Final Project: Constructor

Sahal Sajeer Kalandan | ssajeer@uwaterloo.ca | 20870961
Alec Finkelstein | afinkels@uwaterloo.ca | 20890582

Introduction

We decided to work on the Constructor project for this final project. As avid enjoyers of Settlers of Catan, we thought that compared to any other project, we could understand the data structure and the playstyle of Catan. Subsequently, to us, this meant that we could have the best grasp on what kind of relationships we should develop within each of our classes for the game of Constructor.

Overview and Design

1. Data Structure (completed by Sahal)
    a. Every vertex has: 2-3 edges incident, 1-3 tiles incident, 2-3 vertices adjacent
        i. Each vertex is a kind of Observer
        ii. Each vertex can have a player associated with them
    b. Every edge has: 2 vertices incident, 1-2 tiles incident, 2-4 edges adjacent
        i. Each edge can have a player associated with them
    c. Every tile has: 6 vertices incident, 6 edges incident
        i. Each tile is a kind of Subject
        ii. Each tile contains a boolean field that indicates whether or not geese are currently on them
    - The relationship reflected by these objects is aggregation towards each other (in a cyclical manner)
    - The relationships between each vertex, tile, and edge are set using text files containing their information called (edge/vertex/tile)info.txt. These files are read when the game is created to establish these relationships to facilitate allowing the board to work the way the Catan board works.
2. Board Implementation, including reading in and printing (completed by Alec)
    a. The Board owns 72 edges, 54 vertices, and 19 tiles
    b. The Board is responsible for printing the state of the tiles/edges/vertices and updating them
    c. The Board updates the state of each vertex/edge/tile
        i. For vertices, the board creates a building object and assigns it to a Player. The building object is a concrete decorator for Vertex and varies with what it is (Basement, House, Tower) and the amount of resources it can give a player (1, 2, 3). It takes the place of the previous vertex on the board and holds the previous vertex as its component.

ii. For edges, the board creates a road object and assigns it to a Player. The road object is a concrete decorator for Edge and exists only in that form (there are no other variations). It takes the place of the previous edge on the board and holds the previous edge as a component.
iii. For tiles, the board creates a tile object and holds the previous blank tile as a component. We ensure that this is done at the beginning of the game so that the Board is aware that these objects have been changed and returns the decorated Tiles before we establish our adjacency relationships (more in the next section)

d. The Game class owns a Board and a vector of Player pointers and is responsible for reflecting the changes from the command line or player actions onto the board (more in the next section)
i. The structure of letting the Game control the Board, and letting the Board manage the Tiles/Vertices/Edges enforce the Non-Virtual Idiom so that the state of the Board cannot change unless certain conditions are met, and the Board has complete control of how the objects are designed and manipulated.

3. Observers/Decorators for Data Structure (completed by Sahal)
a. Tile is an abstract base class, with a concrete base class BlankTile and an abstract TileDecorator class, of which the following tiles inherit: Wifi, Glass, Heat, Brick, Energy, Park (we will call these Resource tiles)
i. At the beginning of the game, depending on the arguments passed to the client, the Game class has multiple methods of generating the board. The Game class can load the board with a layout given in a file, or even load an entire game if called with a save file. In this case, it will load the save file and the current (game/player) state. If the Game object is told to create a randomized Board, it will look to see if a seed has been given to generate a random Board. Otherwise, it will use a pseudorandom method relying on the current time at runtime to generate the random values required by the game.
ii. Once the Game knows how to generate the board, it will tell the board object to generate the board given the arguments it provides. From here, the board object creates new resource tiles and passes the previous tile as a component to the resource tile.
iii. By doing this, we ensure no conflicts with incorrect information from tiles when we set adjacency relationships and now the board truly reflects the blank state of the Catan board when you start the game.
iv. The virtual methods from Subject are inherited by the Tile class as they will not change in implementation. The subject class however has a string field called type which will be important for letting the vertex observer know from what tile they are being notified.

b. Vertex is an abstract base class, with a concrete base class BlankVertex and a concrete decorator called Building
i. Rather than having to go through the process of creating more memory resources that we have to manage, it was a mutual decision that the addition of a building should only have to replace the blank vertex on the board object and hold that blank vertex as its component instead.

        ii. Although this slightly differs from the conventional style of the decorator pattern, this process would actually be easier than having to create different kinds of concrete decorators for different kinds of building objects. Since we don't have to worry about a house being demolished in the game, improving residence should just involve changing what we display and the amount of resources the building can now give.
        iii. When a Building object is created, it takes the vector of incident Tiles and calls the Observer's attach method to those tiles. When destroyed, it ensures that it detaches from all tiles it is incident to. The notify method is changed in this scope and calls a player
    c. Edge is an abstract base class, with a concrete base class BlankEdge. Since there is only one kind of decorator for Edge, we call this class Road, this is concrete and it inherits Edge as well
        i. This implementation is similar to our Building implementation, although since the only current necessity of roads is to occupy edges and help build more, it has fewer fields and therefore acts almost purely decoratively.
4. Factory for Dice (completed by Alec)
    a. Dice is an Abstract base class for concrete subclasses LoadedDice and FairDice. The Game class creates the Dice object and chooses/changes what kind of die we use based on commands from the client on runtime
        i. This implementation works exactly as expected from this project's guidelines
        ii. It is not owned by the Game object akin to how Catan may normally work, the die is independent of the game; they exist to sometimes facilitate a player turn and pass a value that the board utilizes.
5. Implementation of resource allocation for Players (completed by Alec)
    a. Each Player works kind of like an observer of different edges and vertices
        i. Players are given a vector of resources, a dictionary of integer keys and string values to indicate where their buildings are and what kind of buildings they are, and a vector containing the indices of various roads they contain. They also hold the number of building points that they have
        ii. Players are responsible for updating their resources, but the existence of the roadIndices vector and the buildingDict dictionary facilitates the idea that Players are made aware of what they build rather than owning it, but in order to build they must have the proper amount of resources.
    b. An enumeration is created for the different kinds of resource types
        i. A vector of this enumeration is created to act as the "cards" that the player has. A method (allocate resources) is added that can modify the number of resources a player can gain or lose based on what happens during the turn (more in the next section)
    c. Upon tile rolls, based on the players' buildings, resources are allocated for all buildings on that tile

        i. The allocateResources method is called from the game object to ensure that the players are allocated their resources. The method adds/removes the given resource type from the player's resource vector. More on how we decide what player gets/loses which resource and how much in the next section.
    d. Geese rolls will randomly discard the players' resources if they have more than 10 resources or if they are chosen to be stolen from
        i. The Geese mechanic is implemented as a boolean method. When a 7 is rolled, the player can choose which of these tiles will be activated as a goose tile and can also decide who to steal from based on who has buildings there. The allocateResources method will be used to allocate/deallocate these resources to each player respectively. The stealing mechanic is managed in the Game object where the Game is notified from main.cc who the player wants to steal from, and then chooses a random resource to steal from the unfortunate player, and allocates that resource to the current player.
        ii. When a player has more than 10 cards and a 7 is rolled, a player method is called where a random amount of resources are lost from the player's resource vector.
6. General Turn Structure/Player Mechanics (completed by Sahal)
    a. A player has an order to their actions: Roll, Trade, Build
        i. When a player rolls, a dice from main.cc generates a random value that then tells the game object what that roll is.
        ii. When a number that is not a 7 is rolled, the game object tells the board object to search for all resource tiles that have a roll value associated with that number and then tells each of those tiles to notify their observers, ensuring that they pass their resource type so that the observers know which of the tiles they are observing is notifying them. When the Building observers call their notify method, they convert the resource string to an integer and call the allocateResources method for the player associated with that building. This dictates how resources are generally managed and allocated throughout the game.
        iii. If a 7 is rolled, the current player can choose which of the tiles to turn into a geese tile so that it generates no resources. This will be done by telling the board to change the state of that tile so that there are geese on it. Now, whenever the board object searches for all resource tiles that have an associated roll value, if the tile also has geese on it, the board will refrain from telling the resource tile to notify its observers.
    b. Trading will be implemented so that the player can make as many trades of different quantities and resource types
        i. Trading is handled by the Game object, where integers are passed that represent the index of the other player. and produces prompts to which the other player can accept/decline. Once again, the allocateResources method is used to facilitate this
    c. Building is implemented so that a player cannot build a building unless it has at least 2 adjacent roads (or it is the start of the game) and can only build a road if it is adjacent to a vertex containing a building that the player has, adjacent to another road that player has, but it cannot build in such a manner that it passes through another player's building.

7. Harness Creation (completed by Alec)
    a. In main.cc, 4 players are created, a Game is created and so are 2 die (loaded and fair; we decide which one we want to use on runtime)
    b. This is responsible for creating the game object, loading a save file, and processing the command line arguments
        i. Command-line arguments are passed as parameters to which the Game object is notified on how to design the board and whether or not the players already have resources.
        ii. The Game object checks if a Player has met the necessary requirements to win the game upon completion of a turn. If they have, it tells the main harness that the current player has won, and can now handle the process of starting another game or exiting the program.
        iii. The Game object is dynamically created so that if players wish to start another game, the game can be reset as with the player's data so that another game can be played. The game will assume that the same board will be used, however.
    c. When a game ends abruptly (end of file is encountered), it tells the Game object to save the current game state to a savefile so that it can be loaded into play and completed later

Resilience to Change (describe how your design supports the possibility of various changes to the program specification)

Many of the aspects of our program are remarkably flexible. The way we implemented different tile types allows for new tiles to be created easily, all you would have to do is change a couple of lines and add values to an enum type. The same goes for different player amounts. You could even theoretically change the player count to allow the game to be single-player or decide the number of players at runtime! The use of the modulus would be crucial in these scenarios. Since we keep track of players through rounds and just reset their data each game we could also have a situation where you could run tournaments in a single program run, with the win count of certain players being kept track of by a field of the Player class, and player selection happening before each game. You could also easily change the program to allow for different types of buildings, again just requiring an updated enum type and some line changes. The tiles themselves utilize a decorator pattern, thus even though we only stack a single concrete decorated tile type on top of a blank tile, tile stacking could be easily implemented allowing for multiple/a vector of tiles depending on what is desired. Similarly, the number of geese could easily be increased, allowing for multiple flocks of geese to be on the field at once. As well as these examples, because of how we create adjacencies using a custom info text file for each of edges, tiles, and vertices we could implement “wormholes” that set elements as adjacent to other elements that they would physically not be adjacent to. All of these aspects have in common the design idiom of decoupling to thank for how little resilience to change there would be for these situations. These classes are fairly isolated and operate with interfaces allowing for the program to rely on them to handle internal calculation and memory management.

Although our program is remarkably pliable, there are invariant aspects to it that we believe to be either crucial to the game of Catan we wanted to implement for this project or were simply side effects of our design decisions. This includes the number of tiles and layout of the board - it is mostly non-algorithmic in our code, meaning that the game will always have 19 tiles displayed in the same way. The same goes for the situation where you would want multiple return types for a single tile. It is integral to multiple parts of our code that tiles will return a single type and counting in multiple would require a fair amount of restructuring. Other than this, changing what is adjacent or even on the same board mid-game or removing adjacencies altogether would mean that our setup of using shared resources would be invalidated and we'd have to make major design changes to accommodate this change. Overall, there are a few aspects that we believe to be invariants but we use them as the foundation of our structure and believe this to be acceptable.

Answers to Questions (the ones in your project specification)

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?

For this feature, it made sense to use a decorator design pattern. We implemented a simple tile parent class and decorated it as we read the tile info at runtime. It also makes it easy to get information from the tile, as we have accessor methods that obtain component details within the decorator. We decided to use this design pattern as it makes it the easiest to accomplish the tasks laid out in the project details. However, we also came to some difficulties with the levels of object creation in our decorator pattern and to what extent we needed it. We realized that road and building objects will remain once constructed and do not leave until the end of the program, so we did not need to add multiple layers of a Decorator pattern for it. Moreover, the geese proved to be one of the hardest things to implement surprisingly, and ultimately we decided on letting a tile be toggleable from either having Geese on it or not as a workaround.

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?

For this feature, we thought it made sense to use the factory design pattern. Our main.cc in this case is the factory, and we can simply choose which kind of Dice we want to use, whether it's Loaded or Fair. We implement a factory method that creates our Dice and then based on whatever command the client passes, we either set the current dice to either Fair or Loaded Dice. If we choose to change the Dice type mid-game, we can do so quite easily using this functionality without having to worry about unmanaged resources.

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

If we wanted to implement rudimentary AI functionality, we think it makes sense to use the template design pattern. Rather than overriding our Player class completely, we can keep some of its functionality and introduce virtual helpers that can change certain plays that the player makes based on the playstyle we want to implement for the AI. For example, we can make a selfish AI that chooses to never trade during a turn and always chooses to steal from the player with the least resources during the Geese roll. Alternatively, we could make a selfless AI that always looks to trade to help the other players and places the Geese tile on Park to not impact anyone.

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?

This could be implemented with the decorator pattern. The way our current tile structure works, we are able to place a resource tile on top of an empty tile. What we imagine is that if we wanted to obtain multiple types of resources, our tile can now hold a vector of resource tiles instead of just one, and that way, if that tile is rolled we tell all the resource tiles in that tile to notify buildings on that tile and allocate resources accordingly. If we wanted to limit the number of resources the tile can provide, we can add a static field to each resource decorator so that it doesn't provide any more resources if this amount reaches zero, or that it prevents giving too much of a resource below a certain threshold.

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

Unfortunately, we did not get around to utilizing exceptions in our project. This is because by utilizing NVI and having our Game and Board classes handle mostly everything, we are able to detect when input received violates invariants we set with our data structure or the capabilities there are for modifying them. For the most part, we believe we have strong to no throw guarantees, but there are certain places we wish we could add them. For example, if we had more time to work on the project, we would make use of exceptions so that when the end of file command is given and we tell the program to terminate, we could have it throw if there is unmanaged memory that we need to resolve. We could also implement exception safety to help us better facilitate how the game resets and how the players reset when the game restarts.

Final Questions (the last two questions in this document).

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

This project taught us both so much about the reality of developing large programs in teams. Firstly, we believe the most important part of developing software in teams is a strong collaboration on the design of the entire program. We spent a lot of time at the beginning of the project just discussing the potential logic of the program and how those issues should impact the design of our program. We wanted it to be clean and easy to follow and as compartmentalized and self-reliant as possible; after all, that is the beauty of object-oriented programming. Another lesson we learned is a lesson that can never be overstated: things will go wrong. What we thought would work didn't, and large key sections of our code had to be redesigned and rewritten several times until we were satisfied. Something that was also made apparent to us was that there's never too much time to set aside for testing and designing. Working together with your partner to fix crucial errors in your judgment about how to design the program is a costly time investment, even though it does pay back dividends. We discovered so much about how we had to design the program by watching it fail, testing to find why, and scheming together at a whiteboard to think up the best way to fix it. You learn so much about developing software, and coding as a whole while working on a project that you can apply in the future; it is truly the best way to learn programming.

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

If we were given the chance to start over, we would probably still adhere to our approach in designing the logic and data structure of this project. If there was something we could change, it would probably be to work in a much smaller scope, perhaps 1-4 tiles to try and focus on the core of what is happening with each tile/vertex/edge construction and what issues they may have caused. Furthermore, a better understanding of smart pointers would have enabled us to make better memory decisions from the start in such a way that we didn't have to find workarounds. All in all, focusing on the fine details of the code would have really made our code even more robust than it already is.

Conclusion

With this, we hope to have illustrated the relationships between our classes and the overall design of our implementation of the game of Constructor. From the data structures to the smart pointers, to the boolean geese that will surely haunt us after this project concludes. We decided to do this because we love Settlers of Catan but during the process of designing and implementing Constructor we find our understanding of the game more detailed and in-depth than before. We love the game of Catan, and we hope that you'll enjoy our computerized ode to it.

Constructor UML

afinkels | ssajeer | August 1, 2023