Overview (Describe overall structure of project)

For our project, a game class is designated the responsibility of handling elements of gameplay directly seen by the user. A game instance is initialised with pointers to four players, two dice: one loaded, one fair, a grid and a seed. The game object calls a "play function" which then handles placing the initial buildings, changes to the board made by rolls and possible actions made by players. When a player achieves the required number of victory points, the game terminates.

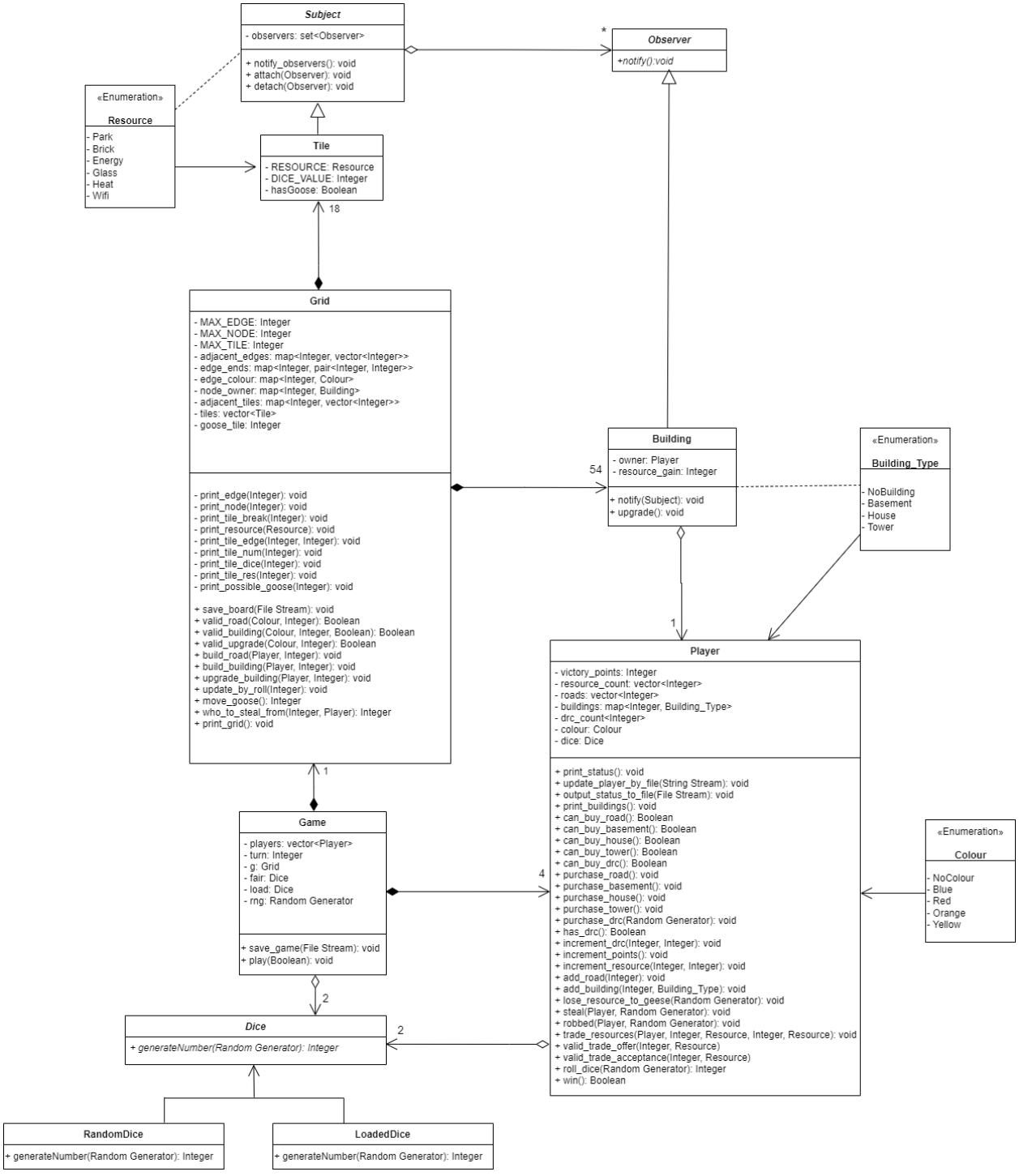
A player class encapsulates all the information and actions of the players. This includes their game colour, the type of dice they are using, the types of buildings they own and how many resources they have. The class provides getters and setters, functions for purchasing items in the game, as well as the ability to steal.

A permitted grid may be constructed randomly or from a file and is where the gameplay takes place. It contains information regarding all aspects of the board state by representing the board as a graph structure with adjacency maps between edges to nodes and vice versa. The grid also keeps maps between nodes to colour and edges to a colour. Besides storing information about the board state, it contains all necessary core functions, as well as helper functions needed to apply changes to the board.

A core part of a board is the tiles. The grid keeps a vector of pointers to all the tiles and a map between nodes to adjacent tiles. Information about the board's tiles, such as the resource type, its corresponding roll value and whether or not the geese reside there, is encapsulated by a tile class. Tiles extend a subject class which notifies its observers whenever its roll value gets rolled. Each tile is observed by several building objects which are built on an adjacent node. Each building maintains a pointer to the player who owns them. The main function is to increment the resource of its owners by a specific amount depending on its type.

Finally we created a game class which stores the 4 players and the grid. It contains a function "play" that runs the game and takes commands from users as input. It serves as an interface between the players and the board so that their decisions affect the state of the board. It also serves as an interface for player-player interactions such as stealing, trading and DRCs (explained in the bonus feature section). Finally, it is the owner of the only random number engine which the Player and Dice class use when performing random actions.

Updated UML


Subject
- observers: set<Observer>
+ notify_observers(): void
+ attach(Observer): void
+ detach(Observer): void
*
Observer
+notify():void
«Enumeration»
Resource
- Park
- Brick
- Energy
- Glass
- Heat
- Wifi
Tile
- RESOURCE: Resource
- DICE_VALUE: Integer
- hasGoose: Boolean
18
Grid
- MAX_EDGE: Integer
- MAX_NODE: Integer
- MAX_TILE: Integer
- adjacent_edges: map<Integer, vector<Integer>>
- edge_ends: map<Integer, pair<Integer, Integer>>
- edge_colour: map<Integer, Colour>
- node_owner: map<Integer, Building>
- adjacent_tiles: map<Integer, vector<Integer>>
- tiles: vector<Tile>
- goose_tile: Integer
- print_edge(Integer): void
- print_node(Integer): void
- print_tile_break(Integer): void
- print_resource(Resource): void
- print_tile_edge(Integer, Integer): void
- print_tile_num(Integer): void
- print_tile_dice(Integer): void
- print_tile_res(Integer): void
- print_possible_goose(Integer): void
+ save_board(File Stream): void
+ valid_road(Colour, Integer): Boolean
+ valid_building(Colour, Integer, Boolean): Boolean
+ valid_upgrade(Colour, Integer): Boolean
+ build_road(Player, Integer): void
+ build_building(Player, Integer): void
+ upgrade_building(Player, Integer): void
+ update_by_roll(Integer): void
+ move_goose(): Integer
+ who_to_steal_from(Integer, Player): Integer
+ print_grid(): void
Building
- owner: Player
- resource_gain: Integer
+ notify(Subject): void
+ upgrade(): void
54
«Enumeration»
Building_Type
- NoBuilding
- Basement
- House
- Tower
1
Player
- victory_points: Integer
- resource_count: vector<Integer>
- roads: vector<Integer>
- buildings: map<Integer, Building_Type>
- drc_count<Integer>
- colour: Colour
- dice: Dice
+ print_status(): void
+ update_player_by_file(String Stream): void
+ output_status_to_file(File Stream): void
+ print_buildings(): void
+ can_buy_road(): Boolean
+ can_buy_basement(): Boolean
+ can_buy_house(): Boolean
+ can_buy_tower(): Boolean
+ can_buy_drc(): Boolean
+ purchase_road(): void
+ purchase_basement(): void
+ purchase_house(): void
+ purchase_tower(): void
+ purchase_drc(Random Generator): void
+ has_drc(): Boolean
+ increment_drc(Integer, Integer): void
+ increment_points(): void
+ increment_resource(Integer, Integer): void
+ add_road(Integer): void
+ add_building(Integer, Building_Type): void
+ lose_resource_to_geese(Random Generator): void
+ steal(Player, Random Generator): void
+ robbed(Player, Random Generator): void
+ trade_resources(Player, Integer, Resource, Integer, Resource): void
+ valid_trade_offer(Integer, Resource)
+ valid_trade_acceptance(Integer, Resource)
+ roll_dice(Random Generator): Integer
+ win(): Boolean
1
Game
- players: vector<Player>
- turn: Integer
- g: Grid
- fair: Dice
- load: Dice
- rng: Random Generator
+ save_game(File Stream): void
+ play(Boolean): void
4
«Enumeration»
Colour
- NoColour
- Blue
- Red
- Orange
- Yellow
2
Dice
+ generateNumber(Random Generator): Integer
2
RandomDice
+ generateNumber(Random Generator): Integer
LoadedDice
+ generateNumber(Random Generator): Integer

Design (Specific techniques used to solve design challenges in the project)

The first design challenge we had to face was to implement how players received resources from tiles. We ended up settling on the observer design pattern. We created a Tile, Building and Player class where a Tile is a Subject, and a Building is an Observer. A Building object contains a pointer to a Player (which was its owner), and Building's notify function was just to increment its owner's resources based on the Tile's resource type and based on what type of building it is (Basement, House, Tower). It did so by calling Player's public function increment_resource(). We also considered the possibility of making the Player class the observer, but that was infeasible because it doesn't keep track of what building is built so it won't know if it should gain 1,2, or 3 resources.

The next natural question for us was how were we planning to implement the fact that there are different types of building which yielded different amounts of resources to its owners. We did consider using the Decorator class to implement this feature since we are dealing with a scenario when an object's functionality changes during runtime. However, we decided it was much simpler to create a variable resource_gain in the Building class that keeps track of how many resources it produces (1 for Basement, 2 for House, 3 for Tower). Everytime we tried to improve a Building, we simply had an upgrade function that incremented the resource_gain parameter. As a result, Building's notify function simply incremented its owner's resource by the value resource_gain.

The other main design challenge of this project is to keep track of the status of the board. We also needed to be able to use this information to determine where a player is allowed to build roads and edges. To solve this problem we simply created a grid class. To keep track of what buildings have been built and what roads have been built, we used 2 maps: node_owner (maps node_index to a building) and edge_colour (maps edge_index to a Colour) that mapped each index to the corresponding owner. Initially, in the grid's constructor, these are initialised so that they map every index to a nullptr. These are updated when we call grid's build_building and build_road functions. Node_owner maps index to a building instead of a Colour because we also needed the functionality to be able to improve a building at a node_index. To determine where players are allowed to build, we had to use a data field to keep track of what edges are adjacent to each node (this maps node_index to a vector of edge_index) and a data field to keep track of what nodes are adjacent to an edge (this maps edge_index to a pair of node_index). These were hard coded to represent the status of the board. We then had 2 functions valid_road and valid_build that used the data in these data fields to determine if a player was allowed to build a road/building on an edge/node.

Resilience to Change (How your design supports the possibility of change)

The simplest change that our design supports is the possibility of having different numbers of players. This is easily solved by changing our vector of players in the Game Class to contain the new numbers of players. If we decide to have more than 4 players we would also have to add more colours to the Enumerated Colour class, but that is also a very quick change. Another small change that our implementation supports is (if for some odd reason) we wanted to have different types of dice in addition to a random or loaded dice. As discussed in the question section below, the use of the factory design pattern allows us to add new dice with different number generation and not change many other aspects of our code.

A big change that our code accommodates is if the board changes its structure, this could be if the number of tiles changes, if the shape of each individual tile changes (perhaps pentagon, or rectangular or octagon), or perhaps if the shape of the overall board changes. The only change that we would have to make in our code is to change our data fields that map nodes to their adjacent edges and edges to their adjacent nodes and modify the print function. Even though this may seem like a lot of work, it is an acceptable amount as our program needs to have access to what the grid looks like, and the two mappings (node to adjacent edges, and edges to adjacent nodes) are what define a graph. Besides this however, the rest of our code would remain untouched and the program will be able to run the game without issue. This is reflective of the high cohesion and low coupling of our code. Each module/class has their own responsibilities and only the Grid class needs to be altered to accommodate for different types of boards.

Another possible change would be changing the types of resources. Since we store the types of resources in an enumerated class, we can simply add or remove resources from the class. Moreover, every player has a vector of resource integers listing the quantities of each resource in their possession. We would also need to change the initial size of the vector to accommodate for the different types of resources. Aside from that, no other changes would need to be made. Furthermore, if we wanted to change the names of the resources, buildings, or colours, we would simply need to change their names in their respective enumerated classes, the names of the player pointers, and the output operator overload.

Furthermore, we could also easily change the costs of roads, basements, houses, and towers. In resource_cost.h, we have constants listing the number and types of resources required to build every structure. Our functions that build structures call those constants to check whether the player has the appropriate number of resources. However, by simply changing the costs in the resource_cost, we would have completely changed the cost of roads and buildings throughout the program.

We could also change the number of geese in the game. Every tile object has a boolean data field stating whether or not there is a goose present on the tile. We could simply add more geese by changing some starting tiles to have geese through the boolean data field. When rolling a 7, every player with 10 or more resources would still randomly lose half their resources. However, we would have to make slight changes to the function that manages moving the geese. Instead of previously searching for the singular tile that has the geese, we would prompt the user to select which geese to move, and then check whether the tile has geese by checking its boolean data field. The actual code responsible for moving the geese would remain the same.

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?**

   The template design pattern can be used to implement this feature. We are using the template method to generate the grid, and we are implementing operations to set up the resources and the dice roll for each tile. We decided to not use the template design pattern because the way the board state is provided via file input makes it more natural to set up both values for the tile at the same time. When reading input from the file, we are given the dice number and resource for the tile at the same time so it is unnatural to read all the resource values first, and then all the dice roll numbers. If the state in the file were presented by printing all the resource values first, and then all the dice numbers (or the other way around), the template design pattern may be suitable. [same as DD1]

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 factory design pattern could be used to implement various types of dice. In this case, we could have an abstract Dice class that is inherited by the concrete Loaded Dice and Fair Dice classes. We could then have a pure virtual function to generate the dice roll (which would differ depending on which concrete Dice object we use). Implementing this in main would simply involve instantiating one Loaded Dice pointer, one Fair Dice pointer, and a Dice pointer (which is actually used to generate numbers). When the player uses the command “load” or “fair”, we can assign the Dice pointer to point at the loaded dice or fair dice object accordingly. When passed the roll command, we would just need to call the generate dice roll function.

Contrary to our original answer, we did end up implementing this design pattern and using it as described in the above paragraph. The main reason as to why we changed our mind is because we didn't read the assignment details carefully and didn't realise that each player had their own individual set of dice (we thought it was one dice shared among the 4 players). Creating this class made it clear the type of relationship between the dice and player. In terms of cohesion, we also felt that it just made sense to put all the dice roll generation in one file and not mix it with the Player Class.

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. hexagonal 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?**

   One possible implementation of this feature could be with the factory design pattern. Our program utilises a grid class that contains the private member variables which detail information regarding the structure of the board. These include the following: number of edges (road locations), number of nodes (settlement locations), as well as an adjacency list of each node to edge and edge to node. These variables are enough to represent any graph. Thus, any alterations to the board, such as hexagonal tiles and different-sized boards can be represented through alterations to these members. With the factory design pattern, we can instead make the grid an abstract interface containing all the shared methods common to all layouts. Each type of board is a class that extends our abstract grid interface with a unique constructor to initialise the information describing its layout. It can also override certain other methods like print() that require changes from board to board. Finally, a creator class with a static factory function will determine the concrete board object to be used depending on its parameters. Overall, this choice of design allows us to avoid tight coupling between a specific board type with the rest of our code. [same as DD1]

4. **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 class would be excellent to implement this feature. We would start with an abstract Tile class with the data fields to store the tile value, the types of resources obtained from the tile, and whether the goose is on the tile. We would then create a base tile class and a decorator class that both inherit from Tile. The base tile class would be the

Park, where no resource is obtained. Extending from the decorator, we would have a class for each resource that determines the quantity of resources produced. During runtime, if we want to add more abilities for each tile, we would simply continue to wrap it with more decorator objects. As per the decorator design pattern, if a tile needs to generate resources, we would simply iterate through the linked list of concrete decorators and alter the resource output of the tile.

However, in our core game, we found it unnecessary to add a decorator class. Since the user only gains one resource if the tile value is rolled, then we simply used an enumerator class to store the different types of resources that could be obtained.

5. **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**.

   We did use exceptions but not in the way described in our DD1 answer. In terms of invalid commands, whenever we prompted for input from the user, we simply dealt with this case by having an else block and then outputting "Invalid Command." Instead, we used exceptions to handle scenarios when reading in standard input fails (e.g inputting a character when expecting an integer or reaching end of file). Using a try-catch made a lot of sense, since we are required by the assignment to save the game's state if we reach end-of-file. We simply wrapped a try-catch around our Game class' play function, and if we caught an ios-failure we simply called the Game class' save_game function. We implemented this because we felt that it was an extremely fast way to deal with this issue while also being able to implement the feature to save game if an ios failure occurs at any point during run time.

Extra Credit Features (What you did, how they were challenging, how you solved them)

One extra credit feature (which was initially only meant as a quality of life improvement for us while testing) we implemented is the fact that when we print out the board and the builders' colours, we colour the text. The main challenging feature was making sure that the text was coloured in all areas of the game. We solved this problem by creating an enumerated Colour class, overloaded the operator<< function and gave the Player class a Colour parameter. Everytime we needed to print out a player's colour, we simply called the overloaded output operator. On a side note, specifically figuring out how to print out orange text took me at least half an hour of digging through resources online :).

In terms of trading, we felt that it was inflexible to only have players be able to trade at a 1:1 ratio. So we decided to have players be able to offer trades with varying ratios (e.g 2 Bricks for 1 Energy, or 2 Glass for 3 Wifi).

We've all played Catan before, so we were disappointed when we learned this version of Catan didn't have DRCs (Development Resource Cards). In addition to the standard set of commands available to the player during their turn, we decided to add the ability to buy and use DRCs. If a player has enough resources, they may elect to purchase a DRC, and the game will notify the player which random one they have received. To use one, we added five commands specific to each DRC and ensured that the player only uses it if they a) own one and b) have not used any DRC this turn already. The "Knight DRC" permits a player to move the geese to a new tile and steal resources from a player who owns a building adjacent to said tile. The "Year of Plenty DRC" allows them to select a resource and receives 2 of them. The "Monopoly DRC" prompts a player to choose a resource and steals all copies of them from other players. The "Victory point DRC" gives the player a building point. Finally, the "Building roads DRC" allows the player to build two roads in valid locations.

One challenge of implementing this feature was determining who gets responsible for facilitating it. We initially thought it was best to give the ability to use a DRC within the player class since the amount of DRCs a player class is specific to them. However, the monopoly DRC requires a pointer to every other player, and we decided it didn't make sense to pass in a list of players or add parameters to the class. We wanted the "Player class" to only handle activity within itself and let the game class resolve interactions between players and the board. Thus we added getters and setters for the amount of each DRC a player has. Then we moved the handling of DRCS to the game class as a part of the game loop. To ensure that only one DRC gets used per turn, we implemented a boolean which begins as false every round and is set to true whenever a DRC command successfully runs.

Another challenge was how best to utilise the existing code base so that a minimal amount of additional code is required. For example, the "Knight DRC " completely reuses code associated with rolling a 7, and the "Building Roads DRC " completely reuses code with building a road minus costing resources. The "Victory Point DRC" and "Year of Plenty DRC" both reuse different setter functions in the player class. Finally, the "Monopoly DRC " required the addition of a new getter function for the exact number of a specific resource but reuses setter functions to change the number of resources a player has.

Final Questions (Last 2 questions in the document)

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

As with any group project, it was important that every group member was on track and communicated their progress. From the beginning, we discussed how we wanted to implement various features and relate our classes together. Then, as we were coding, we had to write legible code and update each other on how we completed our task. This made it easier for us to read our code and debug when necessary. However, we could also have planned out how to implement certain features more thoroughly before starting. As shown by the difference in our first UML draft and our final UML draft, we had to add several more functions and data fields after completing more and more features. This would have made it easier to work on our project and made our code more clean.

Another lesson we learned was to frequently ask questions. When any of us were unsure about how to add some feature, we would ask for each other's opinions on how to implement the code. After all, if someone was stuck on their task, it would slow down the entire group's progress. So it was important that we sought out each other's help when needed.

2. **What would you have done differently if you had the chance to start over?**
   We definitely all wished that we had a better understanding of how git worked before we had started the project. We were familiar with pushing and pulling code, but we didn't anticipate the difficulty that would arise with merge conflicts. We were all keen on developing our git skills because we understand that it's industry standard, however we felt that there wasn't enough time to properly learn git by the time the project had started. Instead our solution was just to always have one person coding at a time, have them push their code, and then the next groupmate continues the work after pulling. This wasn't ideal however as there were many times where we wished to code together but we were too scared of merge conflicts. In hindsight, we all wished that we learned git more thoroughly before the project even began. Another option that we wished we considered was to use Liveshare. We all think it would've streamlined the collaborating experience for us.

   Something else we could have done differently was to modify the UML after every contribution, instead of rewriting it at the end of the project. This would make it easier for the group members to see what was completed, added, or changed since the last update.