Constructor Final Design Plan

Alice Li, Daniel Lu, Matthew Tam

Introduction

This document outlines the design specifications of our project, Constructor. This is a variant of the game Settlers of Catan and is implemented with object-oriented design considerations. The following sections provide an overview of how the program is implemented at a high level, specific design choices, the program's resilience to changes, answers to key questions, and the extra miles we went for the project.

Overview (overall structure)

The project is structured such that a few different modules act as major classes through which either the text-based I/O is handled, the logic is handled, and the game and player states are handled.

Game Class

Upon starting a game of Constructor, an instance of the **Game** class is initialized, and through overloaded constructors, different initial states can be handled depending on whether various options such as -random-board or -load are used. The **Game** class in turn initializes an instance of the **Board** class and instances of the **Player** class that handle the board and player states respectively. The **Game** class has ownership of the **Board** and **Player** classes and through RAII principles as unique and shared pointers, has stack-based objects allocated to handle their destruction when necessary.

The **Game** class handles the majority of the player I/O, while also taking care of the logic behind when and how to call methods of the **Player** and **Board** classes that hold the game state. Specifically, the handling of various moves that are passed in as string inputs is handled inside of the game, which then calls upon various methods of the players and the board to either access or manipulate the state. If a move is invalid, the **Game** class will catch the exception that is thrown by either the players or the board and handles it to maintain a strong guarantee. This can be said to follow the *Facade* design pattern as it is the interface through which the main program calls to do various actions throughout the game. The subsystem underneath of the **Player** and **Board** class is fully handled through the game, rather than having the main program directly ever access the board or players.

Board Class

The **Board** class contains the majority of the game state and can be said to also use the *Facade* design pattern such that it acts as a unified interface for the subsystem classes that exist to handle the board state. This subsystem includes tile states which in turn are represented by **Tile**'s, **Vertex**'s, and **Road**'s. The **Board** class has ownership of all three of these classes such that a tile, vertex, or road will never exist outside of a board class. Moreover, all functionality involving board state whether that be getting access to the board state or manipulating board state is through the **Board** class' public functions, such that the **Game** never has to directly access a **Vertex**, **Tile**, or **Road**. The tiles then have an aggregation relationship with the vertices and roads. As well, each road and each vertex has references to the roads and vertices they are adjacent to which we'll go into more detail for later.

In addition, the **Board** class requires that certain invariant properties be withheld such as residences not being built where there is an adjacent residence, roads being built off of existing roads or residences, etc. If these invariant properties are broken, the public functions throw exceptions and maintain strong guarantees.

Player Class

The **Player** class contains the player states including their resources, locations of their roads, buildings, their dice, etc. The player class handles access and manipulation of player states such as building a residence, acquiring resources, paying resources to trade, etc. Each player has ownership of a **Dice**, and through the *strategy* design pattern, uses their own dice whenever they roll to handle different types of dice rolls.

The **Player** class, similar to the **Board** class, holds its own set of invariant properties. For example, their resources cannot fall below 0 (they cannot build when they do not have the required resources). The **Player** class was implemented with the "Big Five" operators such that when we handle exceptions (incorrect input), we create a copy of the players prior to executing the commands and handle the logic such that we maintain a strong guarantee.

Design (specific techniques used to solve various design challenges)

Design Choice: Board State - Facade Pattern

We chose to implement our original plan of using a central board class that has a compositional relationship with a subsystem of classes that represented each of the Tiles, Vertices, and Roads inside of the board. Through this, we implemented the *Facade* design pattern, creating a central interface for the Game class to more easily handle logic solely through the Board, rather than having to call methods directly on the Roads or Vertices. Looking at our UML diagram, rather than having the game directly interact with any of the sub-system that contains board information, we have one single compositional arrow from Game to Board.

The subsystem that keeps track of board information consists of the Tile class, the Vertex Class, and the Road class. We felt that while one might argue it is better "OO-design" to have a compositional relationship between tile and the somewhat sub-classes of Vertex and Road, the Constructor board, unlike other board games, does not really have distinct tiles in the way other games do. Thus, we felt that the board should have direct access to all of the vertices and roads. From there, it was clear that the board should have ownership of these classes, and the subclasses should simply have weak_ptr's to one another.

Specifically, a Tile object has weak pointers to the Road and Vertex objects that represent the intersections and edges that are associated with any tile. Similarly, each road has pointers to their associated vertex's and vice versa.

From this, the Board class handles any attempts to retrieve state information about the game board as well as any attempts to change the state. This allows a tremendous amount of encapsulation of board implementation and allows us to easily change the board implementation moving forward, as the game logic solely depends on that outward interface of the Board class' public methods.

Design Choice: Separation of Concerns — allowing Game to acting as the intermediary

We chose to allow the Game class to act as an intermediary between the Board and the Players, as well as the primary handler of I/O. Rather than allowing the board to directly communicate with the players, we felt it necessary to reduce coupling as much as possible by allowing no direct communication between the players and the board. This way, both the board and the player implementation can retain encapsulation, while all the game logic is also handled in just one place. This means, if we so choose, we can easily change how the logic of the game moves, and the player and board wouldn't have to change whatsoever. We could introduce new Game moves easily through recycling the methods that we implemented in the player and board interfaces.

Consider for example how we handled rolling the dice. We pass the rolled number to the board as an argument of board->getRollResources(roll). This then returns a map of the players that get resources and a map of the resource types they get and the number of resources for each resource of map<PlayerColor, map<Resource, int>>. Rather than having the board directly give resources to the player, we separated the game logic and passed this back to the game. The game would then take this map

and call on the player methods that give resources to each of the individual players. This key difference from having the getRollResource(roll) method simply give resources to the players allows the board and players to know almost nothing of each other or each other's implementation.

Design Choice: Separation of Concerns — Exception Safety and Exception Handling

This separation of concerns lended itself tremendously well to our exception handling. Specifically, the Game class which handles the logic of the push and pull between the player method calls and board method calls, also handles catching all of the exceptions that are thrown by either the Board or the Player. This allowed us to centralize the exception handling and more easily deal with asking for more input.

Consider for example building a residence. Rather than having a central game that has to parse the adjacent roads and the adjacent residences for validity as well as checking for player resource levels, we leave those to their respective classes. The player maintains the invariant that their resources cannot fall below 0. The board maintains that residences cannot be on adjacent vertices and they can only be built if an adjacent road is owned by the same color (or if it is the start of the game). The board then does this by asking the specific vertex to check for validity of building a residence there. If the residence cannot be built by the vertex that scouts its nearby vertices and roads, then an exception is thrown all the way back up to game, while the board itself maintains a strong guarantee.

From here, we chose to handle maintaining an overall strong guarantee of building the residence between the player and board changes by creating a player copy constructor, and consequently the "Big 5" operators in player. This allowed us to create copies of the player, manipulate these player copies, and ultimately swap the player copy pointers with the pointers in the game only if many method calls to the copies did not throw any exceptions, and the one method call to the board did not throw any exceptions.

Design Choice: Dice Rolling - Strategy design pattern

We chose to stick to our original plan of using the strategy design pattern to implement dice rolling. Specifically, we have a pure virtual abstract base class of Dice. From this, we created the concrete child classes of FairDice and LoadedDice where the override'd method of rollDice() contains different strategies for how the dice should be rolled. On one hand the fairDice contains some simple use of pseudo-random functions to generate a pseudo-random number. The LoadedDice invokes the additional user input required for a loaded response. Both of which ultimately return an int 0.

Resilience to Change

One way our program shows resilience is that we implemented our code following object-oriented principle and separated the concerns from the player side and from the board side. Currently, our Game class handles the moves and i/o from the user, but when it calls for functions, it calls for the Player-side function and the Board-side function. This way, the Board object does not directly call Player methods and Player does not call those from Board. We let the Game, the central controller / facade, handle this. This means that for the changes we implement to Board, the Player is unaffected by it, and vice versa, any change we make to Player, the Board does not need to accommodate and know about it.

Due to this, our Game logic is highly resilient to change — in particular, for implementing new moves or logic to the game. We can quickly and easily implement new moves by using already implemented Player and Board methods. For example, it was quite easy for us to implement market-trade and multi-trade because we could simply re-use the player methods that we had already implemented for build-res, build-road, and trade. Moving forward, these same functions could be re-used to do things such as implement catan cards that allow you to build a residence for free, gain points for free, steal resources, etc. The errors such as stealing resources from people that don't have resources would already be handled by our encapsulated exception throwing.

This brings us to our next key point: separation of exception handling. By separating the exception handling concerns through encapsulation, new player moves that could potentially break invariants and be invalid moves such as those that require resources, those that build residences, or those that build roads, are easily handled through simple exception catching of exceptions we've already implemented. Moreover, by creating custom exceptions such as InsufficientResourceException, we can pass back information about the broken invariant for error outputs such as the resource missing or the color that lacked the resource.

In a broader view, the separation of the Player and Board classes is a key step towards resilience to change. Since the board state is kept completely separate with its own invariants maintained, if ever we choose to make significant changes to how either one functions, through encapsulation and low coupling, we keep the amount of cross module maintenance to a minimum. Say for example we decided to increase the size of the board, or potentially change the number of vertices associated with each tile. In doing so, the player would not have to change whatsoever — a huge difference from a player that has access to vertices or roads, and chooses to build a road by directly manipulating the road. This is particularly potent due to the use of the *Facade* design pattern that separates the interface of the board so that the client only interacts with the interface. In doing so, even the Game class would have to change very little if at all for a differently sized or shaped board.

Some ways we handle future changes is that we made an enum.h file and a constants.h file for maintainability. In our actual code, we rarely used any numbers, strings or enums. Rather, these are stored as constant integers, constant maps, and constant vectors in our constants.h file. This means that if we have to make any changes to the rules, we can simply change the desired constant in the constant file. Because our functions simply call these constants, they also require all the necessary information from them, like calling .size() on the vectors and .at() on the maps to find out how many times a for-loop is implemented or what the corresponding Player color to a turn is.

For example, if we were to change the rule so that the cost to build and upgrade residences is different, we can simply go into the constants.h file, find the map that maps the cost of building, and modify the number (value of the map) required for each resource (key of the map). Let's say that we are facing inflation and want the cost of building to increase over the time of gameplay. Then we can create a function that calculates the amount of increase for each resource, pass in the values from the constant map, and return the new cost.

Similarly, the building points from each type of residence is stored as a constant. It is easy to go in and change the values in the map if the game rules wish to change how many points a type of building is worth. Let's say that we wish to move this game from Waterloo to Cali and wish to change the resources completely, this can be done by changing the strings and resource enums in the constants.h and enums.h.

To handle the changes for a different number of players, this is a similar approach. Since we stored the player numbers, player colors, and their colors to strings (for printing) in the constants file, it would work to add a new name to a player to the vectors/maps in the file. Since our game constructor handles the creation of players and the turns of players in a loop going through each of these players stored in the constant, they will automatically adapt to the addition of a player.

The same goes for the tiles creation, where the tiles are looped through to be created and allocated resource.We would just change the number of tiles we want in the constant. The only complexity for handling more or less tiles is the printing of the board, but the internal tiles' information and interactions would stay the same.

Our program demonstrates low coupling because our modules display high independence and changes to modules don't heavily impact other modules. We designed it so that our central Game object depends on the modules Board and Player. Board is completely separate from Player so that whatever changes we make to Player's functionality, Board does not know about and vice versa. On the Board side of the Game, Board knows about its Tile, Road and Vertex. However, none of the Tiles, Roads and Vertices classes depend on each other. This is because they all live within the Board (owned by the board) but don't need to know each other's implementation. Whenever Tile needs to know its Vertices/Roads, Board takes it from the Vertex/Road module and passes it to the specific Tile. This way, we can make

changes to the internals of the Tile, Vertex and Road modules without worrying about their impact on each other or on the Player end. They would only impact the Board and the Game. On the other side for Players, Players depends on its Dice and so do the FairDice and LoadedDice classes. However, any changes to FairDice or LoadedDice would not impact ANY other module. Similarly, any changes to Player would never change the Fair/LoadedDice either. Since Dice itself is a virtual class, we will rarely make changes to it. Hence, these low coupling designs allow for reusability of the modules and keeps them in small, isolated chunks.

Our program also has high cohesion because the elements in the modules are directly related to the functionality that module is meant to need. In our Player side, Player only calls for its Dice types when rolling the dice. On the Board side, the Board only calls methods in the Vertices and Roads when building/modifying them directly. For example, Board can call its function to upgradeResidence(), which calls the method at Vertex to upgrade(). These are associated behaviors that reflect what wants to be achieved - they simply "make sense" together. Same as the interactions for Board and Tile, their functionalities are very grouped together with a shared goal, whether it is changing the geese or rolling the resources. Furthermore, these calls to the other modules are relevant and the functionality is clearly defined. Each functionality belongs together and serves a common purpose. For example, a board's method to getRollResources() calls for the produceResources() method in its Tile, both of which serve the purpose of allocating resources after rolling dice.

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?

To choose between randomly setting up vs. reading the resources from a file, the **Template** design pattern can be used. This is because the Template pattern allows the steps taken in an algorithm to be defined, and it allows the subclasses to redefine one or more individual steps without changing the overall sequence of steps. In choosing to randomize or read resources, our steps to set up the board would be the same in either case as we set up the game, create the players, and create the individual tiles. However, the step we want to choose / customize is whether we want to randomize the resources. The Template design pattern allows for flexibility so that we can choose one or the other option without changing the structure of our overall code logic.

We did not end up using this design pattern. This is because we found it more straightforward to simply implement the decision via a conditional statement, and the design pattern was not necessary. We found that it was sufficient to let overloaded constructors of the game handle the creation of the players, individual tiles, and set up the game. To handle the difference between reading and random set-up, we simply passed into the game constructor a randomized board if the board was not read in. The overloaded constructor would then handle, based on if there were additional arguments given for a game state that we want to "-load" in, we would then construct the players and whatnot accordingly. We felt this was simpler than to make a separate class altogether to handle these decisions.

2. 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?

The switch between loaded and fair dice at run-time, we opted to use the **Strategy** design pattern. The Strategy design pattern is used to define a family of algorithms, encapsulating each one, and making them interchangeable. Examples of such algorithms are those used to determine the outcome when rolling a fair and loaded dice. The algorithm can be dynamically changed and the design pattern helps to eliminate

the need for conditional statements to select behaviour. We are planning on using this design pattern because this would allow us to switch between a fair dice and a loaded dice at the players' choosing during each turn.

We did use this design pattern because we thought this coheres well with the object-oriented design principles. The players own their dice and when they call their dice to roll, they don't need to check again what dice they had once they have set their dice before. And the loaded dice and fair dice are distinct, different objects. Therefore, we decided to dynamically change the object the dice is in the player, so that rollDice() can be called on either type of dice no matter which one the player has set before.

3. 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?

To facilitate additional game mode modes, in particular those with alternative displays, its critical to separate the board state, the game logic, and the display. To do so, the **subject and observer** design pattern would be quite effective. To implement a graphical display, in the same way we added on an additional x11 graphical display in our Conway's Game of Life assignment, we could incorporate an additional graphical display simply by adding an additional observer. The board state (basements built, roads, etc.) would act as a subject which text and graphical displays will observe.

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?

One possible design pattern would be to utilize the factory design pattern. More specifically, the **factory** design pattern can be used to differentiate between creating different levels of computer players through implementing an abstract base class for the computer player where the concrete method for initializing the computer will construct a different computer player object based on the concrete level of difficulty.

Another way we can dynamically change to computer players and support different strategies is through the **Strategy** design pattern. This is because the Strategy design pattern allows the computer to determine which type of algorithm to use at runtime. If the context is to go easy and implement any valid moves, the Strategy pattern allows the code to take in this run-time instruction and select the appropriate algorithm. If the players want to choose a more aggressive mode, this run-time instruction can also be taken in and select the algorithm.

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?

To add a feature to change the tiles' production of resources, the **Decorator** design pattern is useful. This is because since the tiles have already been produced, we can use it as the base component / concrete component and build features on top of it. The Decorator pattern allows us to add extra features to an object at run-time rather than to an entire class, and it also allows these features to be withdrawn. This is useful since then we can "build onto" our basic tiles and as we wish to change/update/reduce the resources at run-time, we can add a decorator subclass to the base tile without changing other tiles at the time and without permanently changing the tile itself.

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

We used exceptions throughout our code for the project. We added exceptions in all functions that perform changes to the board and to the players, in our Board and Player classes. This is because we aimed to

maintain the invariants in the game and the restrictions on the moves that can be made. This is also so that we can “undo” the effects and prevent the modification of the game’s state in the case of an error in a function. In addition, we also made sure to handle the exceptions with strong exception safety by using swaps and other noexcept functions.

In the Player class, we throw an exception InsufficientResourceException(Color color, Resource resource) every time the player tries to build/improve a building or road, or other situations where resources are taken but the player does not have that sufficient resource. This achieves two goals: to reinforce that the player’s resources are always non-negative, and that the game rules are reinforced so that builders cannot build without sufficient resources. The constructor takes in information about the player that had insufficient resources as well as the resource that was missing. If in the future, we choose to further complicate our multi-trade by allowing multiple types of resources, we could pass a set of resources or even a map and the number of resources missing inside of the custom exception. The Player class also throws PlayerResidenceTypeException() for when they try to upgrade a building but they do not own it / the building is already a Tower. This reinforces the game rule of capping the building type at Tower, as well as preventing a wrong location the builder inputted.

In the Road class, we throw the exceptions RoadExistsException() and InvalidRoadLocationException(). These are to reinforce the games rules that a player can only build on an unoccupied road, and only on a road connected by a road/residence already owned by that owner. Similarly in the Vertex class, we throw the exceptions BuildingExistsException() and InvalidLocationException(). In addition, InvalidLocationException() also reinforces the rule that buildings cannot be on adjacent vertices. Vertex can also throw BuildingNotOwnedException() to prevent a builder from upgrading a residence that is not theirs.

In the Board class, our GeeseExistsHereException() checks the game rule to put the geese on a different tile, and GeeseOutOfRange to reinforce the invariant of valid tile location numbers.

Extra Credit Features

Our program is implemented without explicitly managing our own memory. We handled all memory management via STL containers and smart pointers. The difficulties were to familiarize with the methods of a variety of conventions, including vectors, maps, sets, unique_ptr, shared_ptr, weak_ptrs, and combinations of vectors of maps, maps of maps, and converting the smart ptrs for usage.

We implemented the extra feature of market-trade. This allows the builder to trade for any resource at a 4:1 rate, so if they have four of the same resource, they can trade them for one of a resource of their choosing. We implemented this feature to mimic the real gameplay of catan (trading with the bank). One challenge we had was to handle the continuous input of i/o in the case that an invalid input is taken in, and to make sure the program does not simply exit. We solved this by checking the failbit of cin ad wrapping try-catch blocks in while loops.

We also implemented the multi-trade feature. This works similarly to trade but allows users to trade resources for amounts of their choosing. For example, if Blue desperately needs Wifi, he can propose to trade 5 Bricks in exchange for 2 Wifi from Yellow. This allows more user decisions/strategies and spices up the game play.

We also implemented the checking of the format of input boards and saved games. For example, if the file passed in for -board contains more or less than 19 pairs of integers (since there are 19 tiles), if there contains non-integers, if the resource number is not between 0-5, the park (resource number 5) is not followed by the tile value 7, or if the tile value is not between 2-12, the board file format is not accepted and the game does not run. The same is for the board in -load, and in addition, the saved game file for -load must contain valid integer turn, resource amounts, road locations, and B/T/H followed by any building location number. We put these checking into place so that, instead of blindly taking in wrong boards or games, the end users are notified what the problem detected is.

It is worth noting that we have implemented our program with the separation of concerns for exception handling, as well as for handling different parts of the game. For example, we handled the board-side and

player-side error checking separately, and we distinguished out the different phases of the game in our code. This is so that our program is more maintainable and easier for other developers to collaborate / build onto it.

Final Questions

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 taught us the key teamwork and collaboration skills required in developing software in teams. Firstly, writing a large program with different team members requires coordination and delegation of tasks. Although more teammates means more brains to work on separate parts/features of the program, it is not a purely additive task and it was crucial to coordinate at the beginning so everyone stays on the same page. Our team spent a considerate amount of time before we started coding to plan out all the different classes we need, the to-do's along the timeline of the project, and the division of tasks to each member. This way, we all had an understanding of why we are working on our own parts as well as what others are working on. This also enabled compatibility of our code when we put the different parts together, and we ran into few changes we had to adjust for each others' parts.

This project also taught us the importance of prompt, clear communication in team software development. As we designed and implemented from scratch, there were a lot of details to touch base on, get clarifications on how the team wants to implement something, and get help from another member when there is an issue. Whenever we each merged our changes, we learned to keep each other informed so that we are all up to date and we would all know who made the changes/why. Because we are working in different timezones, we also needed to communicate and accommodate each others' availability to meet.

And very importantly, we learned to take responsibility and ownership in the team. Although it can be easy to put in less work thinking to freeride, or thinking that others would freeride on oneself's effort, we each took on our shared responsibility. For example, we saw through the correctness of each of our individual parts, and when we recognized new agenda items to get done, one of us always took the ownership to take it on. This allowed for us to make progress continuously in an agile approach and increased each of our participation and commitment to the project.

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

One thing we would have done differently is to be more on top of our schedule. Although we initially made our roadmap so that we would finish the project a week ahead, we ended up finishing this later than our schedule by half a week. This was partly due to our overconfidence in the beginning, and also due to our commitments for other courses as finals season hit. What we could have done is to get more done in the beginning week before due date 1, so that we would allow more time for testing and making the demo later on.

Something else we would change is to read the project requirement more carefully the first time round. This is because there were many instances where, while we were developing, we kept checking back and realizing that we were overcomplicating the game rules (we thought players also gained points for building the longest road and freaked out about how we would calculate this). To improve, we should have read the requirement document top to bottom first and then clarified the specific deliverables before we dived in to save some energy and time.

Conclusion

Our group has hesitated in the beginning to commit to a larger scale group project especially during the finals period. However, there is a lot we learned through building Constructor, whether it's about technical C++ conventions, OOP design and knowledge, or working in a team and creating this from scratch. Despite the long nights of documenting and even longer nights of debugging, no regrets for our choice. Kudos to a great team :)