Azul

COSC1076 Assignment 2 (v1.0)
Advanced Programming Techniques
GitHub Repository: https://github.com/usefulmana/apt-azul-game/releases/tag/v1.0

By R.A.M
(Anh Nguyen, Ruby Rio, Mitchell Gust)

Introduction

Azul is the second assignment of COSC1076 *Advanced Programming Techniques: Second Semester,* with a *group* assessment type, before forking off *individually* for the last two of the four Milestones associated with this project. This report is written by *R.A.M:* a group of 3 students namely:

- Anh Nguyen *s3616128*
- Ruby Rio *s3786695*
- Mitchell Gust *s3782095*

The following report aims to analyse the choices behind the style and structure of our implementation; an aim to develop and write a C++ 11/14 solution that allows a version of the board game *Azul* to be played via the IDE console. Following the group discussion, there is an individual report to analyse decisions made when the group work disjoins for the completion of *Milestone Three*. [1]

Discussion of Linked List Implementation

Our linked list has been implemented in a different way to traditional, incorporating templates. Templates act as a blueprint for creating a *generic* class, function or method: an implementation where there is no specific data type assigned to the code in question, and rather, it is determined by the classes which call on it. The team implemented the Linked List in this way as it provided a greater flexibility when using the list and would allow us to reuse the same architecture and implemented structure, in more scenarios or areas in the code than once. For example, if the Linked List implemented for that of an *integer* type; however, could later have a new instance created for that of type *String*, the use of generics would allow for that. [2]

We have applied the Linked List as the structure for our Tile Bag. The Tile Bag acts as a type of queue, with items being added (or *pushed)* to the end, and removed (or *popped)* from the front, which fits into the architecture of a linked list. The processing time of a linked list is also faster than that of a 1D or 2D array, which fits in to the need for the Tile Bag to be frequently manipulated changed and updated throughout the duration of the game.

Discussion of C++ STL Vector

The use of the vector was determined to be most applicable for the Tile Centre; however, was also used to hold *Players*. To brief the use case of the Tile Centre, it is a container that holds one tile on game initialisation (1st player marker). As the game progresses the container is given or subtracted tiles depending on user actions. The game requires Tiles to be added to the end, and Tiles to be removed at any position.

The number of pieces held in the Tile Centre varies for each round, meaning the size of the container will not always be the same, and is therefore not appropriate to be made a fixed size. As tiles also need to be removed and added to the Tile Centre; it was clear that the container for Centre Tiles needs to be dynamically sized [3]

By choosing a vector, it allows for the use of the member function call *push_back()*. The Push Back method allows the insertion of a given element to the end of the container. It does this by copying the old vector values along with the new given element into a vector of appropriate size. This is an appropriate option for the Tile Centre, as this operation is done in amortized time and insertion will only ever occur at the back of the container [4]

As for removing the chosen coloured tiles from the Tile Centre. Vectors provide iterators. In conjunction with the remove () method, we can remove all elements which fit our colour criteria by using the

---

[1] Wiley, T. (2020). Assignment 2 {v 1.0}. [online] rmit.instructure.com. Available at: https://rmit.instructure.com/courses/67111/assignments/480190 [Accessed 20 Sep. 2020].

[2] Cplusplus.com. 2020. *Templates - C++ Tutorials*. [online] Available at: <http://www.cplusplus.com/doc/oldtutorial/templates/> [Accessed 30 September 2020].

[3] Wiley, T. (2020). Lecture 6. [online] rmit.instructure.com. Available at: https://rmit.instructure.com/courses/67111/pages/week-6-learning-materials-slash-activities?module_item_id=2520396 [Accessed 27 Sep. 2020].

[4] Bjarne Stroustrup (2014). Programming : principles and practice using C++. 2nd ed. Upper Saddle River, Nj: Addison-Wesley.

returned iterator (to the new end of range)[5]. Then erase () to resize the vector to the correct number of elements after removal [6].

Discussion of 1D and 2D Array

We utilised both the 1 and 2 Dimensional arrays within our implementation of Azul.

The team used 1D arrays frequently, for many different objects that did not require a large amount of storage. A prime example of this would be our choice for the *Broken Row*. In the game, the broken row sits under our Mosaic, holding a maximum of 7 places at any time. In retrospect, 7 is not a large object, and therefore, can be easily manipulated through using a more simpler data structure, such as a one-dimensional array. Moreover, memory in 1D arrays is contiguous, whereas they are not in 2D arrays, and therefore the primary option will run faster; an important fact when being associated with one of the most constantly manipulated features in the game.[7]

The team incorporated a 2D array for the set-up of the Factories in the *Game* and organising the *Player* Mosaic. We found the 2D array to be the most suitable choice here, as visually when looking at the board, it was easy to set up as a set of x and y co-ordinates; translated into [x][y]. Moreover, when searching through an individual factory, it was efficient to store *all* in the 2D array, and then maintain the [x] point, whilst rotating the [y] point, over keeping them all as individual 1D arrays. Not only did this make a faster search, it also contributed to effective memory management, as it causes less objects to be instantiated. Despite the memory in a 1D array being contiguous and running faster, instantiating it in this manner would duplicate this speed by 5, ultimately making it slower in the long run.

Discussion and Efficiency Evaluation of Abstract Data Types and Structures

Below is a discussion of R.A.M's *Abstract Data Types and Structures* that were utilised. Each class is discussed below:

Game

The Game class was designed to store data and operate methods that are used to run the logic and structures involved when playing *Azul*. The class handles the main operations through two methods: *void play ()* and *void execute (const std::string & command, Player * player),* with the latter being called upon inside the first.

The ***play*** method sets up all physical aspects of the game; firstly the Tile Bag: automatically with *void setTileBagAutomatically(),* which has a pre-defined list of entries, or with *void setTileBagFromString(const std::string & line)*, with a saved game. This is followed by the player names entered by the user, added to the *vector of* Players (discussed above). The function then fills the Factories appropriately, before starting the method of play by printing both them and the mosaic to the terminal. The ***execute*** method performs the required actions to play the games on the tiles. The command reads in the turn input from the user, matching it to the relevant player, before removing the tile from the factory and placing it on the correct row. We have chosen to use 2 separate methods to manage the setup of the game, as we unanimously agreed it was too large of a functionality for one.,

The Game class has also been populated with numerous ***validation*** methods to ensure the game is running as expect, and to support the process of the game. We have chosen to implement a vector that maintains a list of errors of the player's input. This method, namely ***checkInput***, contains multiple if-else statements that checks the input passed into the function to see if it meets any of the conditions- which all signify an incorrect entry. This way, not only can we keep track of all errors that have been entered, but we can also adequately respond to each individual error with a personalised message, in order to reduce confusion with the player.

Broken Row: Although predominately being handled in the *Player* class (discussed below), the team made some executive decisions on how moves would be executed in the Broken Row that were implemented in *Game*. The Broken Row has been given the mosaic coordinate **0**. Whenever a player wants to discard a tile (i.e. it can go nowhere but the broken row), they should use the following command: *turn <factory> <colour> 0*. We have done it this way to guide the user on what to do in this instance, as it is one of the most

[5] en.cppreference.com. (2018). std::list<T,Allocator>::remove - cppreference.com. [online] Available at: https://en.cppreference.com/w/cpp/container/list/remove [Accessed 27 Sep. 2020].

[6] en.cppreference.com. (2018). std::list<T,Allocator>::remove - cppreference.com. [online] Available at: https://en.cppreference.com/w/cpp/container/list/remove [Accessed 27 Sep. 2020].

[7] D or 2D array, w. and Rudolph, K., 2020. *1D Or 2D Array, What's Faster?*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/17259877/1d-or-2d-array-whats-faster> [Accessed 29 September 2020].

confusing parts of the game. It was also a more efficient way to achieve the solution, as programming this to happen automatically proved to be too time consuming for the constraints we were provided.

Scoring: Although having its own class (discussed below), one method of calculating the score, namely *deducting* points, is managed in the Game class. The team decided to incorporate this method here, as the deduction of points happens in relation to the *Broken Row*; an element which is managed in the Game Class, making it a more efficient implementation as it can easily communicate with the broken row structure.

Player

The Player class was designed and implemented with structures and methods to work with information on the users playing *Azul*. As mentioned above, the main data structure used to was a vector.

Our Player class holds numerous methods: some of which were inferred necessary to have in this class due to the outline's specification, and others which we chose to implement within this class for efficiency or effectiveness purposes. Some of the more obvious methods of the Player class were variables to hold both the *name* of the player, and the individual's *score*. These were placed within the Player class, as the contents of these variables are unique to the individual and will be stored and present against its relevant user.

Other choices we made within our design of the Player class was the handling of the Mosaic and Broken Row within this class. We have set up our implementation so that the Mosaic/Game board that is manipulated during the game is stored against the user itself, rather than being saved in the game separate to the individual. We have done it in this way, so that we are able to easily call the player's mosaic board when rotating turns, rather than managing the contents and updates to the board as a separate entity to the player. This way, we will be able to take some logic out of the Game class, which manages most methods and functions that conduct the flow of the game and spread it evenly around other classes. This also allows for incorporation of *Scoring:* direct communication between the Mosaic and Score Class is present and allows for constant updates on the matter. As the main mosaic is managed in the Player class, we have also implemented the printing and adding to the *Broken Row*, which resides under the Mosaic, to be fitted in this class, for the same reasons as above. By keeping all similar logic in the same class, the team felt this made for a easily understandable and effective program.

Tile

An integral part of the gameplay, the Tile class is used to store the properties of the tokens that are used to play the game- Tiles are the main component (aside from the game board) that are manipulated and moved around in order to score in and win *Azul*.

Our tile class contains minimal methods, as the Tile in itself is never directly changed. The Tile class only contains both a getter and setter, aside from the constructor/deconstructor. The manipulation of the tiles, as well as the instantiation of the TileBag, all occur in our *Game* class (discussed above). The reasoning behind this decision, was because we felt our LinkedList implementation (which has its own .h classes) contained and provided the full scope and functionality needed for the bag, and therefore, by creating an instance of this we have access to methods that will cover the scope needed within our implementation.

Utils (utilities)

The Utils class was designed to incorporate structures and functions that work with *outside* tasks needed to run the game. These tasks are not ones that are relevant directly to the game *Azul* in itself, however, are needed in order to allow the IDE/OS to perform out-of-game functions.

Within this class, we have included functions such as: quitGame, renameAFile, writeMultipleStrToFile, writeToFile, getDateTime, deleteAFile, and checkIfFileExists- all of which relate to both the loading, saving and running of the game. We have decided to keep these in their own class in order to limit the amount of code in either Game or Main- the two other reasonable options of where these methods would otherwise be stored. Moreover, we decided for efficiency in regard to understanding the code to an outside developer, to keep these methods in a separate utilities class would make them both easy to find and easy to understand why they exist.

One vital method that hasn't been listed above is splitString, which reads in both a string and a char, and is used to help include the buoyancy and resilience of the game. This method is called within the Game class when a user enters a turn and validate the entrance to see whether it matches the specified format. If it doesn't, it offers helpful feedback to the user in where they have gone wrong. We have included this method here as, although it is used as a part of the game, it does not directly relate to the game method, and rather, as a check for user-entered errors. Thus, for easier understanding, we decided it was more of a utility over anything else.

Score

The *Score* class was designed to perform the horizontal and vertical checks needed to correctly add up the player's scores. We decided to create this in its own class in order to alleviate the number of methods, and the weight of both the Game and Player class. Moreover, it allowed for greater flexibility within the program, and helped with the organisation of the methods.

The class does not contain any data structures, and rather maintains an *int* (namely *roundScore*) that is continuously updated and contains the player's current score. We determined that the incorporation of a structure such as an array was not necessary or a good fit for this functionality. The deduction of points, however, is completed in the *Game* class (discussed above).

Explanation of Tests

As a team, we decided to split our tests up in to 4 categories: a complete and correctly entered test, a half complete test, tests with illegal formatting and tests with illegal movements.

- *Complete & Correct:* These tests aimed to show that the software was working correctly when the user enters all turns and user prompts correctly and successfully. These tests, although showing that the game is working as intended, is not a real reflection of real life, and therefore can never be the only types of tests used.
- *Half Complete*: These represent a game where all turns and prompts have been followed correctly, however, there are less than 5 rounds. This test is used predominately to test the *load game* and *test mode* aspects are working as intended.
- *Illegal Move*: These tests represent a game where the player has entered a turn in a manner that does not match the expected output of the system. For example, if a player tried to grab tiles from a factory greater than 5. With these tests, we expected the system respond resiliently, and help the user correct their mistakes
- *Illegal Format*: These represent a move where the user selects an option or tries to alter the game in a state that falls outside the designed structure. For example, if a user tries to create 3 players for the game. With these tests, we aim that the system deals with them in the same way as the *Illegal Move* above, in that it helps the user identify the error they have made, and how to solve it.

These tests were used alongside the *test mode* we were required to complete as a team. This test mode aimed to automatically make and print the executed Azul game depending on the turn's written in the save file, and then quit without crashing. Along with these tests, we also used a tool called Valgrind [8] to help hunt for and solve memory leaks within the program. Valgrind allowed the team to ensure that the game ran efficiently and didn't have any malicious back end memory bugs that weren't presenting when typically running the game.

Discussion of Group Coordination and Management

The R.A.M team worked efficiently and effectively together, delegating tasks and meeting regularly. The team took the time at the beginning of the assignment on getting to know each other, understanding each other's strengths with C++ and programming in general. As a team, we also took time to learn the intrinsic properties of the board game *Azul*. It was made sure that all team members were emersed in every aspect of the project, however, were able to play to their strengths. Each team member was involved in the programming tasks, however, we delegated tasks on levels of confidence- having the sure programmers take on the more difficult aspects of the project, whereas others displayed their weekly contributions in report writing, team organisation, and code refinement- all of which are equally important. Team R.A.M had a mix of international and national students on different time zones. Although sometimes running into difficulty finding a time that suits all, this issue was combatted by having regular meetings as a team that relayed information mentioned in those meetings, and discussing concepts and tasks that needed to be re-distributed, as well as constant check ins and meetings in the Team's General chat- we all found the idea that no one was left out or confused extremely important and ensured to uphold respectful and honest discussions for the duration of the project.

[8] Valgrind.org. 2020. *Valgrind Home*. [online] Available at: <https://valgrind.org/> [Accessed 2 October 2020].

Individual Report *Ruby Rio, s3786695*

GitHub Repository: https://github.com/rubyrio12/apt-azul-game

For this Milestone, I chose to undertake the task of implementing the following enhancements: *Box Lid & Randomness (minor)* and *Advanced Azul (major)*

Box Lid & Randomness

This enhancement saw the addition of the Box Lid, an area in which to store the tiles that have been dusted off the board from the Broken Row or Unlaid Row of the player at the end of the round. When the tile bag is emptied, it is re-filled using this lid space. For the Box Lid, I have chosen to use a *linked list* structure.

There were three reasons for this choice: Firstly, the box lid feeds directly from the Tile Bag, of which is a Linked List; storing the same number of tiles and using them in a similar way. Secondly, when stripping the Box Lid down to its simplest form, it mimics the structure and process of a Linked List. The Box Lid acts as a type of queue, with items being added (*pushed)* to the end, and removed (*popped)* from the front, which slides nicely into the architecture of a linked list. Similar to what was mentioned above, the processing time of a linked list is also faster than that of an array, which fits in to the need for the for rapid calling of the structure, and the process of the tiles. Accompanying the Box Lid is the ability for the game to randomised. Following the outline of the specification, to **turn off randomness,** the player must enter a *seed*. The seed will provide a fixed variable by which the game is to be shuffled. This is denoted by entering *./azul -s <seed>* when launching the game. Otherwise, if only *./azu*l is entered, the game will be shuffled on a variable picked by the computer based on *time*, and will swap items in the Tile Bag based on my *swap* method. No new data structures were needed to complete this method; however, there were an addition of methods. In order to shuffle the Tile Bag, I have used the method of *srand() and rand()*. From here, I have created a method called fillTillBagSeeded(int seed), which takes in a randomly generated number decided in *main.cpp* and mixes up the Tile Bag depending on this number. This is overwritten if a player enters *./azul -s <seed>,* as the same method is called; however, the seed the player has entered is passed through, allowing for a consistent method of shuffling

In order to show off this change, I have altered the amount of Round to 6. In this way, we see the 100 Tile Bag emptied, and needed to be filled from the Box Lid at the end of the 5th Round. In order to make the game simpler, when the Tile Bag is filled and reset, it is shuffled by a *reshuffle* value of 5. This still provides randomness to the player when playing the game; however, removes the need for the original tile bag and seed to be documented in the save file, rather just needing the post-shuffled tile bag

Advanced Azul

This enhancement saw the addition of both 6 Tile and Grey-Board Azul into the game. To play this version of the game, the user must enter *./azul –adv*. This enhancement also includes the *Randomness & Box Lid* enhancement detailed above. 6 Tile Azul saw the physicality of the game settings altered. The Game Board was increased to 6x6, while an extra broken row space, and an extra tile name Orange (or O). For this, a new set of #defines were created in the Types.h class – where all our fixed variables lie. These added definitions increased the board parameters and Broken Row size to 6. At this point a set of *second* classes were created: **AdvancedGame.h/cpp and AdvancedPlayer.h/cpp**. In these classes, the parameters that were originally in place were switched to the new extensive parameters, and immediately changed the dimension of the board. Main.cpp saw a new "advanced" menu come into play, which called on these two new classes. Moreover, in order to extend the denotion of orange, I have highlighted the parts of the board which have been extended in *orange,* to add extra look and feel to the game. Grey Board Azul built within the two new classes noted above. Again, no new Data Structures were needed in order to create this function. Multiple methods were introduced, most important checkPlacement and placeTiles. placeTiles() saw the change from the automatic placement of Tile to the manual placement of Tiles at the end of the round. This functionality utilises checkPlacement in order to check the moves and entrances entered by the player.

Unfinished Components:

Although striving to fully complete both enhancements, time acted as a limitation which ultimately left items unfinished.

- The **Load Game** and **Test Mode** method have not been updated to suit **Advanced Azul**. The game runs in complete form if a new game is started using *./azul –-adv* after compiling.
    - Because of this, there are no Test or Pre-Load files available that work properly when used

Milestone 4 - Individual Report - Anh Nguyen (s3616128)

Github Repository: https://github.com/usefulmana/apt-azul-game-m4

Minor Enhancement - Randomness / Box Lid

Data Structures

An additional data structure that I implemented for this enhancement was the box lid - a linked list inside the *Game* class. The reason the linked list was chosen because structurally speaking the box lid functions the same way as the tile bag. The time complexity of inserting a tile at the end of the box lid and deleting a tile in the front are both *O(1).* Other than that, I did not make any changes to the existing data structures since the logics and variables of the 'random/box lid' mode were the same as 'default' mode's.

UI Changes

Several menu functions in *main* were repurposed and modularized to accommodate different play modes. For instance, the *showMenu()* function which was responsible for getting the user's input and redirecting the users to the appropriate menu is now modularized. It is now only returning the user's input. This function will be called by other functions such as *engageAdvancedMode()* to get the user's input and redirect the user to the appropriate functionality.

Mechanics Changes

An additional property named *seed* was added to the *Game* class. It is used to generate a randomly ordered tile bag. Since the seed's value is important to the game, its value will be saved in a save file at line 3. Moreover, a '#random' flag is added at the beginning of a save file file to help the program IDs which game mode to engate. Finally, the 'default' mode will now always have the seed's value of 1 to ensure compatibility between the default and the new game mode. Methods such as *setTileBagAutomically()* have been rewritten to work with both game modes.

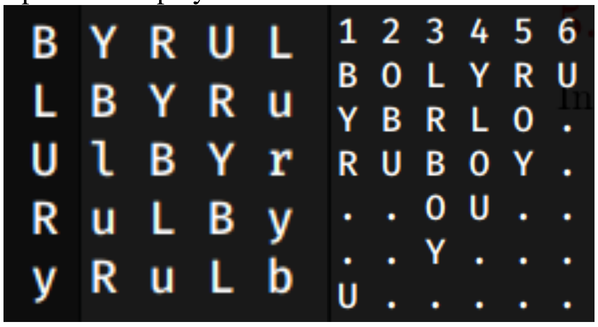
The tile bag generation algorithm involved swapping values of an initial predetermined tiles bag over 100 times. The indexes of the swapped values are determined by the random numbers generated by the *default_random_engine* with a given seed within a given range(0 to 99).

One significant change I made to the 'random/box lid' mode was that I lifted the number of rounds limit. The reason was because after 5 rounds, the players would only play with the initial 100 tiles in the tile bag. The tiles from the box lid would have never been touched. To prevent the game from going for too long, a different end game condition is established- when a player has the first horizontal line on his/her mosaic the game will end.

Major Enhancement - Advanced Azul

There were no changes to the UI or data structures for this enhancement. Several properties have been added to *Types.h* to describe the advanced game mode. This enhancement is an evolution of the previous enhancement. The save file is identified by the '#adv' flag in the beginning of the file. An additional command has been added - place <row> <color> <column> to place the tile to the mosaic. Players can also specify a seed to shuffle the tile bag. At the end of the round, a tile placing phase will take place.

The most significant change was the creation of the *AdvGame* and the *AdvPlayer* classes. The reason I decided to create different classes for the advanced game mode was that many of the logics of the *Game* class is tied in into how my group chose to present the players' mosaics.



```
B Y R U L
L B Y R u
U l B Y r
R u L B y
y R u L b
```

```
1 2 3 4 5 6
B O L Y R U
Y B R L O .
R U B O Y .
. . O U . .
. . Y . . .
U . . . . .
```

Default *Advanced*

Many important aspects of the *Game* class such as the scoring and the placing of tiles are based on the capitalization of letters on the mosaics. To accommodate the changes required for the advanced game mode, a partial or even a complete rewrite will be required. The backward compatibility requirement exacerbates the problem. Additionally, considering the time constraints, it is inadvisable to modify the *Game* class. To maintain the current data structures setup, the *AdvPlayer* class was created. Compared to the *Player* class, this class has an expanded 6x6 mosaics and several other small differences. This approach inevitably resulted in a lot of duplicated code. Nevertheless, the program is working as intended.

Individual Report *Mitchell Gust, s3782095*

Github Repository: https://github.com/s3782095/apt-azul-game
Completed: Minor: Randomness & Box Lid, Major: Advanced Azul (**6-Tile Mode Only**)

Shuffling Algorithm Design

The shuffling algorithm constructed takes reference from the modern version of the *Fisher-Yates shuffle,* introduced by *Richard Durstenfeld* in *1964*. Designed for shuffling a sequence, the original algorithm by *Ronald Fisher and Frank Yates continuously* generates a random index within the range of the data set. The corresponding element to each generated index is copied to a separate result list and is removed from the original data set. This is done until all elements in the original data set are removed.
In *Durstenfeld's version*, we continue to make use of a randomly generated index but use it instead to swap the value that exists in the generated index and the value of the last element not yet swapped in the data set.[9] Providing me with a perfect base plate for my implementation of a *Tile Shuffle*.

To adapt this brilliant algorithm to the world of *C++*, I have developed several elements:
I have made use of the discussed random number generator *std::random_device*. This generator produces uniformly-distributed non-deterministic numbers, and is used in combination with the *std::uniform_int_distribution<int>* to calculate a random index (*swapIndex)* appropriately.[10]
In order to shuffle the full contents *of the Tile Bag*, I have initialised the minimum *(0)* and maximum *(99)* to reflect the number of tiles in the *Tile Bag* without the *First Tile*. Shuffling had to occur prior to adding the First Tile as the placement of 'F' is always set in stone and will always need to appear at the front. These values are important as they classify the range at which the *std::uniform_int_distribution<int>* can pick a random number from.
With the development of my *for loop* I am able to retrieve the index of the last element in the data set, create a new *std::uniform_int_distribution* appropriate to the current *min* and *max* values, and perform the swap. To exclude previous selected elements from being chosen as the next random integer, the max is decremented.

6-Tile Implementation

The 6-Tile implementation was built and set to the default gameplay standard after great success. Increasing the board to 6x6, adding an extra broken tile slot and the addition of an Orange tile provided a more suitable gameplay experience since the implementation of extra rounds *(needed in order to showcase the Shuffling Algorithm)*. The 6-Tile addition was brought about by redefining the production of the *Game* class and ensuring all other elements of the game were well suited to the transition.

Discussion of Box Lid Implementation

The Box Lid is a collection of a varying number of Tiles. It holds a great number of elements, whereby items need to be either added to the end or removed from the start of the data set. 1D and 2D arrays are no match to the Linked List's and Vector's ability to frequently and quickly manipulate the data. A Vector may seem like a good option due to its constant time of performing these operations, but it is throttled in performance by its need to be copied when dealing with scalability.

As our group implementation included the use of a *templated* Linked List, I made use of the same blueprint (*header file -> LinkedList.h)* utilised by the Tile Bag. This decision had a major reduction in code duplication, and further reflects why it is the smart data type choice.

Consequences

The consequences of implementing the Tile Shuffle is the need to adjust the *Main* and *Game* class to account for the seed when saving, loading and testing the game. This required extensive modification that resulted in a trade-off for larger testing files and therefore longer load times. However, these differences are indistinguishable during gameplay. The User Interface is also a trade off in result of this implementation as the only way to input and access the capabilities of the seed is through a console command rather than from the main menu.

[9] Wikipedia.org. 2020. *Fisher-Yates*. [online] Available at: <https://en.wikipedia.org/wiki/Fisher–Yates_shuffle#Fisher_and_Yates> [Accessed 17 October 2020].

[10] Cppreference.com. 2020. *Std::random_device*. [online] Available at: <https://en.cppreference.com/w/cpp/numeric/random/random_device> [Accessed 15 October 2020].