Read Me

## How to run:

Assure the “Startup object” in the “Application” tab of the properties of project is set as “SimpleSpreadsheet.Program”. After compile and generate the program, double click the “SimpleSpreadsheet.exe” in the \bin\Debug or the \bin\Release folder. Enter “help” in the console to get help information.

## How to test:

In the “Application” tab of the properties of project, switch the “Startup object” from “SimpleSpreadsheet.Program” to “SimpleSpreadsheet.TestProgram”, then uncomment the lines in Main() of TestProgram to run each test case accordingly.

## How to view the UML manufactures:

Unfortunately, the UML diagrams feature in menu “Architecture” in Visual Studio 2015 is discarded by Visual Studio 2017, so I have to return to the manual drawing with a tool “Diagram Designer” developed by MeeSoft. You can download the same version as follows to open my diagrams:



## Design considerations:

The whole development process is an iteration process.

At the beginning, I tried to go fast after drew a use case diagram and a flow diagram. I passed a lot of UML designs and made the detailed class design by Visual Studio 2017 directly. I hadn’t realized a data layer (a spreadsheet data model) until I implemented the class design and the requirements. So I decided to go back to perfect the UML manufactures.

I complied with the RUP development. Use case diagrams and simple activity diagrams(or flow diagrams) are used to disclose the main features, system boundary and operational process of the target system. But they aren’t involved with the details and cannot be verified independently. Along with the former manufactures, I drew the communication diagram(collaboration diagram) to discover logic controllers and data entities, and had verified the system boundary by the way. Therefore the three tier concept(view layer, logic layer, data layer) got clear.

Then I drew a sequence diagram to verify the communication diagram and to excavate more information. I named functions to make the design get close to codes. In the same time, I deduced the analysis class diagram from the communication diagram to make the object associations clear. After finished the two, the detailed program logic appeared. And a round of refinement began.

Afterwards, I drew the swimlane activity diagram to verify the program flow, and use it to refine the diagrams before. At last, I drew a status machine for the spreadsheet model to ensure the data flow is not out of my expectation. Then everything got clear: the logic flow is major. So it is necessary to extract a class hierarchy to generalize the common process and specialize the specific handling procedure.

Taking the user input validation into consideration, I chose the regular expression to minify the complexities of uncountable kinds of dirty inputs. However, the regular expression was not almighty, each sub class needed to validate the real data filtered by regular expression respectively. So the parent class took responsibility of the validation logic skeleton while the sub classes implemented key parts.

After implemented the main program features and had a smoke test, I drilled into the logic of the CommandDispatcher class. I had found some if clauses, and discovered that the dispatcher had to take responsibility of the building process of illustrations. Then if a new command is added, at least two places of the CommandDispatcher need to modify. Furthermore, the concrete command classes had knowledge about which command they should handle, but could not actually control which command they handle. At first, my intuition led me to the chain of responsibility design pattern. But I did not accept because there wasn’t a sequence of responsibilities among the command classes until I found the building process of illustrations could be refined by the decorator design pattern and by the way, the chain of responsibility design pattern could utilize the former design pattern to make CommandDispatcher simpler. So according to the single responsibility principle and the open closed principle, I used the chain of responsibility design pattern and the decorator design pattern. After that, the CommandDispatcher just took responsibility of the sequence of concrete command classes, and merely one place needed to modify when a new command class was added. Furthermore, the concrete command classes could control which command they handle really.

Afterwards, I tried to extract the HelpCommand into the CommandBase hierarchy but had found I was wrong. Because it would break the Liskov substitution principle for the HelpCommand didn’t handle the spreadsheet model in fact which should be handled by the CommandBase. And it also broke the interface segregation principle because the Execute() method in CommandBase would either write the spreadsheet model or return an error message while the same method in HelpCommand would just return a string of illustrations. The erroneous design just mingled two irrelative implementations into one general purpose function signature.

At last, I extracted all the command tags to make them easy to change. Eventually, this design along with the extraction of magic numbers gave me a favor in testing. I didn’t need to hard code the command and the boundary values when wrote test cases.

For the naming convention, I synthesized the C++ style and the C# style which were both in Pascal case and Camel case on the same places. In the former, temporary variables start with “t”, function parameters start with “v”, and member variables start with “m”. In the latter, member variables start with “\_” and all are in Camel case. Why didn’t I use full upper case with underscore for the constant variables? Because in C++, it is for macros, it will make macros and constant variables hard to differentiate in the managed C++ which is the bridge linking .NET codes and unmanaged codes. Moreover, isn’t Pascal case with underscore easy to read than the full upper case with underscore?

For the test, I mainly focused on two sides – static white-box testing and dynamic white-box testing.

Firstly, I reviewed all the design diagrams, made sure they were verified each other. For instance, the use cases were described accordingly in the flow diagram and the sequence diagram. The sequence diagram could be transformed into the communication diagram equivalently in the major part, while some trivial details could be ignored. The flow diagram was totally a subset of the swimlane activity diagram, and all the entities could be found in the communication diagram and the sequence diagram. Then the entities would be extracted as classes in the analysis class diagram which revealed the associations among them for the design class diagram. And the classes reflected the abstraction relationship in the use case diagram, contained functions in the sequence diagram. I had refactored the codes with two design pattern, however I did not reflect them in all design diagrams except the class diagrams. Because it was just a refinement, it would make other diagrams not display their business key points clearly. After that, I reviewed the codes to assure they were coincident with the design diagrams.

Secondly, I used back the console program because the display of output of unit test project was incorrect. I’ve covered almost all paths, all branches and all conditions. I used the equivalence partitioning to reduce test cases. For example, “commandDispatcher.CheckInputToDispatch("UNRECOGNIZED", out message);” and “commandDispatcher.CheckInputToDispatch("C", out message);”, “commandDispatcher.CheckInputToDispatch("N", out message);” were all the same because they would enter the same logic path; however, “commandDispatcher.CheckInputToDispatch("S", out message);” and “commandDispatcher.CheckInputToDispatch("S ", out message);” were different because they would separate at the branch “if (!command.StartsWith(CommandTag))”. I ensured to cover all data boundaries, both valid and invalid. What’s more, I had written four stress tests to make sure the program was robust. Certainly, because the program did not target on running in a multi-thread environment, so it was unnecessary to write multi-thread stress tests. I hadn’t written any unit tests for the regular expressions because they were included in the test cases for the concrete commands.

That’s all. I hope the statements above have elaborated my considerations clearly.