Jackie Qiu: jqi26 – 65394578 Hamesh Ravji: hra62 - 43832772

The application has been structured into eight major classes: GameEnvironment, Crew, CrewMember, FoodItem, MedicalItem, Planet, Ship and RandomEventGenerator. An important design choice the team had to make was at the end of the creation of the command line application. The team realised that the GameEnvironment class had become a 'God Class', nearing a thousand lines of code. This made it very complex and difficult to find bugs. The decision was made to move several methods to other classes where it made more sense, such as moving the showInventory method from GameEnvironment to Crew. The method for generating random events was made into its own class: RandomEventGenerator, as it was particularly large. Refactoring also made it easier to test the program. Another decision was made to break up larger methods such as setup in GameEnvironment into several methods, to reduce the complexity of these methods and promote maintainability.

The only class that our GUI classes directly communicate with is GameEnvironment, which is done by passing the GameEnvironment as a parameter in the constructor. Therefore, each GUI class could make changes to the GameEnvironment whenever necessary so that any choices the player makes affect the GameEnvironment. If a GUI class needed to use a method from one of the other classes, it would use GameEnvironment and various getters to achieve this. Our program also uses the Observer design pattern. The RandomEventGenerator is the observed class while Crew is the observer. Whenever a random event occurs, it is set to changed and notifies Crew, passing an argument such as the food item that was stolen. The update method in Crew determines what needs to done using the argument passed to it, and updates the crew's inventory, crew members or the ship appropriately.

FoodItem, MedicalItem, and CrewMember are all superclasses that have children who inherit the attributes and methods. The children call super to use the superclass's constructor, passing relevant values depending on the parent. FoodItem, MedicalItem, and CrewMember are all general concepts whereas their children are more specialised or do something unique. All of the children share similar methods, thus by creating parents, repetition of large chunks of code is avoided. While none of the children of these three superclasses have attributes or methods exclusive to them, they pass unique values in their construction to differentiate them. This was also useful for collections, such as enabling the creation of an ArrayList which can contain all of the Crew's FoodItems, even if the food items are different from each other.

Collections are used throughout the application, especially ArrayLists. In the Crew class, ArrayLists have been used to collect objects of the other classes. An ArrayList of CrewMembers is used to contain all of the CrewMember objects that are still alive. The crew's inventory is stored in ArrayLists of both MedicalItems and FoodItems. ArrayLists are also used for various purposes in the GameEnvironment class. An ArrayList of Planets is used to store the planets that the crew can visit when travelling to a new planet. The setup method takes two ArrayLists as parameters. One contains the crew member types as integers the player has chosen while the other contains the names of the crew members as strings. A TreeSet is also used in the Crew class for the showInventory method. There it is used to display the crew's medical and food supplies with their counts in alphabetical order.

Unit tests are written for every class excluding GameEnvironment, RandomEventGenerator, and the GUI classes. GameEnvironment was excluded as per the specification, whereas RandomEventGenerator relies heavily on the Random class, which unit tests are not appropriate for. The GUI classes were excluded as the vast majority of the code cannot be tested with unit tests. When including these, the test coverage is 32.3%. The total instructions when excluding these is 1468, while 1228 instructions are covered, which gives a percentage of 83.65% coverage. This percentage was maximised by using 'Coverage as' in the Eclipse IDE, which displayed which methods had been tested or not. If a section of code was highlighted in red or yellow, new unit tests were written to cover it. The coverage is lowered due to several methods in Planet for implementing searching a planet. These methods use Random which cannot be properly tested with unit tests.

While no aspect of the project was especially challenging in terms of programming, there was still quite a lot of work. An important thing the team learned while creating the application is how to manage a large project and break it up to smaller achievable tasks and working up to creating larger components. The team also learned how to program simultaneously with another person while working on the same project. This also highlighted the importance of documentation and JavaDoc which allows other people to more easily understand what a piece of code does. The project also allowed us to gain a much better understanding of how to properly test code.

To summarise, the project involved a lot of work but a much better understanding was obtained of how real software projects are conducted in the industry. The team was also surprised at how much work goes into a game as small and simple as this. Feedback wise, we believe the difficulty of the project and the workload is quite reasonable, however, we believe there should be more coding for GUIs taught in lectures and labs, especially the more advanced techniques. The project is a great way to showcase the importance of things like modularity, documentation, and testing which may be underestimated otherwise.

One part in particular that went quite well was the GUI section of the application. Despite the specification stating that the command line application would take around 20% of the total effort while the GUI would take around 50%, the team had managed to create the GUI in less time than creating the command line application even when excluding tasks such as creating the UML class diagram or writing JavaDoc. This was achieved by keeping our code modular during the creation of the command line application, which meant that most code could be reused for the GUI with only a few modifications. WindowBuilder also allowed the very quick creation of interfaces.

Another aspect that went well was communication. Throughout the project, communication was maintained through texting, voice communication and screen sharing using the internet. We would let each other know what we were working on, how we were going to do it and asked for advice whenever any of us were unsure about something. This was important so that we did not accidentally work on thing same thing and duplicate work or make components that were incompatible with each other. Effective communication allowed us to collaborate on the project without even needing to meet in person. In general, the project went very well, with only a few parts that could be improved.

What did not go as well as it could have was during the planning/design phase(s). The team failed to foresee many aspects of how the application would work in the end meaning that the final application is quite different from the initial UML class diagram. Many attributes and methods which are in the final application were not in the initial draft of the UML class diagram as the team was unsure how it would work at the time. This required us to constantly update the diagrams whenever a new method or attribute was required. The team even missed an entire class: Planet, during this initial stage. The lack of vision before programming started meant that sometimes whole sections of code was scrapped when it was realised that something would not work. While this was not a huge issue for our small program, it can be seen how it would be a major blow to a real-world large scale software project. As such, an improvement the team could make on the next project is to spend more time during the planning/design phase(s). We should improve on picturing how a program should work before any programming occurs. This would lead to a smoother process with fewer setbacks during the development stage.

Name	Effort spent(hours)	Agreed contribution (%)
Jackie	~100	55
Hamesh	~85	45