Tabassum Fabiha -- tf2478

1.1 - Access modifiers

- The JFrame class has the public static int field EXIT_ON_CLOSE which is also declared as final to protect the field from being changed by the user.
- The System class also contains three public static fields, InputStream in, PrintStream out, and PrintStream err that are all also declared as final to prevent them from being changed by the user after being declared in the class.

1.2 - Primitives

- 5 bytes can store a range of -2^{39} to $2^{39} - 1$ integers
- Base 10 range of exponents $\approx 10^{-76}$ to 10^{76}
- Number of significant decimal digits = 9

1.3 - Total order

- Reflexive: Yes, this relationship is reflexive, because for every NET A all the points in A will fit on or inside all the points in A.
- Antisymmetric: Yes, this relationship is antisymmetric, because for NET A and NET B if all the points of A are on or inside the points of B and all the points of B are on or inside the points of B, then all the points of A are on the points of B, and so A is equal to B.
- Transitive: Yes, this relationship is transitive, because for NET A, NET B, and NET C, if all the points of A are on or inside the points of B and all the points of B are on or inside the points of C, then all the points of A are on or inside the points of C.
- Total: No, this relationship is not total, because for NET A and NET B if all the points in A are not on or inside the points of B then that does not mean that all the points of B must be inside all the points of A.

1.4 - Critiquing a class

- Cohesion: The JFrame class does only have what you would expect it to have as a container and contains no additional methods that have nothing to do with the purpose of the class.
- Completeness: The JFrame class is complete, and you don’t need to write extra methods to do tasks that occur often given the purpose of this class.
- Convenience: JFrame is convenient in that the class allows a programmer to worry less about the micro scale in creating a GUI and focus on the macro. Programmers can create GUI components put them in a container like an

instance of the JFrame class and know that the class will handle maintenance such as displaying all the GUI components in their appropriate locations.
- Clarity: The interface is clear to programmers. The methods have consistent names, are easy to understand and do exactly what they say they will.
- Consistency: The JFrame class has consistency in that the fields and methods are appropriately named in regards to their purpose and that the parameters and returns on methods make sense and are what you would expect to see.

1.5 - Test case documentation
- Test case ID: rsplk_5
- Author: Tabassum Fabiha tf2478
- Summary: Testing the advanced AI in step 5.
- Precondition: Creation of working RSPLK game system with a Sim player which simulates the human by alternating between playing the last throw made by the AI or the next throw in the sequence “rsplk”. The AI should also be able to receive information about the last moves played in the game.
- Test Data: N/A
- Expected Results: The game should run through a full 10000 rounds of RSPLK. We should see in the result summary at the end of the game that the AI substantially won more times than the Sim by recognizing the Sim’s strategy and then counteracting it.
- Postcondition: N/A

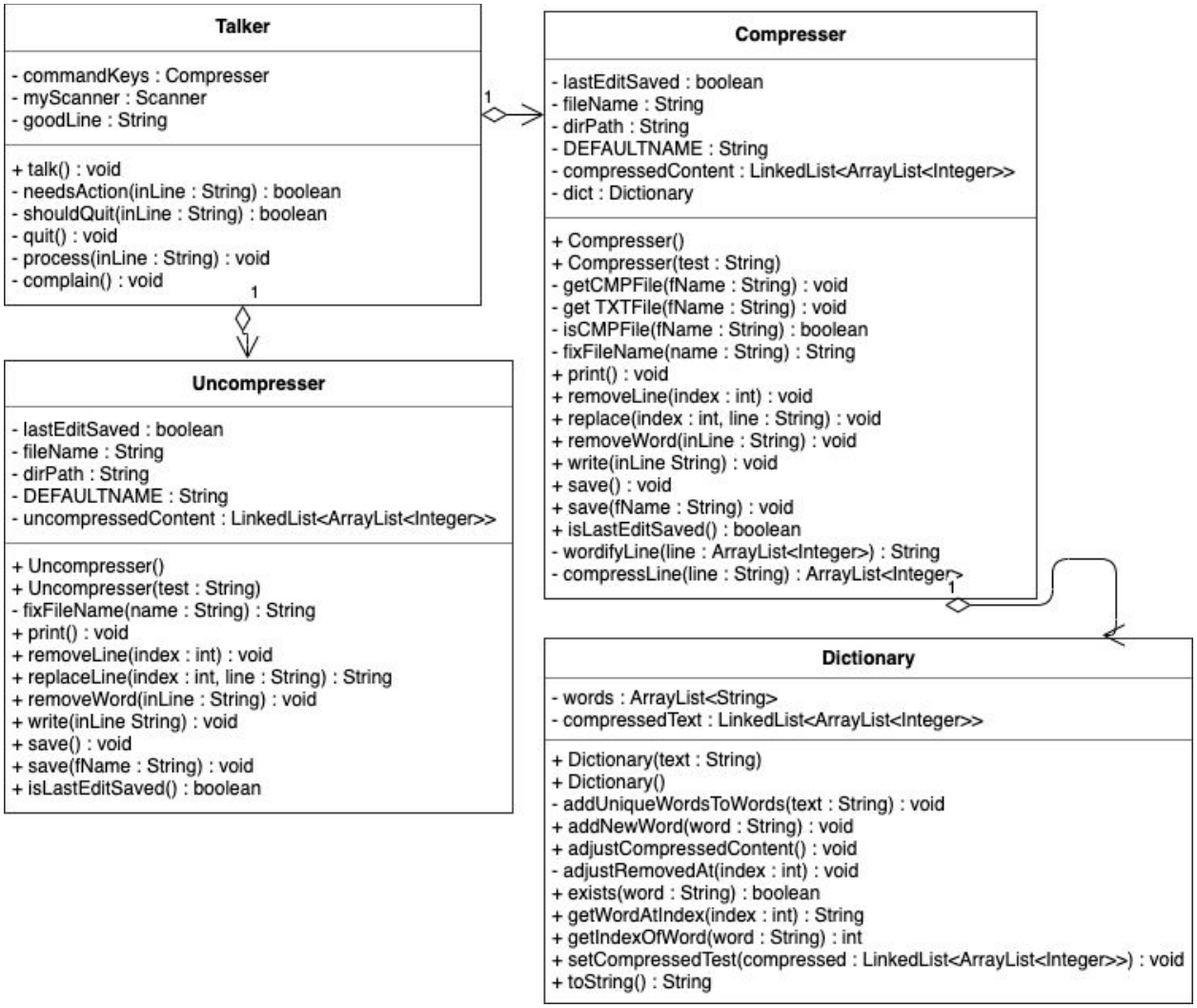
1.6 - Class categories
- Information Holders
    - This type of class should know information and so I expect to see a lot of fields.
- Service Providers
    - This type of class should be able to do things when asked. I expect to see a lot of methods in this class.
- Controllers (Sequencers)
    - This type of class makes choices and decides what to do when. I expect to see a lot of if statements and for, and while loops.
- Structures
    - This type of class knows how things relate to each other. I expect to see references to files and data structures in this class.
- User Interfaces

    - This type of class talks to the user and handles the input and output. I expect to see a lot of try and catch statements because of the lack of reliability when it comes to interacting with an external source.
- Coordinators
    - This class coordinates/assigns tasks appropriately, but without making decisions. I expect it to have a lot of import statements and method calls.

1.7 - Factory method for the Programming Block

```
public class Compresser {
        private Compresser() {
                …
        }
        private Compresser(String fName) {
                …
        }
        public static Compresser newFile() {
                return Compresser();
        }
        public static Compresser getFile(String fName) {
                return Compresser(fName);
        }
        ...
}
```

1.8 - UML for the Programming Part



- Complexity: 2 * (lines/boxes) = 2 * (60/8) = 15
- All the classes Uncompresser, Compresser, and Dictionary have more than seven methods, but that is because I wanted to make the code more readable by splitting a method up into more function calls. For say Compresser I could have placed getCMPFile and getTXTFile all in the Compresser constructor method however I felt at that point the method would become too cluttered. At the same time I did not split these up into separate classes to reduce the number of methods in each class because I felt that all the methods needed to be in the classes they are in now for each class to be self sufficient and complete.
- Note:
    - I know it's Compressor not Compresser, but it's too late to change it now.

Talker

- get a command from the user
- can ask compressers and uncompressers to print, get a file, replace a line, save a file, and replace a word
- can quit
- can know if we recently saved

Compresser
Uncompresser

Compresser

- can print, get a file, create a new file, replace a line, save a file, replace a word
- can compress txt files
- can open txt files and cmp files
- can save txt files and cmp files
- has the fileName
- has the path directory
- has the default name
- has the compressed content

Dictionary

Uncompresser

- can print, get a file, create a new file, replace a line, save a file, replace a word
- has the fileName
- has the path directory
- has the default name
- has the compressed content

Dictionary

- can give the index of a word in the dictionary and give the word at an index in the dictionary
- add new words to the dictionary
- fix its dictionary to rid of unused words
- has unique words used in the compressed content
- has the compressed content

Compresser

1.9 - Clean code for the Programming Part

- Best Name: Class Dictionary → exists(String word)
    - This name explicitly describes what the method will do and is short and to the point. It is pronounceable, searchable and contain no encodings, and tells the user at first glance that the return statement will be a boolean.
- Second Best Name: Class Compresser → removeLine(int index)
    - Much like the previous this name also explicitly describes will do. It is short, pronounceable and searchable, just as Chapter 2 suggests method names should be.
- Best Method: Class Compresser → print()
    - This method does only a single thing and each line in the method is completely readable and understandable. It calls the method wordifyLine to uncompress a line instead of doing it itself because the print method has no responsibility to know how to uncompress a line. These mentioned techniques were all suggested in Chapter 3.
- Worst Method: Class Compresser → replace(int index, String line)
    - I think this is the worst method because it does something in secret that the programmer will not know it does explicitly from looking at the name of the method. It updates a field to say that there has been an edit made. Even though the method name does not explicitly say it will do it i still thinks updating the field belongs here because of how hand in hand this goes with editing the text. Also I feel that leaving that for the programmer to update on their own later could have lead to possible bugs if they became careless.
- Best Comment: Class Compresser → LinkedList<ArrayList<Integer>> compressedContent

  The LinkedList is used to store the compressed content because we have the need to constantly add or remove lines from the content and since we will need to traverse the entirety of it for many of the times that we access the content the flaw with LinkedLists having to traverse through from the very beginning is not an issue.

  The LinkedList is of type ArrayList<Integer> mostly in part to create an easily recognizable difference between what is the content as a whole and what is a single line. I used an ArrayList instead of an array because I still needed each line in the content to be mutable to change if the user were to want to change a specific word in the line.

- This comment was the nonnegotiable kind mentioned in Chapter 4 to explain the aggregations I used to hold content.