



students of watan!

Final Design Document

Angela Jin, Yujin Bae, Halle Chan
CS246 Fall 2024

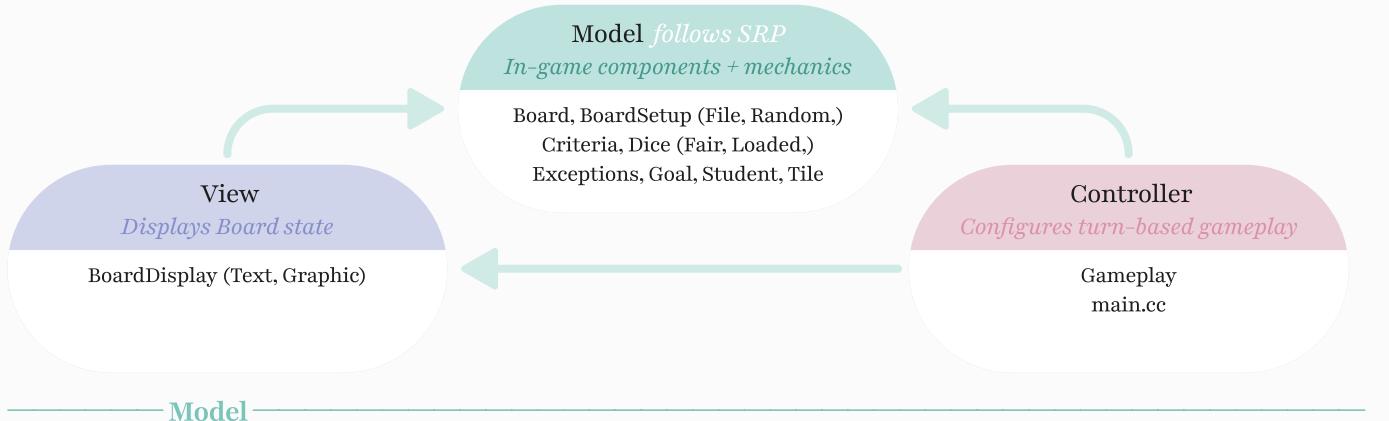
introduction.

We, as in the mutual decision-making of Yujin Bae, Angela Jin, and Halle Chan, present our adaptation of *Students of Watan* as implemented through an object-oriented C++11 structure, for the purpose of CS246's final assignment. *Students of Watan* is an adaptation of board game *Settlers of Catan*, in which users race for ten victory points in a turn-based claiming and upgrading of criterion and goals.

To aid with the explanations in this design document, we have added visualiser formats for some concepts.

overview.

To bring *Students of Watan* to life, we organized a object-oriented design structure that follows an **MVC architecture**:



Board

files: board.h/cc

The Board class serves as the central Model component in the MVC architecture for *Students of Watan*. It manages the key game elements in the context of the game—vectors of Tile, Goal, and Criteria, and an index geesePosition.

It provides functionality for core game actions that alter the attributes it possesses:

1. handling dice rolls to distribute resources (tileRolled);
2. enabling players to purchase goals or criteria (buyGoal, buyCriteria);
3. improving criteria upgrades (improveCriteria);
4. trading resources (trade);
5. setting the geese index (moveGeese).

Board interacts with other game elements to hold up adherence to adjacency rules and updates only to the game state it manages—we consider that in real life, a board directly owns its tiles, goals, and criteria. While one can argue that a tile owns its goals and criteria, we actually realize it does not—goals and criteria may be ‘on’ a tile, but multiple tiles share them as edges and vertices respectively, so we determined that Board as a whole owns them instead.

It utilizes an **Observer design pattern** to notify attached observers (BoardDisplay) of state changes, which upholds that the View remains synchronized with the game’s current state.

BoardSetup

files: boardSetup.h/cc, fileSetup.h/cc, randomSetup.h/cc

The abstract base class BoardSetup class is responsible for initializing the state of Board, with two board setup types. It defines the core steps required to configure the game board: (1) assigning resources and values to tiles, goals, criterion, and (2) ensuring the board adheres to the specified game rules.

RandomSetup and FileSetup implement the BoardSetup interface using the **Template Method pattern** (allowing specific steps to vary.) RandomSetup generates the tile resources and values randomly; FileSetup reads these configurations from a file stream to create specific gameplay scenarios. We encapsulate the board init. logic to ensure that the setup process is reusable + extendable to future features.

Criteria

[files: criteria.h/cc](#)

The Criteria class represents a vertex on a hexagon Tile on Board, where players achieve milestones through completions and upgrades. It tracks integer index on the board, enum completionLevel, adjacent goals, and a pointer to Student owner. In Criteria, we enforce game rules specific to how Criteria can change (i.e. adjacency, improvement conditions, and ownership constraints). Non-constant methods comprise of attaching a new owner to a Criteria instance and improving it, to the tune of these conditions. Constant methods allow basic accessors, checking for conditions of attributes, and looking for the existence and ownership of adjacent Criteria/Goal.

Note that in this context, ‘adjacent’ means that on the display of the board, a Criteria is connected by one edge.

Dice

[files: dice.h/cc, fairDice.h/cc, loadedDice.h/cc](#)

The Dice class represents the dice-rolling mechanism in the game, with two concrete implementations: LoadedDice and FairDice. Using the Factory design pattern, the game instantiates either LoadedDice for player-specified rolls or FairDice for random rolls. Both implement roll w. the intention of consistent interaction for clients.

Exceptions

[files: exceptions.h/cc](#)

Exceptions in the code handle invalid game actions, such as insufficient resources, invalid inputs, or I/O errors, which provide clear error messages so we can be aware of exactly what undefined behaviour happens.

Goal

[files: goal.h/cc](#)

Goal represents an edge on a hexagon Tile on the Board, where players achieve milestones by claiming connections between vertices. It tracks an integer index on the board, a pointer to the Student owner, and adjacent Criteria. In Goal, we enforce game rules specific to how goals can be claimed. Non-constant methods handle claiming ownership of a Goal (playGoal) and attaching adjacent Criteria. Constant methods provide basic accessors + validate conditions of adjacency, & the ownership of a goal or its associated criteria.

Student

[files: student.h/cc](#)

Student is a placeholder ‘inventory’ for a player. It tracks attributes such as Colour for identification, collections of owned Criteria and Goal objects, and an unordered_map for managing resource counts by enum ResourceType. Methods addResource, removeResource, and playCriteria interact w. game elements (but adhering to resource requirements and adjacency constraints.) The designs associates Student objects with their achievements and resources, and includes Student-specific features like calculating victory points (calculatePoints) and exporting the object state (save). This structure enforces clear separation of player-specific logic.

Tile

[files: tile.h/cc](#)

Tile is a hexagon on Board with integer index, resource type, value, and associated vertices (criterion). It tracks whether it is occupied by the goose, impacting resource distribution. We associate Criteria and Goal pointers with each Tile to enforce adjacency relationships. The class provides basic accessors (getResourceType, getIndex, getValue), but does not have modifying behaviour because Tile itself does not change states throughout the game.

Controller

Gameplay

[files: gameplay.h/cc](#)

[main.cc function](#)

[files: main.cc](#)

Gameplay acts as the mediator between the View and Model by managing game state and coordinating player actions with Board updates. It processes input to determine game progression + adherence to rules, which allows Board and Student models to reflect the current state (and thus the state of Criteria/Goals.)

main.cc initializes the models and board loading, before it begins Gameplay (which drives the game loop.) This is where the controller can take the driver’s seat.

BoardDisplay

files: boardDisplay.h textDisplay.h/cc, graphicsDisplay.h/cc

Abstract BoardDisplay (with TextDisplay and GraphicDisplay subclasses) serves the game board's state to players as a visualiser. Using the **Observer design pattern**, BoardDisplay attaches to the Board (Subject) to receive updates when the board changes. Plus, The **Factory design pattern** provides the dynamic creation of either a textual (TextDisplay) or graphical (GraphicDisplay) interface, so we can now vary the type of display for users.

design.

Challenge: Who should have ownership of certain game elements—in particular, goals, criteria, and students, that seem to be useful in many places?

The challenge was determining ownership for goals and criterion, which are shared among multiple tiles on a board. We resolved this by assigning ownership to a class that manages all of these entities Board rather than individual Tile objects, as the board logically encompasses all game elements. This centralized ownership simplified adjacency validation and separated tile game logic, which doesn't necessarily 'own' goals and criteria, but uses it constantly.

Challenge: There are two different types of dice rolls, but we don't want to keep writing *if* statements to determine which method to use. How can we use a design pattern to facilitate different types of rolls that change at runtime?

To handle both fair and loaded dice, we used the **Factory design pattern** to instantiate the appropriate dice type, in which can be changed as the game progresses. This abstraction allowed the game to create either FairDice or LoadedDice based on player input. By providing a consistent roll interface, dice behaviour can now vary without affecting the rest of the gameplay logic.

Challenge: What categorization fits better as a Class, versus an Enum?

To decide what to model as classes versus enums called for balancing complexity (i.e. how much information something stores) and functionality (i.e. what we need it for.) As an example, Tile and Criteria were implemented as classes because they have state and behaviour—tracking ownership + determining resource production. Conversely, simple, unchanging attributes like resource/completion types (CAFFEINE, EXAM, etc.) and player colours were represented as enums to reduce overhead. We wanted to ensure that elements requiring dynamic interaction had sufficient structure without over-complicating static identifiers.

Challenge: How do we store and connect adjacent Goals and Criteria?

From the provided watan.pdf, we observed that each tile, goal, and criteria has an ordered index according to the board format display. We use this concept, giving each of them a unique 'identifier' saying, "I'm criteria/goal/tile number x ," so that we could identify by integer. The hexagonal layout prompted us to use a constant map to define adjacency relationships, which connects tiles to their surrounding criteria + goals, and goals with criteria. This approach avoids recalculating adjacency dynamically for the requirements of our program (a fixed board structure.)

Challenge: Cool—now we have all these elements that have behaviour based on game mechanics. Where can we put together the sequence of play that allows the game to actually run, in a way that is encapsulated and does not interfere with mechanics?

Constraining responsibilities of Gameplay needed careful delegation of tasks to avoid bloating the controller and over-forcing its involvement. We decided that defining exactly how much Gameplay is involved in managing things was key to separating concerns in game logic. Using a DAC204 - Game Design concept 'Formal Game Elements - Sequence of Play,' we could determine that direct game state changes should be away from the controller and only done in the model component. That is, Gameplay focuses on managing game progression and processing inputs, while state updates are handled by Board, Student, and related models. For example, when a player completes a goal, Gameplay validates the action but delegates ownership and adjacency updates to the Goal object, which adheres to principles of MVC architecture.

resilience to change.

Design aspect	How we structured it	Changes we can process
Setting board up with resources + values	The Template Method design pattern allows new setup types to extend <code>BoardSetup</code> and override specific steps of configuring Board resources, without affecting others.	Ways to configure setup. Current specification processes random + file, but what if we want a ‘plentiful’ set up (mostly non-NETFLIX) or a ‘scarce’ set up (mostly NETFLIX?) These custom configurations are possible.
Types of dice rolls	The Factory design pattern alters the behaviour of <code>LoadedDice</code> and <code>FairDice</code> subclasses—both of which take from virtual class <code>Dice</code> .	Introducing new dice types (such as weighted dice, different # of sides, or advantage/disadvantage dice—when the dice roll happens twice and takes the highest/lowest value respectively.)
Storing data on criterion + goals	Created class <code>Board</code> where all instances of <code>Criteria</code> and <code>Goal</code> are stored in a vector and identified by its integer index. Used constant mapping for adjacency.	Changing adjacency rules (say being able to claim neighbours of neighbours), adding new relationships between criterion and goals. With the assistance of other maps, creating more/less tiles or differently shaped tiles.
Types of ways to view the board of the game	The Observer design pattern allows additional displays to attach to the <code>Board</code> w/o modifying existing code.	Adding new display types (for instance, 3D graphics, web-based UI, accessibility-focused interfaces (like colourblind), audio-based display).
Managing the turns in the turn-based sequence of play	A custom Iterator is used to rotate through the player vector (and automatically return to the start after reaching the end.)	Ever-changing turn orders or looping turn management (accommodating new player actions or skipping turns).
Enforcing game rules and game logic in mechanics	Encapsulating rules in classes like <code>Criteria</code> , <code>Goal</code> , and <code>Student</code> which examines specific conditions and permissions for each mechanic.	Modifying rules or adding/subtracting conditions, such as adjacency checks (e.g. neighbours of neighbours can be bought) or resource trade constraints (e.g. instead of x amount of resource, make it $x + 2$ amount of resource.)
Managing the inventory of each player	Student uses an <code>unordered_map</code> for resource tracking, allowing easy addition of new resource types. Methods like <code>addResource</code> , <code>removeResource</code> , and <code>playCriteria</code> define the behaviour of players’ actions.	Attaching new resource types (say we wanted to add EXTRACURRICULAR or INTERVIEW) or expanding the range of player actions (like introducing unique abilities or special trades, such as force-trading or bartering.)

How high cohesion and low coupling supported resilience to change... as stated in lecture, high cohesion and low coupling are key to modularity and maintenance, such that we have that door open to expand the program. To achieve this, we designed each class to focus on a single responsibility (SRP.) For example, the `Tile` class handles only attributes and operations related to resource production and adjacency, and does not stick its nose in the functionality of its owned components (like `Goal` and `Criteria`.)

Broader gameplay mechanics like turn order and resource distribution are managed by `Gameplay` to ensure high cohesion by keeping related functionality localized within classes. To reduce coupling, we relied on indirect communication through design patterns like the Observer—which is strongly regarded as a solid pattern for this aim—for `Board` and `BoardDisplay`, where the board doesn’t directly manage how updates are rendered.

Similarly, by using enums for immutable properties (w/o specific behaviour) like `ResourceType` and `Colour`, we avoided unnecessary dependencies between classes. This approach allowed us to modify one part of the system (e.g., how resources are displayed) w/o requiring changes to unrelated parts (e.g., how players interact with the board).

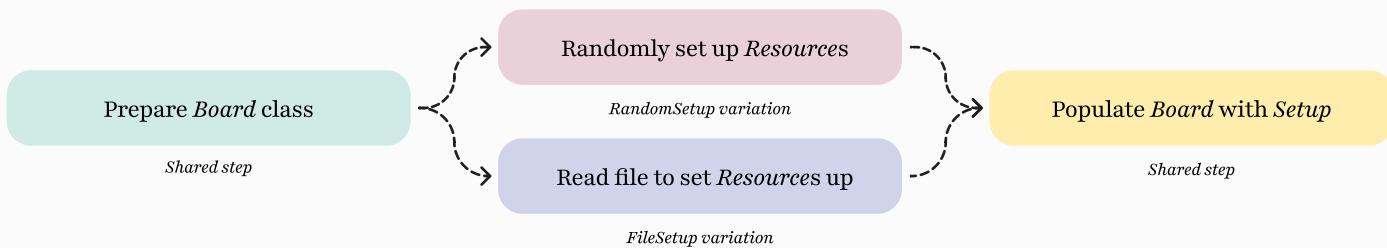
answers to questions.

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?

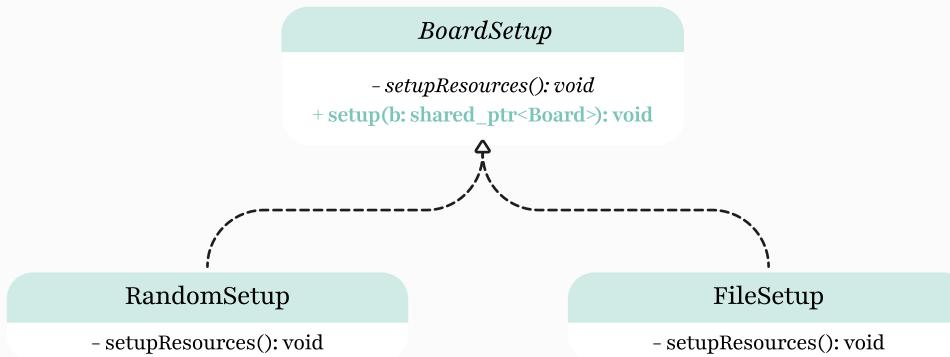
After implementing the code, we reaffirmed that the **Template design pattern** was the ideal choice for supporting both random and file-based board setups. Our reasoning remained consistent—the process of setting up the board possesses shared steps (e.g., populating the board) and steps that vary based on a random or file-based setup method.

1: The Template design pattern is ideal for altering steps of a process/algorithm, which is what we're doing here—we are promoting variations in the step it takes to set the board up, but ultimately, we end up with a *Board* instance that does not ultimately vary in how it handles behaviour.

2: That is, whether we set up randomly or with reading, we can actually end up with the same *Board* (in theory.) We don't require differentiating behaviour based on which "method" of set up is chosen, so we only must implement a change in one of the steps in algorithm, as shown below.



What *did* differ from our initial plan was the plan to implement the diagram above as three different methods (and the second one as the virtual, overridden method.) Instead, we created one public method, `setup(shared_ptr<Board> b)`, that wrapped `virtual void setupResources()` and added the shared steps together. The idea behind this approach was to avoid complexity in implementing the setup, since setup serves a sufficiently singular purpose. This is shared in the fixed diagram below:



In this diagram, the varying step is still `setupResources` method, implemented in *RandomSetup* and *FileSetup*.

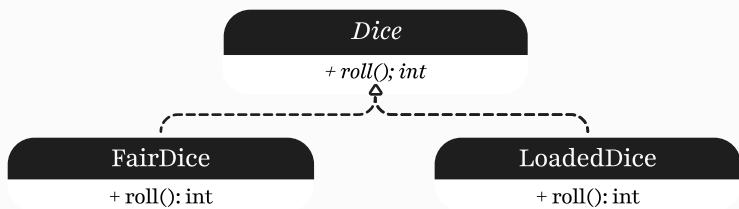
The public `setupBoard` function acts as the Template Method. That is, it organizes steps of the setup process in a sequence while delegating the resource-specific logic to its subclasses.

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 manner in which we implemented this feature does not differ from plan.pdf (DDI.)

The loaded and unloaded (fair) dice can be dynamically switched between at the player's discretion—a player could, on their turn, do something like: switch to fair, switch to loaded, fair, loaded, fair, loaded, loaded, loaded, fair, and *then* deploy their dice. Because of this runtime-based functionality, we implemented *Dice* with the **Factory design pattern**.

The purpose of Factory is for subclasses to alter the behaviour of its virtual superclass—that is, define distinct behaviours for loaded and fair dice—while allowing to keep it tucked away (encapsulated.) A player can switch between dice types by simply calling the factory to create the desired dice, adhering to runtime needs. By keeping the dice creation in the Factory, we avoid hardcoding behaviour changes in the dice itself, which separates the "what" (dice roll) from the "how" (roll retrieval.)

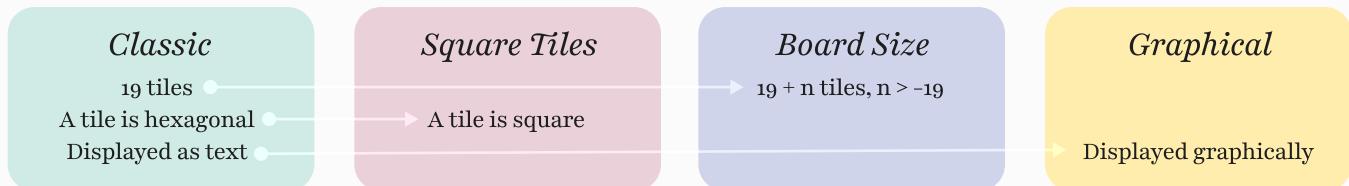


In this diagram, the *Dice* class acts as the abstract base class that defines the interface for rolling a die, encapsulated by *roll()*.

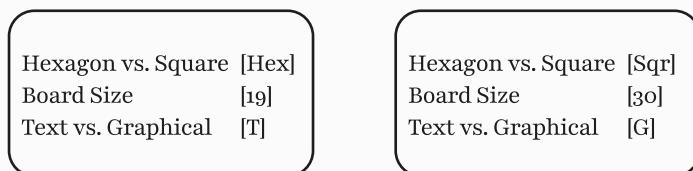
The two derived classes, *FairDice* and *LoadedDice*, override this *roll()* method to define rolling behaviours as provided.

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?

The game modes mentioned in this exploration—square tiles, board size, graphical display—are all different manners of building and displaying the *Board* class. In particular:



In our interpretation of these game modes, we have that each game mode feature is able to be ‘toggled.’ That is, we can frame it as ‘game settings’ in a light switch format as demonstrated below.



The settings of a ‘Classic’ game mode.

A game with 30 Square Tiles, displayed graphically.

If we consider ways to configure *Board Size* as two (as in 19 or not 19), then we actually possess 2^3 ways to configure the game:

(Hex vs. Square, Orig. Board Size vs. Custom Board Size, and Text vs. Graphical.)

In other words, we can ‘stack’ (dare we say *adorn*) different types of features to create these different game modes. If we were to implement it to allow our interpretation of these game modes, it would be the [Decorator design pattern](#).

By interpreting game modes as toggleable ‘features’, the Decorator design pattern is ideal because it allows us to dynamically combine or stack these features without predefining rigid game mode combinations. Each feature (say *SquareTileDecorator*, *GraphicalDisplayDecorator*, etc.) acts as a decorator that wraps the base configuration of *Board*, *Tile*, *View*, and other relevant classes, and modifies its behaviour as needed.

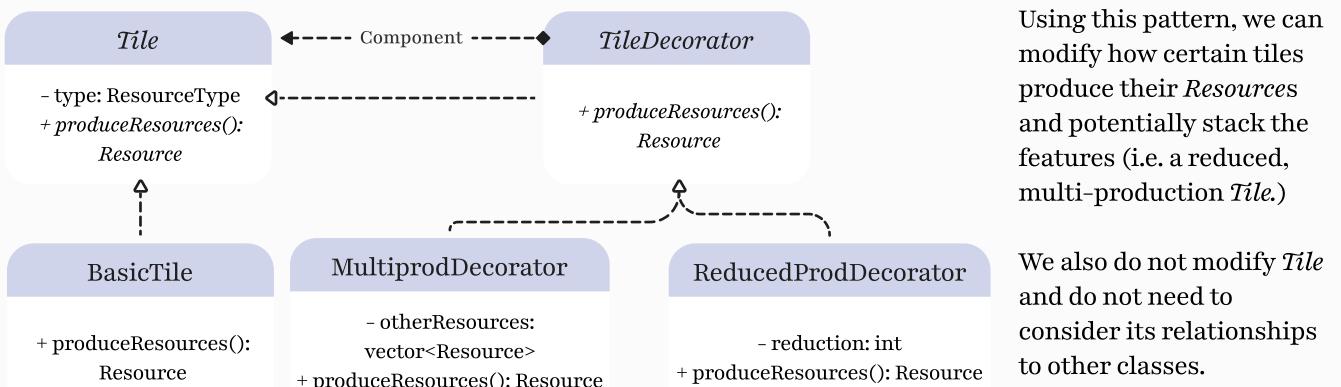
This approach avoids a combinatorial explosion of classes for every possible mode (otherwise, we’d have to make every combination such as *HexGraphicalBoard*, *SquareTextBoard*, etc.), and instead allows modular composition of features at runtime. It adheres to the **Single Responsibility Principle** by ensuring each decorator focuses on modifying only one aspect of the board—its shape, size, or display format.

This allows for a maintainable and scalable design that supports future extensions, such as adding new features (like rendering on 3-dimensional graphics,) by simply creating additional decorators.

We note that if, conversely, we are only letting one non-classic feature be present at a time, we would use the **Template design pattern** (to change *one* key implementation/step.) If we only implement one of these game modes, or multiple game modes that can only be toggled ‘on’ one at a time, we will pursue the Template design pattern.

6: Suppose we wanted to add a feature to change the tiles’ production once the game has begun. For example, being able to improve a tile so that multiple types of resources can be obtained from the tile, or reduce the quantity of resources produced by the tile over time. What design pattern(s) could you use to facilitate this ability?

In the case of features added to a tile, in which each modifies the manner in which tiles produce resources, can be interpreted as creating an alternative/modification to basic, or concrete, production. Thus, we see a use case for the [Decorator design pattern](#), which allows layering custom features on the production of *Tile* without altering its underlying class. We'd see this structure:



Noting the question's wording of 'once the game has begun,' the **Decorator design pattern** is a strong choice because it is designed to add or modify behaviours at runtime without altering the original class. In this context, where we want to update how a tile produces resources after the game has started (e.g., improve/reduce production or add multi-type production), the **Decorator** allows these changes to be applied without disrupting the core *Tile* class or its relation to key components like *Goal* and *Criteria*.

This method provides coverage for the base concept of feature implementation. However, say we want to group certain features of tile production together—for instance, we want a few *Tiles* that produces all types of resources (as in all of Caffeine, Assignment, Tutorial, Study, and Lecture) upon being selected, but it deprecates over time. Let's call this *Tile* the 'Luxury Tile.'

In the implementation of the 'Luxury Tile' lies the principle of grouping certain features together. In this case, we employ the [Factory design pattern](#) that produces certain types of Tiles with the Decorator-made features:



Finally, we employ the [Observer design pattern](#) and make *Tile* an Observer of the game state (so that might be *Gameplay* and *Board* objects, which harbour information about the state of the *Board* and the state of each *Student*.) This way, we can manage changes in the production of a *Tile* whenever an action that causes it comes into play. For instance, if a *Student* upgrades a *Tile* such that it is now a 'Luxury Tile', the according *Tile* object is notified to make those changes.

Doing so lets us decouple *Tile* updates from the objects that trigger them. *Tile* only needs to listen for notifications and adjust its behavior accordingly—stacking Decorators (such as adding resources) or replacing them (such as demoting from 'Luxury Tile' to 'Basic Tile'). This creates a scalable system where tiles react to events like upgrades or penalties w/o tightly coupling game logic.

We also encourage game state updates automatically propagate to *Tile* objects, eliminating explicit update calls. New events or upgrades can trigger notifications through the `notify()` method instead.

7: Did you use any exceptions in your project? If so, where did you use them and why? If not, give an example of a place that it would make sense to use exceptions in your project and explain why you didn't use them.

For *Students of Watan*, we implemented exceptions.h/cc into our program under the sections of our project:

File I/O Errors

WHEN Loading a game or a board config. file, or saving the current game state.

WHY If the file doesn't exist, is improperly formatted, or cannot be read/written.

Invalid User Input

WHEN User input is prompted for a player to act during their turn.

WHY If a user inputs a series of characters that does not validly translate to an action.

Conditions Not Met

WHEN An action or a state of the game attempts to change, but there require certain conditions to enact it.

WHY If one or more conditions for the action/game state change are not met.

The following exceptions have been named to describe specific situations encountered under each section:

FileNotFoundException

Thrown when the file to be loaded does not exist.

FormatException

Thrown when the file format is invalid or corrupted.

FileWriteException

Thrown when the game cannot save the current state due to write errors.

NEW!

InvalidCommandException

Thrown when the user inputs a command that is not recognized or does not correspond to any valid action.

InvalidInputException

Thrown when the user inputs invalid data that cannot be processed.

OutOfRangeInputException

Thrown when a numerical input is outside the acceptable range.

AlreadyOwnedException

Thrown when a user tries to purchase a Criteria or Goal that has already been purchased by the same or another player.

CHANGED FROM:

CriterionAlreadyPurchasedException
+ GoalAlreadyAchievedException to also include Goals and remain concise.

NonAdjacentPlacementException

Thrown when a user attempts to place a Criteria or Goal that is not adjacent to any of their existing placements, violating placement rules.

InsufficientResourcesException

Thrown when a user lacks the necessary resources to purchase or upgrade a Criteria/Goal.

InvalidCriteriaImprovementException

Thrown when a user tries to improve a Criteria that is either incomplete or already at its maximum level (i.e. "Exam").

InvalidGeesePlacementException

Thrown when a user attempts to place the Geese on a tile that already contains Geese, violating placement rules.

Misc.

We've also included one additional, miscellaneous exception as follows:

ResourceTypeConversionException

Thrown when there is an error converting between different resource types, such as during data parsing or type casting.

We implemented exceptions through the use of try-catch statements, in which exceptions were printed to the output stream as a manner of informing the user that their action is considered invalid, or that there are errors related to file input/output. We implemented try-catch for some (but not all) of these exception circumstances in order to avoid aborting or terminating the program in the event that an exception need not do so.

extra credit features.

We implemented smart pointers as described in the project guidelines, such that we do not clean heap-allocated data up using ‘delete’.

We implemented the handling of exceptions through thoroughly built try-catch statements.

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?

This project was sort of akin to running a *superhero* headquarters. Each of us—Angela, Yujin, and Halle—brought our own powers—whether it was design, implementation, or game mechanics expertise (shoutout DAC204)—and we learned the importance of playing to those strengths. We also had differing levels of team experience from the work done in previous co-op positions, personal projects, and hackathons—but this was *helpful* rather than a hinderance.

Delegation was our foremost handy tool. Halle took charge of Board mechanics, Yujin tackled BoardSetup and Student, and Angela worked on some key related elements like Tile, Criteria, and Goal. We brought this together with Gameplay, our view components, and writing this document collaboratively. Through Git and GitHub version control—in which was Angela’s first time using it—we pushed/pulled on *main* branch regularly.

Discord served as our command centre for quick check-ins, calls, and long discussions, like deciphering the semantics of game rules, as demonstrated on the right. We also had in-person meetups.



When disagreements arose, we hashed out solutions until they clicked—for instance, we had different interpretations of how the Geese move mechanic worked. Lots of screenshots later, we worked through its semantics.

Ultimately, this project proved that teamwork doesn’t solely rely on task division (as we students do in elementary-level work) but harmonizing strengths, collaborating ideas in-person, and staying united when the going gets tough.

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

If given a fresh start, we’d put a stronger emphasis on scalability and resilience to change. Namely, the following:

- 1: While representing resource/completion types as *enums* worked for this iteration, future expansions (maybe unique resource behaviours, like CAFFEINE slowly depleting if not used quickly) would benefit from modelling them as classes.
- 2: Configuring Student as the entity driving actions—rather than delegating solely to Gameplay—would leave the door open for custom computer-controlled players.
- 3: While constant maps were efficient for this scope, relying on them for adjacency felt like using a hammer where a compass would suffice—geometric calculations would have offered more resilience to changes or alternate layouts, like the square layout brought up in program specification questions.
- 4: We noticed the potential for smarter cohesion choices with respect to handling updates; encapsulating changes in a dedicated GameState handler could simplify propagation of actions across components.

Though these tweaks might have slowed us in the short term scope, we would’ve built a sturdier foundation for *Watan*’s evolution.