students of watan

plan.pdf
angela jin + halle chan + yujin bae
for cs 246 f24

Plan of Attack

We want to emulate a 'ground-up' style of building for *Students of Watan*, following the principle of **MVC implementation**. For us, that is (a: Model) developing primary *Resources* in the game and their properties, (b: Model) individual *Mechanics* and how *Resources* interact with each other, and (c: Controller, View) the string that ties these together and creates the *Sequence of Play*.

What this might look like is exemplified below, using *Criteria* and *Student* as an example.

Build *Criteria* class, w. members (i.e. adjacent *Criteria* array, pointer to a *Student...*).

Build *Student* class, w. members (i.e. *Criteria* and *Resource* vectors).

(a) Create in-game *Resources* (i.e. Classes)

In *Student* and *Criteria* class, add methods to assign a *Student* to a *Criteria* and attach a *Criteria* pointer to a Student. Configure a method to add *Victory Point(s)* to a *Student* when a Criteria is gained or upgraded.

(b) Configure related *Mechanics*

Create *Game* similar to test harnesses we've seen in A1-A4, where each input focuses on a *Student* who produces an action that affects its members, including that of *Criteria*.

(c) Produce *Sequence of Play*

We have created a sequence to demonstrate a plan that incorporates this three-phase implementation, called **Hatch 'n Honk.**

[DD1, PLAN] Hatch Phase:

There are two deliverables for Hatch (DD1.) Angela and Yujin, having more in-class knowledge of design patterns, will work on the UML Class Diagram with guidance from Halle, who has previously created simple UML Class Diagrams. Halle, having experience in Game Design principles from DAC204, will take on the plan.pdf, following a discussion of when things are ideally due.

mon. nov 18

Mon. Nov 18 3-7PM: Produce rough draft of *UML Class Diagram* to map out *Resources* and *Mechanics*, as well as their relationship to each other, based on the watan.pdf explanation of the game. Discuss different design pattern usages for certain "corners" of the game (e.g. a Factory vs. Decorator design pattern for Resources.) Write the *Plan of Attack* document.

Tue. Nov 19: Create a finished *UML Class Diagram* with all public methods (and members as necessary) as well as relationships between each Class + design patterns used in specific corners of the game.

nov 20 wed.

Wed. Nov 20: Finalize both deliverables from peer review of each others' work. Submit both.

[DD2, IMPLEMENTATION] Honk Phase:

For (a) and (b), group members will each manage some specific *Resources*. Halle will manage the *Board*, *BoardDisplay* and *Gameplay*. Angela will manage *Tiles*, *Criteria*, *Goals*, *Dice*, and *Gameplay*. Yujin will manage *Student*, *BoardSetup*, and *Gameplay*.

thu. nov 21

Thu. Nov 21 - Fri. Nov 22: (a) Create in-game *Resources*.
Resources to create: *Student;* *Dice: LoadedDice, FairDice;*
Criteria, Goals, Tile; *Board, BoardDisplay, TextDisplay, GraphicDisplay*
Which may be adjusted as necessary.

Sat. Nov 23 - Mon. Nov 25: (b) Configure related *Mechanics*.
Some examples of *Mechanics* include: The consequences of a Goose being moved on a *Dice* roll 7.
Improving an obtained *Criteria*.
Expending a *Resource* to obtain a *Goal*.
Setting resources up on a *Board*.
Which may be adjusted as necessary.

sun. nov 24

Mon. Nov 25 - Wed. Nov 26: (c) Produce *Sequence of Play*.
For this phase, we will all meet together to combine the mechanics of *Board* and *Game* such that we can create the function that operates the *Sequence of Play* (that is, the input and output itself during each turn of the player.) We will do so by creating one main function that operates the sequence, and then delegating tasks in the main function to mechanics as necessary. This also includes features such as the command line features.

nov. 27 thu.

Wed. Nov 26 - Thu. Nov 27: Test the program by writing down and verifying edge cases, the functionality of each *Mechanic*. Once the base game is created, our time commitment and priorities will decide if we want to add additional enhancements.

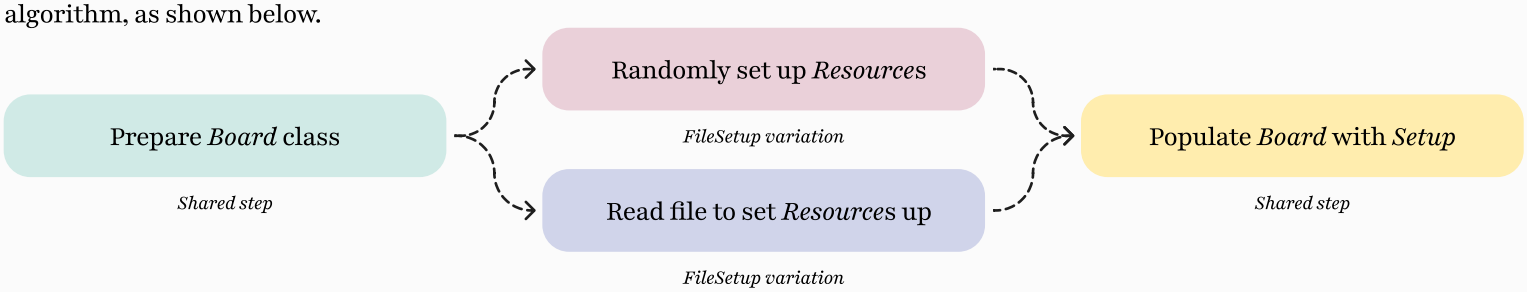
Program Spec. Questions

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?

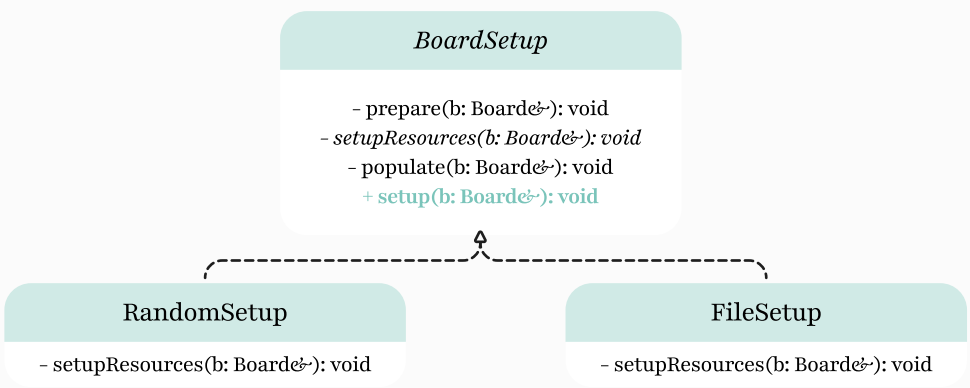
The 'tell' of the question are the words 'setting up,' which prompts us to think that we are looking for two different ways to generate the board—and 'generate' hints us to choose the Template design pattern. Here lie a few reasons that Template spoke to us, especially after comparing to other design pattern candidates like the Decorator or Factory.

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.



To implement the variations, we plan to construct the Class relationships below as an application of the Template design pattern:



In this diagram, the varying step is the **setupResources** method, implemented in *RandomSetup* and *FileSetup* to handle resource initialization accordingly.

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?

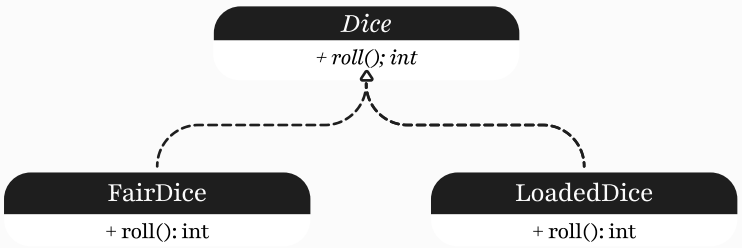
The loaded and unloaded (fair) dice can be dynamically switched between at the player's discretion—a player could, on their turn, do something like: switch to fair, switch to loaded, fair, loaded, fair, loaded, loaded, loaded, fair, and *then* deploy their dice. Because of this runtime-based functionality,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

We compare this implementation to that of the easy/hard 'levels' example in the Factory design pattern. We note that switching between an easy and hard level, each with unique attributes, utilises the Factory design pattern. Similarly, switching between the fair and loaded dice can be interpreted as 'levels' themselves, with unique behaviours, and can be produced/altered at runtime.

Program Spec. Questions

This is implemented with the structure below, demonstrating the relationship between the *Dice* classes:

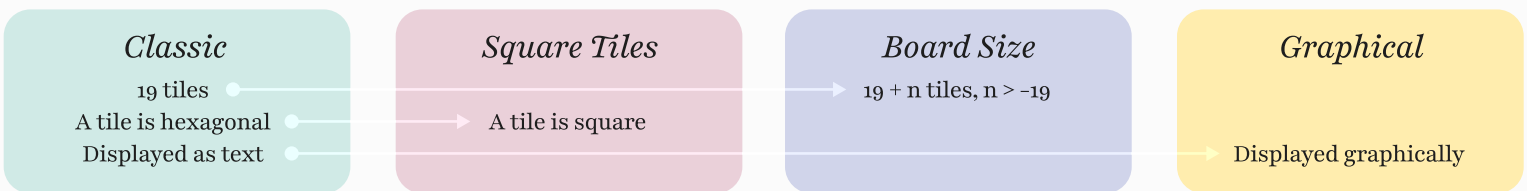


In this diagram, the *Dice* class acts as the abstract base class that defines the interface for rolling a die, encapsulated by roll().

The two derived classes, *FairDice* and *LoadedDice*, override this roll() method to define rolling behaviours as provided. *FairDice* generates the sum of two random, independent, and balanced dice rolls, while *LoadedDice* takes user input for its roll.

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.

Hexagon vs. Square	[Hex]
Board Size	[19]
Text vs. Graphical	[T]

The settings of a 'Classic' game mode.

Hexagon vs. Square	[Sqr]
Board Size	[30]
Text vs. Graphical	[G]

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.

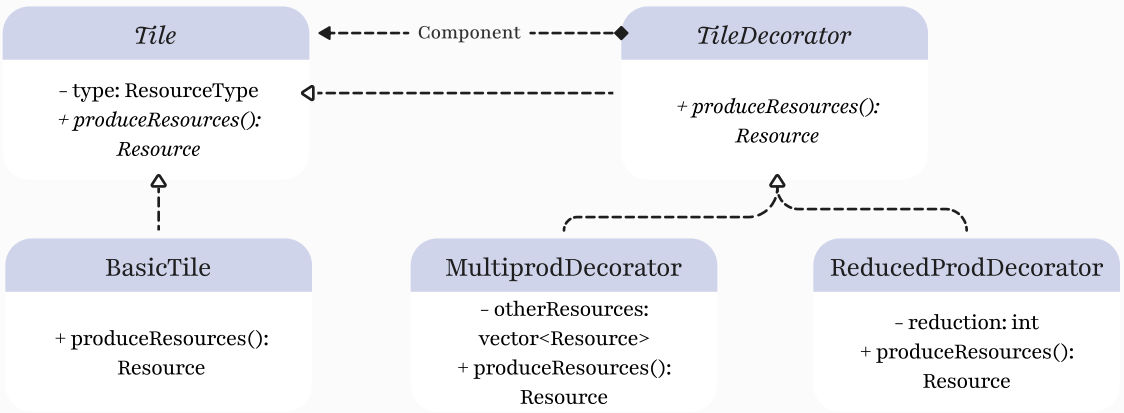
This allows for a maintainable and scalable design that supports future extensions, such as adding new features (like rendering on 3-dimensional graphics,) by simply creating additional decorators.

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.

Program Spec. Questions

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:



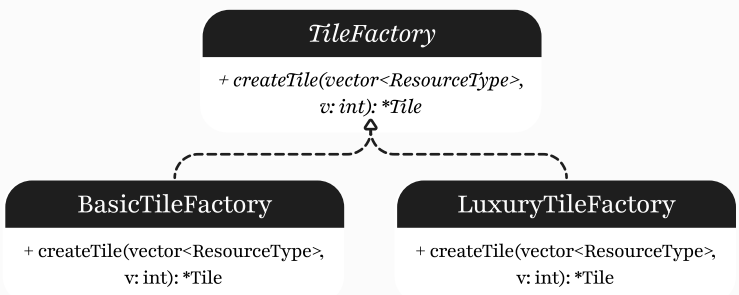
Using this pattern, we can modify how certain tiles produce their *Resources* and potentially stack the features (i.e. a reduced, multi-type production *Tile*.)

We also do not require to modify *Tile* and therefore do not need to concern ourselves about its relationships to other classes.

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 *Tile*s 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:



Using this design, we can bundle Decorator groups together to create certain types of *Tile* classes with certain attributes.

When thinking of scalability, this means we only need to add/remove/edit Decorators in one place instead of manually across many *Tile*s, if we want to make changes. Additionally, we can bundle different types of Decorators by adding more Factory Classes. This allows for a strong foundation for future updates.

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.

Program Spec. Questions

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 have identified and plan to use exceptions under the following 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.

FileFormatException
Thrown when the file format is invalid or corrupted.

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

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

OutOfRangeInputException
Thrown when a numerical input is outside the acceptable range.

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

NonAdjacentPlacementException
Thrown when a user tries to purchase a *Criteria* or *Goal* that is not adjacent to any of their existing completions.

InsufficientResourcesException
Thrown when a user does not have enough resources to purchase or upgrade a *Criteria*/*Goal*.

GoalAlreadyAchievedException
Thrown when a user tries to achieve a *Goal* that has already been achieved by another player.

InvalidCriterionImprovementException
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 tries to place the *Geese* on a *Tile* that already contains *Geese*.