BCPR301 Advanced Programming – Assignment 1

List of Components for marks

# Online Repository Link:

<https://github.com/Matt-13/AraPythonProject>

# Textual Modelling Language used in my solution:

PlantUML

Textual diagram based off our current code:

@startuml

package GraphInterpreter {

class FileController {

command : String

data : String

file\_location : String

display()

handle\_command()

read\_file()

print\_file()

save\_file()

quit()

view\_help()

output\_error()

}

class FileConverter {

classes : ArrayObject

converted\_classes : ArrayObject

codeToText : String

convert\_file()

add\_class()

print\_program()

return\_program()

read\_file()

}

class FileReader {

allMyClasses : ArrayObject

code : String

check\_if\_plantuml()

count\_occurences()

find\_classes()

}

class ClassBuilder {

name : Object

attributes : Object

methods : Object

all\_my\_attributes : ArrayObject

all\_my\_methods : ArrayObject

add\_class\_attributes()

add\_class\_methods()

print\_class()

return\_class()

}

class Attribute {

name = Object

\_return = Object

\_\_str\_\_()

}

class Method {

name = Object

\_return = Object

\_\_str\_\_()

}

class FileView {

error\_message : String

fc\_defaults()

fc\_file\_found()

fc\_file\_not\_found()

fc\_syntax\_error()

fc\_permission\_error()

fc\_load\_file\_error()

fc\_plantuml\_converting()

fc\_plantuml\_classes\_not\_found()

fr\_file\_accepted()

fr\_plantuml\_error()

file\_written()

fe\_defaults()

fe\_too\_many\_args()

fe\_command\_syntax()

fe\_abs\_syntax()

fe\_abs\_path\_error()

output()

general\_error()

print\_help()

display\_graph\_code()

file\_error()

}

class FileWriter {

write\_file()

}

class FileExecuter {

command : String

main()

print\_to\_screen()

}

FileController \*-- "Reads command and Sends file to" FileConverter

FileController \*-- "Updates user with" FileView

FileConverter \*-- "Loads file with" FileReader

FileReader o- "Updates user with" FileView

FileController o- "Writes file with" FileWriter

FileExecuter --> "Starts the" FileController

FileConverter \*-- "Builds classes with" ClassBuilder

ClassBuilder \*-- "Creates an" Attribute

ClassBuilder \*-- "Creates a" Method

}

@enduml

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # | Feature | Used by my peers (2 Marks) | Comments on Robustness  (2 Marks) | Complete and well implemented (2 Marks) | Expected Marks  (out of 6) |
| 1 | Support Command-line arguments | Liam Brydon  Clement Yeo (Liam has also created commands of his own which I am using) | The command line argument method checks and washes user input data twice. Once in the FileExecuter, where the Sys.Argv argments are input, and again in the FileController where the command is sent to the Model. The FileExecuter sys.argv checks the string, and lowers it to lowercase. It also checks if there are more than two arguments entered. If nothing is entered, the program continues with defaults. If the load, save, or absload commands are entered with no parameters, the program stops and sends an error message with the syntax of the command. In the FileController, the second part of the commands where you choose filenames are checked if they exist, or end with .txt. If they don’t end with .txt an error message is displayed. | This feature is well implemented because it cleanes user input so the program doesn’t raise exceptions. I have tested this feature with multiple test cases, such as integers, wildcards(symbols). And are all correctly handled by the program. | 6 |
| 2 | Has a line based interpreter | Liam Brydon  Clement Yeo  (Liam has also created his own commands I worked on the load, help, and absload) | As said above, I am using sys.argv to check for arguments passed with the program through the cmd. Each command and option is passed into the program with str(sys.argv[i]) The program checks the arguments to a maximum of 2. For example FileExecuter [command] [option] is allowed. FileExecuter [command] is allowed. FileExecuter [command] [option] [option] is not allowed. | This feature is well implemented because it expects so many arguments on the line. And verifies user input based on a select number of commands and options. | 6 |
| 3 | Display commandline help | Liam Brydon  Clement Yeo | The commandline help method shows a help page based off the MS-DOS Command prompt style help. It shows all commands in the program. FileExecuter help – shows the help page. | This feature is well implemented because it is easy to use, if you are confused you can just type in help and it will show you all the commands in the program. | 6 |
| 4 | Change commands and options | Liam Brydon  Clement Yeo | As said above, I am using sys.argv to check for arguments passed with the program through the cmd. Each command and option is passed into the program with str(sys.argv[i]) The program checks the arguments to a maximum of 2. For example FileExecuter [command] [option] is allowed. FileExecuter [command] is allowed. FileExecuter [command] [option] [option] is not allowed (program only takes one command and one option). | This feature is well implemented because it expects so many arguments on the line. And verifies user input based on a select number of commands and options. | 6 |
| 5 | Extract Data | Liam Brydon | Program loads the file from a default or chosen directory, it then sends it off to the ClassBuilder which turns it into Object Attributes and Methods. The ClassBuilder checks what a class is by the word “class”, it checks this by checking if the line before has a “}” symbol on it. The Attribute class checks an attribute by checking if a line has a “:” symbol on it. The Methods class checks a word on a line if it has the “()” symbols next to it. | This feature is well implemented because it checks the data while extracting it. | 6 |
| 6 | Validate Data | Liam Brydon | Program validates the graph being input by checking if the first line contains the word @startuml. And the final line containing the word @enduml, otherwise the code stops and sends an error message. | This feature is well implemented because it checks the graph file for known plant UML syntax before running the program. | 6 |
| 7 | Can load data from a file | Liam Brydon | Program is able to load data from a default Graph.txt, or a filename.txt of your choosing. (must be a .txt format) | This feature is well implemented because the user has choice over what file they wish to load. | 6 |
| 8 | Can provide exception handling | Liam Brydon | Program is able to handle exceptions with the use of try and if statements. If an exception is raised, the program will try to operate around it. | This feature is well implemented because the program hasn’t raised an exception to date. | 6 |
| 9 | Amount of error handling/trapping | Liam Brydon | Inside the FileView, there are over 150 lines of code containing error message details. | This feature is well implemented because there is an error message for every type of possible error. | 6 |
| 10 | Can deal with directories and subdirectories | Liam Brydon  Clement Yeo | The Load command and the default command deals with the current working directory and the directory above. The Absload (absolute load) deals with a path of the user’s choosing. | This feature is well implemented because the user can choose where they want to load the file from. | 6 |
| 11 | Can save and read data from a database | Liam Brydon | The program is able to save data to a database. | This feature is well implemented because the program is able to save the graph result to a database for further use with the loadcode and savecode commands. | 4 |