UG-17: Code review

YOUTUBE LINK: https://youtu.be/eGxtdlISnjo

Code structure

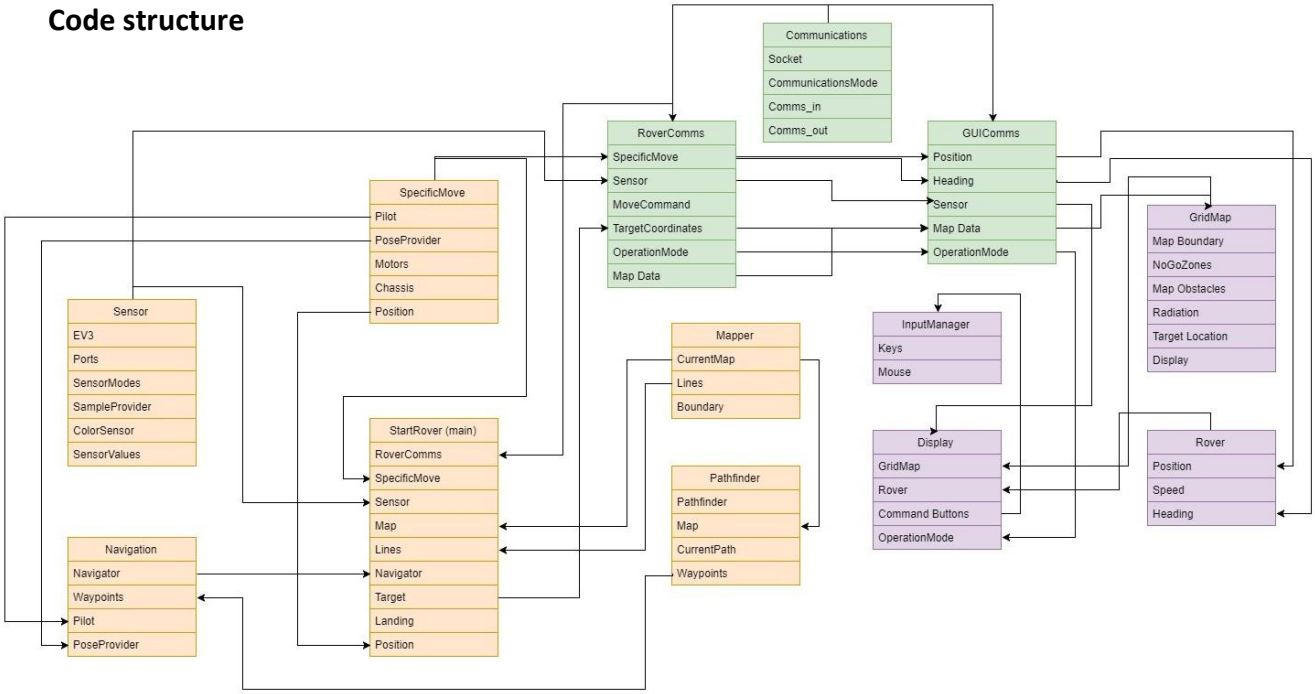

Figure 1: Data structure diagram

Before fully implementing our code, we've drafted out a data structure diagram to decide on how we want our classes to be structured, implemented and communicate amongst each other. Figure 1 presents how the classes shall be structured, separated as the GUI (Graphical User Interface) side, Rover side, and communication side.

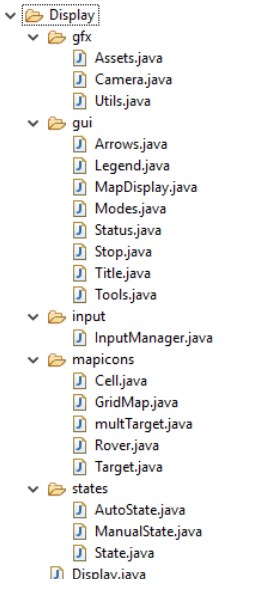

Figure 2: GUI structure

For each 'side', we separate our sub classes into folders. Taking the example of the GUI side. The subclass of the GUI side is separated by:

- gfx: Everything related to loading images, graphics
- gui: Everything related to initialising GUI such as buttons
- Input: Everything related to user input
- mapicons: Everything related to the rover map
- state: Everything related to the state of rover

The folder structure of the GUI side is grouped by relations in their functionalities. This structure also applies for the Rover and communication side.

Coding convention and standards

```
/**
 * The GridMap class contains function relating to map operations. It draws
 * map as a grid with each cell representing a small area. Functions relating
 * to the modifications of these cells such as the lining function are contained
 * within this class.
 * @author Hoang Long Ngoc Pham, Yong Yang (Maximilian), Katon Akhmad Zaky
 * @version 0.2.0
 * @since 20-09-17
 */
```

Figure 3: Class comments

Figure 3 presents the commenting standard that we use for each class. Our class commenting standard informs the user of: the class functionality, the contributors of the class, the version, and the last time it was updated. We think that these information are beneficial as our works are shared amongst each other. Queries regarding the class can be easily directed to its author. This commenting is also applied for code documentation.

```
/**
 * This method is used to change the id's of an area to the input id
 * @author Hoang Long
 * @param shape (Vector<Cell>) First parameter is a vector used to create shape
 * @param id (int) Second parameter is the id to change to
 * @return Nothing
 */
```

Figure 4: Method comments

Figure 4 shows the commenting standard for each methods. Our method commenting standard includes: the functionality of the method, the contributors, the parameters, and the return. This commenting is also applied for code documentation. For very complex methods, we've applied coder comments within the methods to improve readability for future contributors.

```
public void newLines(Vector<Cell> dots, int id, boolean shape)
{
    for ( int i=0; i<dots.size()-1; i++ )
    {
        lining(id, dots.get(i).getX(), dots.get(i).getY(), dots.get(i+1).getX(), dots.get(i+1).getY()) ;
    }
    // It will connect the first dot with the last when drawing a shape
    if ( shape == true ) lining(id, dots.get(dots.size()-1).getX(), dots.get(dots.size()-1).getY(), dots.get(0).getX(), dots.get(0).getY()) ;
}
```

Figure 5: Coding style

We've decided on following a specific coding style so that our codes will look neat and consistent when we start merging our codes together. We decided on to use the tab to indent, and have curly open brackets on a new line. The coding style was decided shortly after we all started our part of the code. We were able to simply configure the general style of the coding by using the Eclipse code style preference function.

Coding efficiency

For the purpose of code efficiency, we created a method for the functionality that will be used more than once, so that method can be called multiple times with only creating a block of code.

Figure 6: Eclipse's code coverage tool

With the functionality of Eclipse's code coverage tool, we were able to identify and delete every block of code and variables that we're not used in the full. We made sure that this is correct by performing every possible functionalities that could be performed with the program.

Code Testability

As the code was implemented, we performed unit testing for each possible methods that we finished working on. This made things more efficient as we were able to identify bugs and acknowledge its source.

On the rover side, the codes were testable by having a driver function to perform movements on the rover. As for the GUI side, our testing was mostly by visualisation and printing values to ensure that we are on the right state of operation. We tested for every different possibilities that can be performed and how it can move on to the next state.

Variables

For readability purposes, we ensured that all of our variables are defined with meaningful names, except for variables that are used for looping. The variables are consistently defined based on the locality of where the variable is required to be initialised and used.

Data and memory

Multiple type of data structures were used, such as vectors, arrays and lists. Most of our code revolved around vectors, so we had to make sure that our data usage was as efficient as possible. Our rover map is define by 2D array, filled with cell objects. We ensured that the maximum size of the cells are fixed to prevent from possible overflow. Rather than making new object cells, map implementation is done more efficiently by changing the id variable of

the defined cell object to determine which type of map element it is (NGZ, unexplored, and etc.).

In the rover side, we utilised the buffered functions such as reader and writers. After the buffers are finished from being used and before the program closes, we've ensured to close them.

Error handling

Our code does not implement any error return values, instead, we throw exceptions if any error occurs. These thrown errors are acknowledged, although they are caught so that the process can be continued rather than crashing the program. Most exceptions are created for system IO such as connecting rover Bluetooth and reading XML file. For every error handlers in our code, we've added a piece of information for it so that we know where the errors have occurred.

Many errors and bugs were identified during the implementation process. We've managed to cover errors that have occurred, such as: null pointers, language exceptions, and index exceptions.

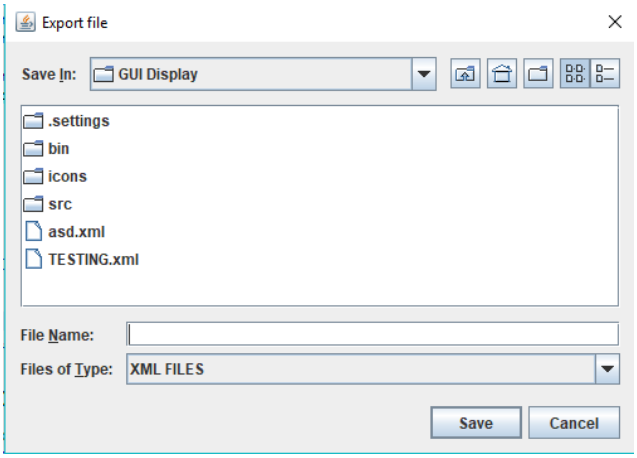

Figure 7: File explorer

To minimise I/O errors with importing and exporting map file, we've used the java-created file explorer, so that users have control to which files they want to save and load.

We decided not to implement any format error handling for reading XML files, as the XML files will be created by our own code which follows the DTD format. The XML file should be readable and shouldn't cause any error as long as it follows the DTD structure.

Things that could be improved in the code:

- Cleaning the commented test codes
- Possibly making our own error exceptions for XML file format
- A few visual bugs on the GUI
- Create advanced recovery for hardware issues or rover Bluetooth errors.
- Possibly make errors 'more user friendly' by having it as a popup window rather than a string output in the IDE.

- Improve O complexity on methods

Katon Akhmad Zaky – a1680219

Reflections report

Positive parts

Presentation flow
I've written a script for this presentation and practiced it for more than 6 times. I can feel that my presentation was much smoother than my first try, as I have felt less stuttering. More practice definitely do better things for me.

Presentation structure
I am satisfied with the way I've structured my contents. Structuring my presentation helps improves the flow of the presentation and watching through my video, it does feel like start to end.

Checklist covered
The contents I've presented are based on the checklist that was given by the course to act as a guideline. I made sure that the contents of my presentation covers the checklist.

Parts to improve on

Enthusiasm
I tried my best to sound as enthusiastic as possible when doing my presentation. Despite recording multiple shots, I am still left unsatisfied with the result as I sound monotone for all the recordings. This is really something I really have to practice on to be able to catch other people's interests.

Presentation introduction and ending
I really feel like my presentation had no proper introduction and ending to capture the audience's attention. I should have started my presentation with a brief description of the project and the contents that I will be covering. My presentation should have ended with a summary of the contents I've presented.

Less boring contents
I had a hard time choosing which images and sentences I should be using for my presentation slides. Some of my slides had just images of my code and no images at all. To make my presentation more entertaining, I could be using more web images and use analogies to describe some things. I need to improve on how to balance out a technical presentation to be more entertaining.

Code Review Checklist

1. **Q:** Does the code completely and correctly implement the design?
   A: Yes, the code is performing correctly as expected and as we designed.

2. **Q:** Is the code well-structured, consistent in style, and consistently formatted? Is the structure clean and indentations correct?
   A: Yes, we made sure to utilise the Eclipse code style preference function so that our codes looks neat and consistent.

3. **Q:** Does the code adhere to a documented standard or convention?
   A: We did not follow any specific official standard or conventions, but we agreed on the commenting format, opening bracket format, indentation style and more.

4. **Q:** Is the code clearly and adequately documented with an easy-to-maintain commenting style?
   A: Yes, because our works are shared with each other and needs to be readable by others. We've applied our commenting format for each method and additional comments are added for complex parts within the method.

5. **Q:** Are there any uncalled or unneeded procedures or any unreachable code?
   A: No, we ensured that this does not happen by running a code coverage tool.

6. **Q:** Are there any blocks of repeated code that could be condensed into a single procedure?
   A: No, we made sure that repeated codes should be in a method so that it can be called multiple times.

7. **Q:** Is the code testable?
   A: Yes

8. **Q:** Are any modules excessively complex and should be restructured or split into multiple routines?
   A: Yes, the import and export is complex, although we only split into multiple routines when a code needs to be called more than once.

Variables

1. **Q:** Are all variables properly defined with meaningful, consistent, and clear names?
   A: Yes, except for variables that is used for looping.

2. **Q:** Do all assigned variables have proper type consistency or casting? and correct scope?

A: Yes

3. **Q:** Are there any redundant or unused variables?
   A: No, we ran a code coverage tool to prevent this.

4. **Q:** Data and Memory
   A: Yes, data and memory are used.

5. **Q:** Are the correct data operated on in each statement?
   A: Yes

6. **Q:** Is every memory allocation deallocated?
   A: Yes we made sure every buffer and connections are closed when finished.

7. **Q:** Is storage use efficient?
   A: As efficient as we can think of

8. **Q:** Are imported data and input arguments tested for validity and completeness?
   A: No, no validity test for XML file format.

Error Handling

1. **Q:** Are return values (in particular error returns) not ignored?
   A: Yes, we made sure they are printed out and caught.

2. **Q:** Are timeouts or error traps used for external device accesses?
   A: Yes

3. **Q:** Are files checked for existence before attempting to access them?
   A: File explorer is used for import and export, so things are selected manually.

4. **Q:** Are all files and devices are left in the correct state upon program termination?
   A: Yes

5. **Q:** Are System I/O mechanisms are consistently used?
   A: Yes, buffered reader, write, IO exception and etc.

6. **Q:** Are input values (or other data used) are checked for reasonableness before use?
   A: No, feel like not needed for XML file format.

7. **Q:** Are errors are detected and handled, and processing continued?
   A: If error occurs, they are caught and process shall be continued.

8. **Q:** Are error handling conventions are followed (standard use of error handling task, etc.)?
   A: Yes, in the code if there are any method that has potential to create an error, we

ensure that it also output where the error occurred so it can be debugged.

9. **Q:** Does code pay attention to recovery from potential hardware faults (e.g. arithmetic faults, power failure, and clock), also pays attention to recovery from device errors?
   A: No recovery methods.

10. **Q:** Is there any redundant code?
    A: No, the redundant codes have been covered by the code coverage tool.

11. **Q:** Are there any leftover stubs or test routines in the code?
    A: No, but there are some that are commented out, although not executed.

12. **Q:** Are constants and literals are not hardcoded?
    A: No

13. **Q:** Are indexes, pointers, and subscripts tested against array, record, or file bounds?
    A: Did not use any pointers, we made sure that indexes in data structures are within boundary.

14. **Q:** Are all loops, branches, and logic constructs complete, correct, and properly nested?
    A: Yes