep)

Answers to Questions

1. You have to implement the ability to choose between randomly setting up the resources of the board and reading the resources used from a file at runtime. What design pattern could you use to implement this feature? Did you use this design pattern? Why or why not?

The Factory Method Pattern (in conjunction with the Strategy Pattern) was the ideal choice for implementing variable board setup. This pattern is useful when we want to create different versions of an object based on a policy, such as selecting a board setup based on command-line arguments. We successfully implemented this feature by defining the abstract IBoardSetupStrategy and creating two concrete classes: RandomBoardSetupStrategy (which

generates resources randomly) and FileBoardSetupStrategy (which reads the resource layout from a given file). This makes adding future setup methods simple, as we only need to create a new concrete strategy class.

2. *You must be able to switch between loaded and unloaded dice at run-time. What design pattern could you use to implement this feature? Did you use this design pattern? Why or why not?*

The Factory Method Pattern (and the underlying Strategy Pattern) was the perfect fit for allowing runtime switching between dice mechanics. We implemented this feature by defining the abstract IDiceStrategy interface and creating two concrete subclasses: FairDiceStrategy (which produces a random roll) and LoadedDiceStrategy (which prompts the user for a roll value). The Player class holds a pointer to the IDiceStrategy interface and uses a method (setDiceStrategy) to swap the strategy dynamically at runtime when the load or fair commands are used. The central game logic only ever interacts with the roll() method defined by the abstract interface, ensuring consistency and flexibility.

3. *We have defined the game of Watan to have a specific board layout and size. Suppose we wanted to have different game modes (e.g. square tiles, graphical display, different sized board for a different numbers of players). What design pattern would you consider using for all of these ideas?*

A good design pattern for supporting different game modes is the Template Method pattern. All versions of the game would follow the same general steps (board layout, size), but each mode might do certain steps differently. For example, creating a square board instead of a hex board, using graphical display instead of text, or setting up a bigger board for more players. With the Template Method, we can put the common setup steps in the base class and let each game mode override only the parts that change. This makes it easy to add new game modes later without rewriting the whole setup process.

4. *At the moment, all Watan players are humans. However, it would be nice to be able to allow a human player to quit and be replaced by a computer player. If we wanted to ensure that all player types always followed a legal sequence of actions during their turn, what design pattern would you use? Explain your choice.*

The Template Method pattern could be a good choice to implement this. We can define the sequence of actions that every player must take during a turn (rolling dice, trading, building) in a base player class. Then, both human players and computer players can follow that sequence,

with just the difference that human players choose actions manually, while computer players decide actions automatically. This ensures that no matter what type of player is in the game, the turn always follows the correct rules.

5. *What design pattern would you use to allow the dynamic change of computer players, so that your game could support computer players that used different strategies, and were progressively more advanced/smarter/aggressive?*

To let computer players use different strategies, we can use the Strategy pattern. This pattern lets you define different behaviours (like aggressive, defensive, or random) in separate classes. Each computer player has a strategy object that determines how it acts during its turn. The player itself does not need to know the details of the strategy, so we can easily change the strategy at runtime, even while the game is running. This makes it simple to create computer players that evolve or become more advanced over time, or to swap out strategies for testing or different difficulty levels. By keeping the strategies separate from the player logic, the code is easier to maintain, extend, and customize without affecting the rest of the game.

Extra Credit Features

Players

- On the second run-through (Yellow -> Orange -> Red -> Blue), each player will be able to pick up 1 of each resource adjacent to their second settlement for their starting hand
- On the first and second run-throughs, each player will also be able to place down one goal achievement directly adjacent to their respective criteria
    - We leveraged the Board::tileCoords adjacency map. During the snake-order placement, the Game class uses the newly placed criterion ID to look up all adjacent tiles, determines their resource type, and grants the current player one of each found resource using Player::addResource.

Trading

- Although the original rules only allow one-for-one trading, we decided to extend the system to allow trades involving any number of cards

Development Cards

- General
    - In Watan, we will call these “Event Cards”
    - Can be pulled by using up 1 Lab, 1 Study and 1 Caffeine
    - Player can pull as many event cards as they like as long as they have the materials
    - Event cards will be played immediately after they are drawn
- Events

    - **Gain a study buddy:** When a player pulls a study buddy, they are immediately played and the player must move the geese to another tile, blocking that tile and stealing 1 random resource from a player that fits the criteria laid out
    - **Goal speedrun:** Achieve 2 goals immediately
    - **Year of plenty:** Automatically select 2 resources of their choice
    - **Monopoly:** Request a specific material and every other player must give them all of that material
    - **Course Criteria:** Gain 1 course criteria the moment this card is drawn
- Implementation
    - We defined a base EventCard class with a virtual apply(Player&, Game&) method. Every specific card (MonopolyCard, StudyBuddyCard, etc.) inherits from this base. The Game class simply calls card->apply() on the purchased card, deferring the complex, unique logic to the card's encapsulated implementation.

Largest study group

- If a player gets 3 or more study buddies, they gain the “Largest Study Group” status if they are the first to do so
- If a player already has the status, another player must have strictly more study buddies to take the Largest Study Group status
- Largest Study Group status gives 2 course criteria

Longest Goal Achievement

- If a player gets 5 or more goal achievements in a connected line, they gain the “Longest Goal Achievements” status if they are the first to do so
- If a player already has the status, another player must have strictly more goal achievements in a connected line to take the Longest Goal Achievements status
- Longest Goal Achievements status gives 2 course criteria

Final Questions

1. 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 with a partner on *Students of Watan* taught me that successful team software development depends on clear communication and thoughtful design. We learned (through resolving frustrating merge conflicts) that defining responsibilities early so we could work independently without breaking each other’s code is very important. Regular check-ins and incremental testing helped us catch issues early and avoid integration problems.

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

If we had the chance to start over, we would spend more time upfront planning the system architecture and finalizing interfaces before jumping into implementation. Some parts of our design evolved mid-project, which led to rework and a few integration challenges. Establishing a more detailed UML diagram, clearer communication protocols, and a consistent set of coding conventions from the very beginning would have helped us maintain alignment throughout development. We also would have created small test harnesses earlier, so we could verify components independently before connecting them into the full game. Overall, better initial planning and earlier testing would have made the development process smoother and more efficient.