



Constructor Final Design Document

Adrian Seto, Omar Baranek, Dhruv Mittal

Overview

For our CS 246 Constructor final project, we decided to implement the Model-View-Controller (MVC) design pattern. This architecture promotes high cohesion and low coupling of the model, view, and classes.

The Model, implemented as the BoardModel class in our final project, is responsible for handling the in-game logic and state. The specific implementation of the Model is explained in the following section.

We implemented the View framework of the MVC pattern as BoardView. The BoardView is responsible for dealing with the output to the user. The methods in BoardView are either called by the BoardModel class. In the case where the printController class requires an output method from BoardView, we would call the method in the BoardView class through the BoardModel class. Overall, this ensures that the MVC design pattern and encapsulation is upheld throughout print method calls.

The Controller of the MVC pattern was implemented through the BoardController class, which is solely responsible for handling and processing user input. It then works with main.cc to call the correct methods from BoardModel. We understand that the BoardController class is not intended to store data as member fields. However, to more efficiently utilize our time constraints during the exam period, we decided to store the seedValue, input file, and command-line option read in from main.cc as member fields so that users could play again in the event where the player indicates to replay. Thus, if we were given more time, we would have not stored data in the Controller class. We're also aware that the BoardController should not be responsible for printing any output to the user interface. Rather, all outputs should be handled by the BoardView class. This is because in the case where we want to change the implementation so that the output is printed elsewhere, for example a text file, then the print methods can all be changed easily in the BoardView class. However, due to time constraints, we decided to print any output that did not require data processing, such as `std::cout << "Invalid command" << std::endl;`, directly from the BoardController class.

We also implemented a main.cc file (as expected since a C++ program that needs to be run must have a main function). The main.cc file was responsible for dealing with the command line inputs and making sure the correct type of game is created. The main.cc file had a Controller class object. Through this object, the main.cc file could call functions such as start game or load game. The rest of the functionality did not concern main.cc.



Design

We constructed a BoardModel class to implement the Model framework of the MVC design pattern. The BoardModel class stored all the state of the game and delegated logic to the relevant classes that it has a composition relationship with. As the controller should not call the View directly, our model helped solve the communication gap. It also included key design decisions (other than MVC design pattern) as explained below.

To adhere to RAll principles, we decided to use smart pointers throughout the project. However, we soon realised that each vertex in the board needs to keep track of its adjacent vertices. Our previous implementation of storing a vector of Vertex pointers would include a cyclical reference as the neighbouring vertices would reference each other. To avoid this, we decided to only store the index location of each vertex in the adjacentVertices vector. Instead, we only stored all the shared pointers to Vertex, Edge, and Tile objects in vectors in the BoardModel class. The BoardModel would then provide each class (Vertex, Edge, Tile) with all the data needed to implement their own functions. This protected the shared pointers in only 1 class and also encouraged high cohesion as each class would have all the necessary data to fully implement its own functions.

To ensure consistency across all our classes, we chose to use enums for storing the various builder colour options and resource types. Realising that enums are effectively integers, we were able to define a colour of 0 as Blue, 1 as Red etc. Using enums made our code less error prone and more readable to our team members during code reviews.

Our choice of data structures for various features also helped solve numerous data storage problems. We chose to store all the builder's resources as a `std::map` with keys of ResourceType enum and integer values representing the quantity. This allowed us to access resources by constant references instead of dealing with the ResourceType and quantities separately. Using a map also helped make our functions more generic. A builder could receive a pair of any ResourceType and an integer quantity and it would be dealt with a single function. In BoardModel, we also chose to store the 4 players in a `std::vector` of shared Builder class pointers. We indexed these pointers by the Colour enum such that our first builder matched the enum Colour value of Blue (int 0).



Resilience to Change

We decided to implement the MVC model for our Constructor final project as it allows changes to any of the three major classes (Model, View, and Controller) with minimal changes to the other classes. The MVC design architecture promotes high cohesion and reduces coupling between the classes.

Due to the low coupling between classes, if the specifications of the final project were to be changed to incorporate a Graphic User Interface, we could simply change the BoardView class without needing to restructure our entire implementation. If the output is required to be redirected, such as writing to a text file, then most of the print method calls can be found and changed directly in the BoardView class.

Another example of how the MVC design architecture promotes ease to implementation adjustment is that if we were to reuse the same board for another game, the MVC design pattern would allow us to swap the BoardModel, which contains the game logic of Constructor, with the BoardModel of the new game while keeping the BoardController and BoardView class intact. This demonstrates both low coupling and high cohesion between the three major classes of the MVC design.

We implemented the residence specification as its own Residence class, achieving higher cohesion as all the residence's data is stored in one location. This means that any residential type additions such as duplexes, castles, and mansions can be made in this class, and the rest of the code will not be affected since everything needed to build and improve the residence type is done through the single Residence class.

By using maps, enums, and text files, we are able to adapt our code very quickly should there be additional resource types in the future. We would only need to add the resource type to our enum. All the member variables that use a map of ResourceTypes as keys could then be adjusted very easily. For example, if we were to implement a cottage residence which needed 3 Wood, we could simply add Wood to our enum and then edit the cost variable to include 3 Wood for a cottage residence. Of course all our functionalities such as playing Goose and trading resources would automatically adapt as a result.

Answers to Questions

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?

As previously explained, this can be implemented using a Non Virtual Interface Idiom- an extension of the Template Method design pattern. We would have a superclass called `SetUpGame` which has subclasses called `RandomBoard` and `FixedBoard`. The `SetUpGame` class will have a non virtual public function called `setUpGame()` which defines the basic steps required to set up the game. These include non board related components like setting up the builders. We would also have a private virtual function called `createBoard()` which would be responsible for setting up the board. The `setUpGame()` function would call the virtual `createBoard()` function which would create the relevant board based on the subclass implementation.

Based on command line inputs from the user, we can determine if we would like a `RandomBoard` or a `FixedBoard`. The `RandomBoard` subclass will override the `createBoard()` function and use the seed (if any) to generate a random board. The `FixedBoard` subclass would instead read from the given text file to create the board. This allows us the flexibility to simply change which board class we point to during “runtime” so that our `createBoard()` function builds the correct board. This implementation was first referenced in the due date 1 document.

We decided not to use this design pattern because we thought it would bring in unneeded complexity to our code for the given requirements. Instead, in our `BoardController`’s `startGame()` method, we simply check if the user wants a fixed or random board and proceed directly from there. The `BoardController` would call `BoardModel::initBoard()` or `BoardModel::initLoad()` depending on the user command line options. Whilst this may not be as “clean” an implementation, it reduces the class relation complexity and helps simplify the code for us.

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

A “factory method” design pattern will be most appropriate in this case. We can build a superclass called “Dice” which has subclasses of “LoadedDice” and “FairDice”. We would also have another superclass called “RollAbility” which has subclasses “BonusAbility” and “RegularAbility”. The `RollAbility` class will have a pure virtual `getDice()` function which returns a pointer to a Dice object. This can be seen in the UML diagram below:

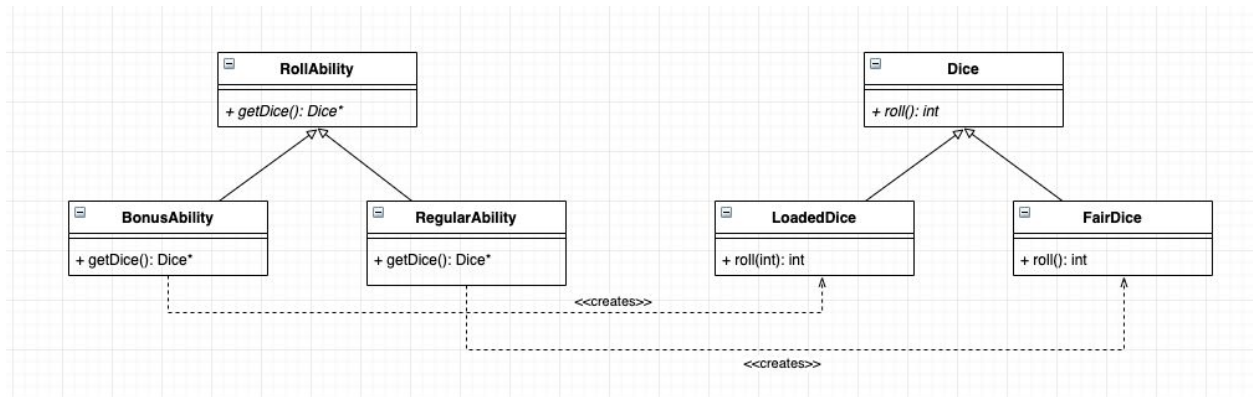



Figure 2: Factory Method design pattern for rolling dice

In our code, we would have a variable which points to a `RollAbility`. During the game, we can swap out this pointer based on if the user has unlocked the bonus ability (loaded dice) or not (fair dice). When the user is on bonus, their dice will be a `LoadedDice`, whose `roll` function takes an integer of their liking. However, when the user doesn't have the bonus, they will only have a `FairDice`; hence, they can only get random numbers from the `roll()` function of a `FairDice`.

We decided not to use this design pattern as the project guidelines did not specify when a user unlocks the loaded dice (we believe the loaded dice served as a testing tool instead of a game feature). Instead, the user has the choice to use the loaded dice or fair dice whenever they want.

Initially, to implement this, we thought of simply using a `Dice` class which has 2 functions overloaded, one which takes an integer and the other which simply produces a random number between 2 and 12 randomly given no arguments. We thought this would simplify our code and still satisfy the given requirements. However, when actually coding the project, we realised that the function which simply returns the user input is redundant. Instead, to save time we chose to simply implement this in the controller. We recognise that this should not be in the Controller, but in an effort to save time, we thought this was a logical compromise.

In the implementation of our final project, we decided to not use the Factory Method design pattern for rolling dice. We believed it would be an inefficient use of our team's limited time as the Factory Method would be an overkill of a design for a simple non-trivial implementation of a single `roll()` method. Instead, we implemented `setDice(char)`, which takes in a character (L for loaded dice and F for fair dice) as a parameter, and `rollDice()`. Then until the player inputs the 'roll' command, the `BoardController` will continue to set the dice type, either loaded or fair, by calling `BoardModel`'s `setDice(char)` method. Once the player locks in their choice of the dice by typing the 'roll' command, the `BoardModel`'s `rollDice()` method would be called by the `BoardController`. The `BoardModel::rollDice()` uses the `std::shuffle` algorithm to generate a random number between 2 and 12 inclusive.



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

The design pattern that we would use to incorporate all the different game modes is the Model View Controller (MVC) design model. By using the MVC model, we would be compartmentalizing the program's state, presentation logic, and control logic into separate sections.

By separating the Model, View, and the Controller, we would be able to change any part of the architecture without modifying the other two. For example, if we currently have a text display but we would like to implement a graphical display instead, then we could simply switch the View section of the architecture with no to minimal changes to the Model and Controller sections.

Regarding the program's state and logic implementation, the model defines the shape of our board, and thus we could simply change the model depending on the desired game mode while keeping the implementation of the Controller and View with minimal changes.

As explained above, our team did end up using the MVC model. In fact, as all the output is done through the BoardView class, we can simply swap out the View class with another one that displays the tiles as hexagons instead of rectangular tiles. If the shape of the board itself was changed, all we have to do is use the new View class to display the new board and provide new textfiles to the BoardModel class so that the init functions can initialise the board with the new layout. No change to the structure of our code will be needed.

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?

The decorator design pattern is suitable for the intent here since the decorator design pattern is intended to let you add functionality or features to an object at run-time rather than to the class as whole. One Way to implement the decorator design pattern to add/reduce the quantity produced by a tile is outlined below.

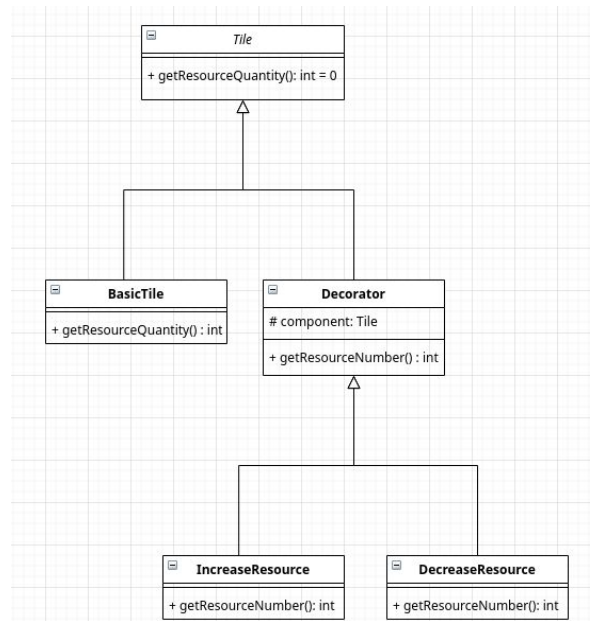



Figure 3: Decorator design pattern to change the tiles' production

In the above design pattern, we could define `getResourceQuantity` in the **BasicTile** to return the 1 representing the normal quantity. However, in the decorators, we could define `getResourceNum()` in **IncreaseResource** to return `component->getResourceNumber() + 1` to increase the resourceQuantity by 1. In **DecreaseResource** to return `component->getResourceNum() - 1` allowing us to chain these together and increase/decrease the quantity at any point during run-time. In this example, introducing the decorator design pattern might not be worth it due to the added complexity. However, as we add more and more features to the Tiles, the decorator is going to be crucial in making the code understandable and maintaining high cohesion and low coupling.

Whilst we were not able to implement this additional feature, our code is built to be able to adapt to these additions. Our **Tile** class can be decorated upon using the Decorator design pattern such that the resourceType, number of resources, and other special effects can be added or removed during runtime. As we effectively query our **Tile** class for information about each tile, making these additions will require minimum changes in the rest of the code.



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 where it would make sense to use exceptions in your project and explain why you didn't use them.

Yes, we plan on implementing exception safety in our program to ensure that the Constructor game is user-friendly. The exceptions we plan on using in our project can be split into two sections: validating user input, and game scope validity.

Validating user input includes catching exceptions when a saved file to read from is not readable and checking the bounds of user input (e.g. when a loaded dice is chosen, the value must be between 2 and 12). The game scope validity exceptions will catch exceptions related to the gameplay of Constructor. This includes if the user does not have enough resources to complete the current move, offering invalid resources in a trade, improving residences that a player does not have, building on occupied edges or vertices, and not saving before quitting.

Overall, catching the exceptions in validating user input and game scope validity ensures that the game is user-friendly (e.g. an exception message is given on the display to notify the user of the invalid move).


In the implementation of our Constructor final project, we utilized the Standard C++ exception library to throw exception errors, particularly `logic_error` and `invalid_argument` errors. By using the Standard C++ exception library to throw errors, we ensured that the exceptions would be consistent and reliable with dealing with errors through the throw expressions. We also made an internal team document where we agreed upon the types of exceptions we will throw in all the situations and what the error messages will be.

Final Questions

What lessons did this project teach you about developing software in teams?

One key takeaway from the CS 246 final project is that when developing software in teams, it is crucial to delegate the work evenly amongst the team members. Additionally, it is recommended to delegate the work in a way where the classes are related to each other. For example, the teammate who is delegated the `BoardController` could also take on the `BoardView` and `main.cc` aspects. This is because both classes and `main.cc` handle and process user input.

Another key lesson we all learnt was that in order to succeed at creating a team-based project, we must communicate with each other. Since everyone programmed their own delegated section of the project, continual communication is key to successfully putting the pieces together. After completing each of our own delegated classes, we merged the code together. We realized that there were slight deviations between what each teammate understood and assumed. For example, invariants that were thought to be upheld were actually broken.



Moreover, this project gave our team valuable experience in using Github and git version control. We decided to host our code in a private github repository so that all members could have access to the code and work on different branches. We also learnt how to connect our ssh servers to our github repositories to allow us to test our code on the school environment. By creating up to 20 branches, we were able to learn how to manage all the version control and make sure that changes were not being made on the master branch and each person was able to test their branch before merging it with the master code in the main branch.

We also understood the importance of a well-drawn and well-planned UML diagram. It provides a basis for everyone to refer back to. Additionally, it communicates the methods available in one class for other respective classes. For example, without the UML diagram, the `playTurn()` method in `BoardController` would not know of the `buildResidence` method in `BoardModel`.

Lastly, as this was the largest project any of our teammates had been involved in, we all learnt how to manage the timeline of a project as big as this one. Initially we were determined to improve the user interface and implement additional features. However, due to other final examination commitments and a lot of time spent on debugging the code, we realised that it is not always possible to accomplish everything that a team initially sets out to achieve. Overall, whilst we are disappointed that we could not add any additional features, we are proud of the final product and the effort gone into coding it.


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

Points of improvement can be dissected into two main categories: Planning and Implementation.

When coding the `BoardView` class, we left the `printBoard()` method to end as we were adamant on finding an algorithm to print out the entire Constructor's board. This was obviously a mistake as it prevented us from visually validating our output. Thus, one planning-related improvement would be to complete the `printBoard()` function much earlier.

When trying to combine all the git branches together, we ran into many bugs and errors. This is partly because whilst coding our separate classes, we were unable to compile the code to validate whether or not our logical reason was sound. Thus, another planning-related point of improvement would be to begin bringing all the different classes together to make a minimum viable product earlier.

We would also have started coding the smaller classes, such as the `Vertex`, `Edge`, and `Residence`, first before tackling the `BoardModel`. This is because the smaller classes are easier to code and have less dependencies on other classes. On the other hand, `BoardModel` needs `Tiles`, `Vertex`, `Edge`, `Residence` and `Builder` classes to be coded. Additionally, by coding the smaller classes first, we would gain a better understanding of the interdependence and logic of the game before addressing the bigger, more difficult `BoardModel` class. Having said that, it was a good decision to implement the `initBoard()` function early on to help us test the code.



When debugging our program, we were required to review our teammate's code. However, we often ran into the problem where we did not understand our teammate's code due to the fact that sufficient comments were not provided. Thus, an Implementation-related point of improvement would be to include more commentary explaining any non-trivial methods.

If we had the chance to start over, we would put more emphasis on the `printBoard()` method in the `BoardView` class. One possible alternative implementation would possibly be to read from a text file so that we can print different types of board depending on the text file. This would drastically improve adaptivity to change for the board type for better efficiency.

Lastly, as previously discussed, due to the limited time during the exam period, we did not implement all the output in the `BoardView` model. Instead, we decided to implement only the output methods that process data in the `BoardView` class. We understand that this is a limitation because if all output methods were located in the `BoardView` class and we wanted to change the implementation so that the output is printed elsewhere, then the print methods can all be located and changed accordingly in a single class. Thus, if we had the chance to start over, we would implement all the print methods in the `BoardView` class.