CS246 Spring 2023 Project – Constructor

C. Kierstead, B. Lushman, M. Petrick

- DO NOT EVER SUBMIT TO MARMOSET WITHOUT COMPILING AND TESTING FIRST. If your final submission doesn't compile, or otherwise doesn't work, you will have nothing to show during your demo. Resist the temptation to make last-minute changes. They probably aren't worth it.
- This project is intended to be doable by three people in two weeks. Because the breadth of students' abilities in this course is quite wide, exactly what constitutes two weeks' worth of work for three students is difficult to nail down. Some groups will finish quickly; others won't finish at all. We will attempt to grade this assignment in a way that addresses both ends of the spectrum. You should be able to pass the assignment with only a modest portion of your program working. Then, if you want a higher mark, it will take more than a proportionally higher effort to achieve, the higher you go. A perfect score will require a complete implementation. If you finish the entire program early, you can add extra features for a few extra marks.
- Above all, MAKE SURE YOUR SUBMITTED PROGRAM RUNS. The markers do not have time to examine source code and give partial correctness marks for non-working programs. So, no matter what, even if your program doesn't work properly, make sure it at least does something.

1 The Game of Constructor

In this project, you will implement the game Constructor, which is a variant of the game Settlers of Catan (often called Settlers for short), with the board being based on the University of Waterloo. If you are unfamiliar with Settlers, please see `https://www.youtube.com/watch?v=8Yj0Y3GKE40` or `https://www.catan.com/service/game-rules` for an overview. It may also be beneficial to play a game or two of Settlers to become familiar with the gameplay. Note that the rules of Constructor vary from the rules of Settlers.

1.1 Gameplay Overview

You have recently decided that you are going to become a housing developer in Waterloo. You want to build housing for the many students who attend the University. There are three different types of residences you can build. For each residence built, you will earn a building point towards winning the game. Aside from that, residences will influence which materials you receive, and where you and other builders will be able to build in the future.

During the setup of the game, each player selects starting vertices. Once the game begins, each builder takes turns rolling the dice to determine which tiles produce resources. After rolling, the builder who rolled the dice has the opportunity to build roads and residences using their resources.

The goal of the game is to be the first of the four players to obtain 10 (or more) building points.

2 Game Components

2.1 Board

The board is the visual representation of the state of the current game (see Figure 3 for a sample). The board will always display the 19 tiles present in the game, all edges and vertices, and the roads and residences the builders have completed. The representation for each of these components is described in their respective sections.

2.2 Tiles

Each tile, displayed as a rectangle on the board, represents one type of resource in the game. The middle of each tile lists the tile number, the type of resource and the tile value, in this order. In Figure 1, the tile number is 9, the resource is BRICK and

the tile value is 2. The tile number refers to where the tile is on the board. The tile value is used to determine which resources are allocated when the dice are rolled (discussed later). Additionally, if geese (See Section 2.3) are present on the tile, it will be displayed as in Figure 2. The outline of each tile is composed of the edges and vertices surrounding it.

There are 6 types of tile: BRICK, ENERGY, GLASS, HEAT, WIFI, and PARK. The Resources section (3.5) discusses how resources and values are distributed. The PARK tile produces no resources.

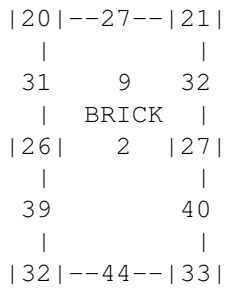

Figure 1: Sample Tile

Note that the shape of the board is symmetric, but not in a way that lends itself to developing a formula. You need some way to remember the road/residence/tile relationships, and re-build them for each game. There aren't too many possibilities, so you will either have to hard-code it, or read it in from somewhere. Either approach is fine, and you wouldn't lose marks for it.

2.3 Geese

A tile can be overrun with geese, which play a similar role to that of the robber in the original game. In the event that a tile has been overrun by geese, builders will not receive resources when the value matching the tile is rolled.

2.4 Vertices

A vertex is located at each corner and in the middle of each vertical edge of each tile. Each tile has 6 vertices. Note that most vertices are shared by multiple tiles.

In Figure 1, the vertices are numbers 20, 21, 26, 27, 32, and 33.

2.4.1 Housing Details

A residence can be built at a vertex. Only one builder can build at each vertex. Each vertex can have at most one of 3 types of residences on it. Once a builder has built a residence, they can improve the residence twice. When a residence is built, the display is updated by replacing the vertex with the first character of the colour of the owner, e.g. 'B' for "Blue", followed by the first character of the name of the improvement, e.g. 'B' for "Basement".

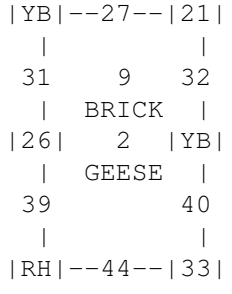

Figure 2: Tile with Geese

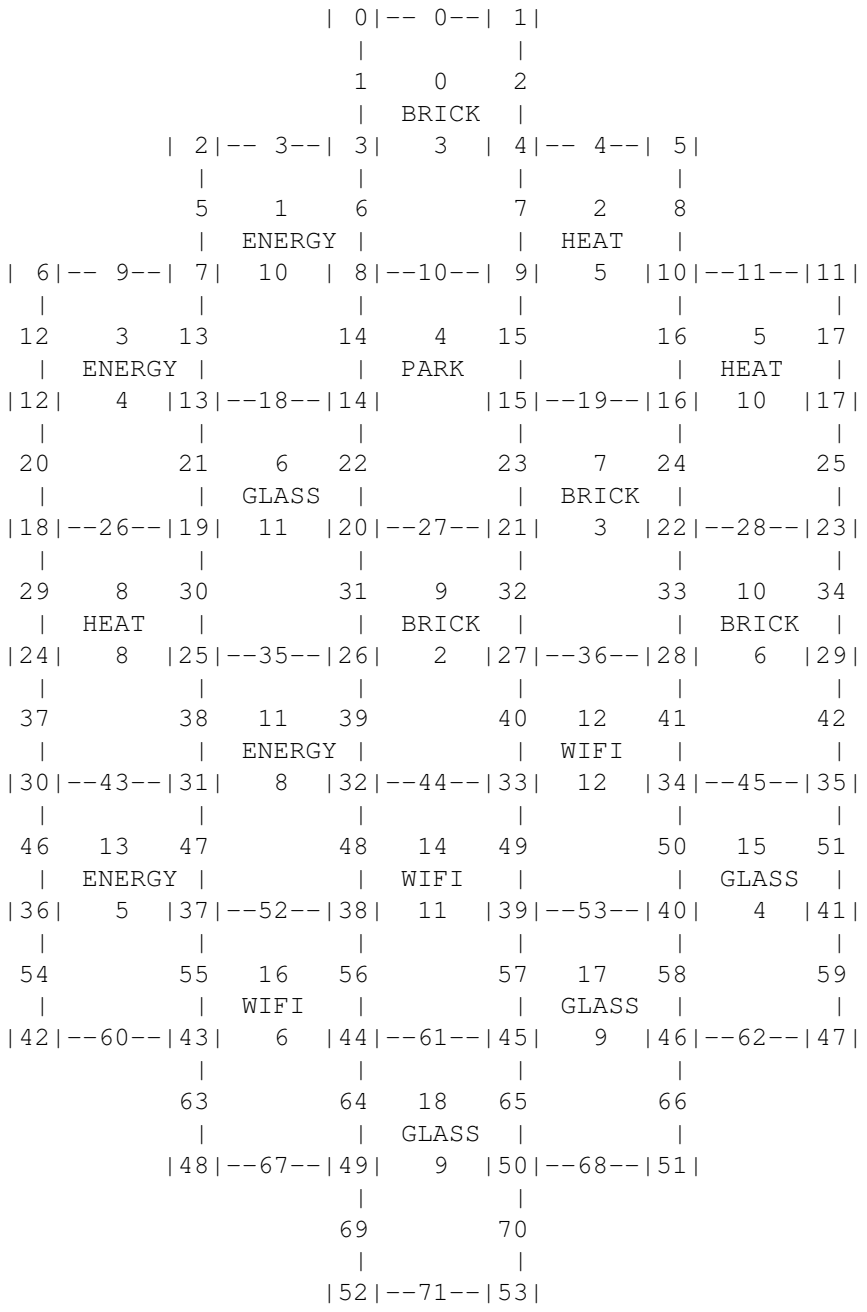

Figure 3: Sample Board

Housing must be built in the following order:

Basement (B) Building a basement gives the builder one resource when the value of any adjacent tile is rolled. When acquired, a basement gives the owner one building point. A basement costs one BRICK, one ENERGY, one GLASS, and one WIFI to build.

House (H) Building a house gives the builder two resources when the value of any adjacent tile is rolled. When acquired, a house gives the builder one additional building point (2 in total). A house costs two GLASS, and three HEAT to build.

Tower (T) Building a tower gives the builder three resources when the value of any adjacent tile is rolled. When acquired, a tower gives the builder one additional building point (3 in total). A tower costs three BRICK, two ENERGY, two GLASS, one WIFI and two HEAT to build.

If a vertex currently has no residence on it, then a builder may construct a Basement there; on the same or a subsequent turn, that same builder may upgrade the Basement to a House, and later still may upgrade the House to a Tower. Once built, a residence cannot be removed or change owners.

2.5 Edges

Edges are located at the edges of each tile, going between vertices. Each tile has 6 edges. Most edges are shared by multiple tiles.

For example, in Figure 1, the edges associated with the tile are numbers 27, 31, 32, 39, 40, and 44. Residence 38 is adjacent to edges 48, 52, and 56. If builder Red has built a road on edge 48, and builder Orange has a basement on residence 38, then Red may not build a road on either edge 52 or edge 56 (through vertex 38) unless it can approach legally from the other direction(s). In other words, you may not build through a residence of a different colour.

2.6 Roads

Each edge can have at most one road built on it. When a road is built, the display is updated by replacing the edge number with the first character of the colour of the owner, e.g. 'B' for "Blue", followed by 'R' for "Road". Note that once a road has been built on a given edge, it will stay there for the remainder of the game.

A builder can build a road only if an adjacent road or residence has been built by the same builder.

The cost to build one road is one HEAT and one WIFI resource.

2.7 Builders

All builders in the game are "human" players and controlled by the commands issued on standard input. Your game must support exactly 4 builders.

Each builder is assigned a colour at the beginning of the game with builder 0 being Blue, builder 1 being Red, builder 2 being Orange and builder 3 being Yellow.

When the status of a builder is printed, output:

```
<colour> has <numPoints> building points, <numBrick> brick, <numEnergy> energy,
<numGlass> glass, <numHeat> heat, and <numWiFi> WiFi.
```

When the buildings of a builder are printed, output:

```
<colour> has built:
```

followed by the following line for each building:

```
<vertex> <buildingType>
```

where `<vertex>` is the vertex of each building that `<colour>` owns and `<buildingType>` is the type of building (B, H or T).

2.8 Dice

Two types of dice are supported by the game: *loaded*[1] and *fair* dice. Each builder has their own set of dice to roll. When dice are rolled, any builder who has a residence on a tile with the same value as the roll receives resources associated with the tile and the residences present. When the game begins, all builders will have loaded dice.

For example, in Figure 1, if a builder owned residence 21 and the dice rolled were 1 and 1 (i.e. the total is 2), the builder will receive a BRICK resource.

2.8.1 Loaded Dice

When loaded dice are chosen, the builder is prompted with the statement:

```
Input a roll between 2 and 12:
```

If the builder's input is invalid, i.e. not an integer or out of the valid range, output the message:

```
Invalid roll.
```

followed by the previous prompt.
Once a valid roll has been entered, the entered roll is used for the turn.

2.8.2 Fair Dice

When fair dice are chosen, two numbers are randomly generated, each between 1 and 6. The sum of the two generated numbers are used for the turn.

3 Game Rules

An important part of the game is how each of the game components interact with each other. This section describes many of the rules explaining how pieces interact.

3.1 Building Residences

For a builder to build a residence, the following two conditions must be met:

- A residence may not be built on a vertex that is adjacent to a vertex with an existing residence.
- It is either the beginning of the game, in which case a residence can be built on any vertex, or they have built a road that is adjacent to the vertex.

3.2 Improving Residences

A residence can be improved if the builder has already built a residence and the residence has not been improved to a tower yet. A basement can only be improved to a house and a house can only be improved to a tower.

3.3 Building Roads

A road can be built if the builder of the same colour has built either an adjacent road or residence.

3.4 Turn Order

Once the first residences are chosen, it is the Blue builder's turn, followed by Red, then Orange and then Yellow. This sequence then repeats until the game ends.

[1] Using loaded dice is a form of cheating in games. This is the only time we will encourage you to cheat in this course.

3.5 Resources

3.5.1 Printing Resources

When printing owned resource types during the game, resources should always be printed in the order of BRICK, ENERGY, GLASS, HEAT, then WIFI.

3.5.2 Obtaining Resources

When the die roll is a non-seven number, builders receive resources based upon where they have built residences, and the types of these residences.

3.5.3 Using Resources

A builder can use any resources that they have in their possession. Attempting to complete an action that requires more resources than the builder has causes the action to fail and the message

```
You do not have enough resources.
```

to be printed.

3.5.4 Trading Resources

The active builder can propose a trade with any other builder. When a trade is proposed, the following output will be displayed:

```
<colour1> offers <colour2> one <resource1> for one <resource2>.
Does <colour2> accept this offer?
```

where `<colour1>` is the current builder, `<colour2>` is the colour of the builder `<colour1>` is proposing to trade with, `<resource1>` is the resource `<colour1>` is offering and `<resource2>` is the resource `<colour1>` is requesting.

After the prompt, player `<colour2>` responds. The response from `<colour2>` is either "yes" or "no". If "yes", `<colour1>` gains one `<resource2>` and loses one `<resource1>`, while `<colour2>` gains one `<resource1>` and loses one `<resource2>`. Otherwise, neither players' resources change.

3.5.5 Tile Distribution

The value and resource associated with each tile is read from a file at the beginning of the game. By default, the file that is read is named `layout.txt` from the current directory. (If the file isn't present, the player needs to use the `-random` command-line argument.) A command-line option can be given to change the file that will be read. The format of the file is described in the Board Layout (3.8) section.

There is also a command-line option to generate a random resource allocation on the board layout i.e. the board will always have the same shape, but the resource placement may change. When randomly generated, the board consists of the following resources: 3 WIFI, 3 HEAT, 4 BRICK, 4 ENERGY, 4 GLASS, and 1 PARK. The values on the board will have the distribution of one tile with value 2, one tile with the value 12, one tile with the value 7 (the Park), and two tiles each for the values: 3, 4, 5, 6, 8, 9, 10, and 11. Note that this makes it possible to have a board with two tiles with the same value. The tile with PARK does not display a value.

For example, suppose a 2 was rolled with the board in Figure 3. There is only one tile on the board that has the value 2 and it has the resource BRICK. The residences on the tile are 20, 21, 26, 27, 32, 33. Each builder who has completed one of these residences receives some BRICK resources. The number of BRICK they receive is dependent on the type of residence they have built (one for basement, two for house, and three for tower). If a builder has completed multiple residences on a single tile, they receive the resource(s) for each residence.

After the resources are distributed, print the following line for each builder who gained resources, in the order of builder number:

```
Builder <colour> gained:
```

followed by the line:

```
<numResource> <resourceType>
```

for each resource they received from the dice roll.
If no builders gained resources from the roll, output

```
No builders gained resources.
```

3.6 Moving Geese

Geese initially start on the Park tile since it doesn't generate any resources. When a seven is rolled, the current builder moves the geese to any tile on the board. (The geese may not be placed on the tile on which they were previously.) Any builder with 10 or more resources will automatically lose half of their total resources (rounded down) to the geese. The resources lost are chosen at random and the overall process is described below. For each builder who loses resources, output:

```
Builder <colour> loses <numResourcesLost> resources to the geese. They lose:
```

followed by the line:

```
<numResource> <resourceName>
```

for each resource lost.
The active builder is prompted:

```
Choose where to place the GEESE.
```

The builder responds with the number of any tile except where the GEESE is currently placed. The current builder then steals a random resource from one builder who has a residence on the tile to where the geese are moved.

If there are any players which can be stolen from, the following is output:

```
Builder <colour1> can choose to steal from [builders].
```

where `<colour1>` is the active builder and `[builders]` is a comma separated list of those builders, represented by their colour, who have residences on the tile where the geese were placed, have a non-zero number of resources, and are not the active player. The builders should be printed in the order of player number.

The active builder is prompted:

```
Choose a builder to steal from.
```

The active builder responds with the colour of the builder from whom they want to steal a random resource. Exactly one resource is chosen. The probability of stealing/losing each type of resource is the number of the resource the builder being stolen from has, divided by the total number of resources the builder being stolen from has.

After a resource is stolen, the following is output:

```
Builder <colour1> steals <resource> from builder <colour2>.
```

If the tile has no builders who can be stolen from, the following is output:

```
Builder <colour1> has no builders to steal from.
```

The display is updated with the word GEESE being added below the value of the tile that the geese were placed on. If the geese were previously on a tile, the word GEESE is removed from that tile.

After handling the geese, the active player completes their turn as normal.

For example, consider Figure 2. Say builder Blue has 5 HEAT, Red has no resources, Orange has 11 BRICK, and Yellow has 3 GLASS. Now Blue rolls a 7. The program outputs:

```
Builder Orange loses 5 resources to the geese. They lose:
5 BRICK
Choose where to place the GEESE.
```

Blue inputs 9 to place the GEESE on tile 9. The program outputs:

```
Builder Blue can choose to steal from Yellow.
Choose a builder to steal from.
```

Blue inputs Yellow. The program outputs:

```
Builder Blue steals GLASS from builder Yellow.
```

Now Blue has 1 GLASS, 5 HEAT, Red has no resources, Orange has 6 BRICK, and Yellow has 2 GLASS.

3.7 Saving a Game

When a game is saved, the following information is printed to a file in the following order:

```
<curTurn>
<builder0Data>
<builder1Data>
<builder2Data>
<builder3Data>
<board>
<geese>
```

A description of each line follows.

`<curTurn>` is the identity of the builder whose turn it will be when the game starts.

A builder's data is printed out as follows:

```
<numBricks> <numEnergies> <numGlass> <numHeat> <numWifi> r <roads> h <housing>
```

where `<roads>` is the number of each road, separated by one space, and `<housing>` is the list of residences that have been built, each represented as a pair. Each pair in the list represents the vertex of the residence followed by the letter representing the residence on it.

For example, the row

```
1 2 1 2 3 r 16 36 19 h 10 B 15 T 27 H
```

would be the output for a player with 1 BRICK, 2 ENERGY, 1 GLASS, 2 HEAT, 3 WIFI, the roads 16, 36 and 19 and residences at vertices 10 (a basement), 15 (a tower) and 27 (a house).

`<board>` is described in section 3.8.

`<geese>` is a number between 0 and 18 representing the tile that contains the geese. Traditionally, geese would start on the Park tile since that tile produces no resources. It is up to you whether they are initially displayed or not before a 7 is rolled.

3.8 Board Layout

For loading and saving a board layout, the following format will be used.

The resources will be printed in the order of the resources on the board with 0 representing BRICK, 1 representing ENERGY, 2 representing GLASS, 3 representing HEAT, 4 representing WIFI, and 5 representing PARK with each resource being followed by its value

For example, the board in Figure 3 is

```
0 3 1 10 3 5 1 4 5 7 3 10 2 11 0 3 3 8 0 2 0 6 1 8 4 12 1 5 4 11 2 4 4 6 2 9 2 9
```

Note that in the provided files, a tile for PARK will always be followed by the number 7 even though it doesn't have a value associated.

While the board is symmetric, it is not symmetric enough to allow an easy formula for the relationship between tiles, edges and vertices. You will therefore need to come up with some reasonable approach to handle this.

4 Gameplay

There are four distinct stages in the game, during which input should be handled differently.

4.1 Beginning of Game

At the beginning of the game, each builder chooses two initial basements to build. The basements will be chosen by builders in the order Blue, Red, Orange, Yellow, Yellow, Orange, Red, Blue.

Each builder is prompted with the statement:

```
Builder <colour>, where do you want to build a basement?
```

Each builder responds with a number that represents a valid vertex.

Once a valid vertex has been entered, prompt the next builder. Once all builders have placed their two basements, print the updated board and begin the game.

4.2 Beginning of Turn

At the beginning of a turn, display the board and print the following:

```
Builder <colour>'s turn.
```

followed by the status of the builder. A builder can then enter any of the following three commands:

Command	Description
`load`	sets the dice of the current builder to be loaded dice
`fair`	sets the dice of the current builder to be fair dice
`roll`	rolls the current builder's dice. This ends the "Beginning of the turn" phase and moves the builder to "During the turn".

4.3 During the Turn

During the turn, a builder can input any of the following commands:

Command	Description
`board`	prints the current board
`status`	prints the current status of all builders in order from builder 0 to 3
`residences`	prints the residences the current builder has currently completed
`build-road <road#>`	attempts to builds the road at `<road#>`
`build-res <housing#>`	attempts to builds a basement at `<housing#>`
`improve <housing#>`	attempts to improve the residence at `<housing#>`
`trade <colour> <give> <take>`	attempts to trade with builder `<colour>` giving one resource of type `<give>` and receiving one resource of type `<take>`
`next`	passes control onto the next builder in the game. This ends the "During the Turn" phase.
`save <file>`	saves the current game state to `<file>`
`help`	prints out the list of commands

The `help` command prints the following

```
Valid commands:
board
status
residences
build-road <edge#>
build-res <housing#>
```

```
improve <housing#>
trade <colour> <give> <take>
next
save <file>
help
```

4.4 End of Game

The game ends when a builder has a total of at least 10 building points. The first builder to have 10 building points is the winner. At this point, the builders are prompted:

```
Would you like to play again?
```

If the response is "yes", the game starts over from the beginning. If the response is "no", the game exits.

4.5 Handling Input

When waiting for a response from standard input, the program prints '> '.

If a builder attempts to improve or build a residence, or build a road, and the space is invalid (for any reason), output:

```
You cannot build here.
```

If a builder attempts to build anything and they do not have the required resources, output:

```
You do not have enough resources.
```

If an unrecognized command is entered, output:

```
Invalid command.
```

If the game receives an end-of-file signal at any point, the game ends with no winner, and saves the current game state to `backup.sv`.

5 Command-line Interface

Note: Command-line options are important to implement as they aid in easy testing of the correctness of your program. Programs which do not implement all command line options will not be fully tested and will receive low marks.

Your program must support the following options on the command line:

Command	Description
`-seed xxx`	sets the random number generator's seed to `xxx`. If you don't set the seed, you always get the same random sequence every time you run the program. It is good for testing, but not much fun.
`-load xxx`	loads the game saved in file `xxx`. You may assume that the file being loaded was saved as a valid game state as described in the Saving section. **Note: loading from a saved file will be heavily used during the testing of the project and should be considered a priority while implementing the project.**
`-board xxx`	loads the game with the board specified in the file `xxx` instead of using random generation. This file will contain the order of the tiles for the game as described in the saving the game section. The number of each tile present does not have to match the random generation. e.g. it is valid to have a board which is entirely WIFI resources with the value 12^2.
`-random-board`	starts a game with a randomly generated board.

The command-line options can be given in any order. Note that `-board` and `-load` are contradictory commands to each other. Similarly, if a game is being loaded or a custom layout is being used, `-random-board` is ignored and the layout specified in the saved game or custom layout is used instead.

6 Questions

These questions are to be answered as described in the **project_guidelines.pdf**. We recommend reading and thinking about these questions while you are designing you project.

You must answer questions 1 and 2, AND any 3 of questions 3-7.

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?

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?

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 numbers of players). What design pattern would you consider using for all of these ideas?

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 alway followed a legal sequence of actions during their turn, what design pattern would you use? Explain your choice.

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?

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?

7. Did you use any exceptions in your project? If so, where did you use them and why? If not, give an example of a place that it would make sense to use exceptions in your project and explain why you didn't use them.

Grading

Your project will be graded as described in the project guidelines document.

Even if your program doesn't work at all, you can still earn a lot of marks through good documentation and design, (in the latter case, there needs to be enough code present to make a reasonable assessment).

Above all, make sure your submitted program runs, and does something! You don't get correctness marks for something you can't show, but if your program at least does something that looks like the beginnings of the game, there may be some marks available for that.

7 If Things Go Wrong

If you run into trouble and find yourself unable to complete the entire project, please do your best to submit something that works, even if it doesn't solve the entire project. For example:

- only one housing type exists
- does not recognize the full command syntax
- building points are not calculated correctly

You will get a higher mark for fully implementing some of the requirements than for a program that attempts all of the requirements, but doesn't run.

A well-documented, well-designed program that has all of the deficiencies listed above, but still runs, can still potentially earn a passing grade.

8 Plan for the Worst

Even the best programmers have bad days, and the simplest pointer error can take hours to track down. So be sure to have a plan of attack in place that maximizes your chances of always at least having a working program to submit. Prioritize your goals to maximize what you can demonstrate at any given time. One of the first things you should probably do is write a routine to draw the game board (probably a `Board` class with an overloaded friend `operator<<`). It can start out blank, and become more sophisticated as you add features. You should also do the command interpreters early, so that you can interact with your program. You can then add commands one-by-one, and separately work on supporting the full command syntax. Take the time to work on a test suite at the same time as you are writing your project. Although we are not asking you to submit a test suite, having one on hand will speed up the process of verifying your implementation.

Note that for testing purposes, it is more important to have the non-randomized features, e.g. loaded dice, reading a saved game from a file, working than the randomized features, e.g. fair dice, random generation.

You will be asked to submit a plan, with projected completion dates and divided responsibilities, as part of your documentation for Due Date 1.

9 If Things Go Well

If you complete the entire project, you can earn up to 10% extra credit for implementing extra features. These must be outlined in your design document, and markers will judge the value of your extra features.

10 A Note on Random Generation

To complete this project, you will require the random generation (or rather, pseudo-random) of numbers. See the provided `shuffle.cc` for an example of declaring a random number generator that is used to randomly shuffle a vector of integers. The random number generator is seeded *once* with the (possibly global variable) `seed`, and then used in `std::shuffle`. Games that are started with the same seed value have the same sequences of random numbers generated.

Due Dates and Deliverables

See the project guidelines for due dates and submission requirements.