2AA4 Assignment 1 Report

Q1.1) Is this MVP really minimal?

I would say not, specifically if we examine the second feature in the backlog, “rolling eight dice.” This feature can be further simplified into simply rolling two dice. This would still show functionality for rolling and keeping track of multiple dice, and still allows the random choosing method to keep certain dice rolls. Immediately aiming for 8 dice is quite unnecessary for this minimal product, and thus can be simplified.

This of course would also change the feature where three skulls (cranes for this product) need to be rolled to end the turn, however, now including a feature where the game ends after two skulls would represent the same functionality and viability (while being arguably simpler as well). Similarly, the 3-of-a-kind scoring system can also be changed to a 2-of-a-king feature to match the 2 dice system.

Q1.2) Is this MVP really viable?

Almost (so technically no.), as it well represents the Piraten Kapern final game, but there are a few incorrect features.

Showing that there are two players in the game easily represents functionality for more than one player with different scores to be involved. The ability to roll die of course is quite a prominent feature in the game that needs to be shown in all products of the game. The ability for the game to recognize when enough cranes have been rolled and calculate scoring based on certain symbols also recognizes room for different scoring systems for the future. The three cranes in combination with the 3-of-a-kind feature show functionality for counting points through x-of-a-kind methods for different symbols, where the gold and diamonds can each have their own scoring amounts. The ability for the user to decide how many games to play is also very representative of how the players in real life would decide beforehand how many rounds they want to play.

Having said all this, the feature that ends the game when the player has rolled three cranes is quite incorrect according to the rules. Ideally this feature would have to change into three *skulls* which is a value that could actually be rolled on the defined die and also matches the original game rules.

Q1.3) Can some features be simplified?

Yes, as mentioned before, rolling eight dice at once can easily be simplified to rolling two dice as it would still showcase multiple dice tracking functionality. Consequently, the three cranes and the 3-of-a-kind features can be simplified into two cranes and 2-of-a-kind scoring.

Q2.1) Which part of your code do you consider to be technical debt?

Currently, I would say the calculate score function is a technical debt since there is no real way to implement future scores for when the player achieves several dice of the same kind (x-of-a-kind). The scoring system as of right now is just a for-loop that goes through all the dice and adds 100 points for any diamonds or gold faces that are seen. This makes it difficult for the game to score different number of points based on different faces and cannot track any x-of-a-kind scores at all.

The bigger technical debt, however, that I see in the program is the two-player system tracking that is currently implemented. As of right now, there is a fixed sized array that represents the number of wins for each player. In the while loop, the function is just called twice, once for each player, and the returned score is saved into a different scoring value. As I am creating new variables for each player, this system is really not scalable at all over time. The various if-statements to check which player won, and the print statements to show the win rates are all hardcoded and are not scalable at all.

If I were to add a new player, I would have to increase the size of the array, add a new line of code to track the new player's score, add another "else if" branch to the if statement to check the winner, and add another print statement to show the new player's win rate. A good code structure would be dynamic enough to be able to adjust for the number of players without any changes in the code itself. Since the program still works, this poor code design proves itself to be a major technical debt.

Q2.2) Do you think your features were the right size? Too big? Too small?

From the features that I recorded (based off the business logic outlined in step 2), "simulate a game" was too large of a feature. This feature included too many mini-features inside and should have been broken down further. For example, in simulating a game, all dice had to be rolled, a random number of dice had to be rerolled (repeatedly), the dice had to be checked and the turn was to be ended if there were 3 or more skulls, and the score was to be calculated. Aside from the score, the other three things that were listed above were all grouped into this large feature, "simulate a game."

Breaking this feature down into those components would have definitely helped in organizing the workload as it was being committed to the GitHub repository. If an error should have occurred, for example the final score of the player was not calculated correctly, we have no idea which part of the round simulation would have caused it.

Q2.3) Is it worth tracking the realization of each feature in the backlog? As a tag in VCS?

I believe the tracking of the realization of each feature in the backlog was very important as many features were dependant on one another, and tracking this change proved to be a very good organizational tool. For example, the "calculate score" feature was dependant on the "simulate game" feature, which was dependant on the "roll a die" feature. These dependencies helped organize the timeline for these features, and when they should be completed. Tracking the realization of each of these features helped organize and follow this timeline.

Secondarily, any issues that were created in these features were easily identifiable on the GitHub commit history as the date of the feature realization on the backlog matched the commit date on GitHub. This cross-referencing helped traceback issues on the large project timeline.

Q2.4) Does it make sense to sacrifice quality in the long run? For short-term?

It makes total sense to cut corners and sacrifice quality during a short-term period, for example while developing a small prototype or test, but not long-term such as when production time comes. Cutting corners allows for the current prototype to finish earlier, permitting earlier recognition of any fatal flaws in the current design (given that the corner cut was not the reason the prototype failed). Spotting these poor design choices earlier on in the process allows for more time to reassess and improve the design without much time loss in the actual project timeline.

Continuing to sacrifice quality in the long-run leads to certain issues never being addressed and the final viable product not being produced. Unless there is a lack of time and no room for implementing the best quality work, cutting corners needlessly will of course ruin the product.

Q2.5) What are your plans to reimburse this debt during the next iteration?

On the next iteration, to address the scoring system, I would utilize a switch statement to handle the scoring of each possible die face individually. This would allow gold and diamond faces to score different points if necessary in the future. I would also implement an array that tracks the number of times each face is seen in the eight dice which would allow for x-of-a-kind scoring systems.

With reference to the two-player feature technical debt, I would implement dynamic data structures such as an ArrayList with a Player class. The Player class would have a “current score” property, “number of games won” property, and a “total games played” property. Since ArrayLists are dynamic, I can add as many Players to the list as required and can easily iterate through the list to access their score values, as well as their win rates.

Q3.1) What are the pros and cons of delivering the MVP first and then repaying your debt?

Typically, technical debt is accepted when there is a simpler and faster but costlier solution to a certain problem. Implementing these simpler solutions can help the developer provide valid deliverables within tight timeframes. That begin said, due to their unscalable nature, it is crucial that this technical debt be paid as soon as possible.

Since solutions to technical debt are more complex and advanced than the debt themselves, it is always a great idea to deliver the MVP first and then repay the debt. If the integration of the advanced system were to cause an error (which is very likely), it is great to have a checkpoint of the product that is functional and recent.

Even if the debt is paid off successfully without errors, it is quite likely that later in the future, an even better solution will present itself. In that case, it is also significantly easier to go back to another MVP before the first solution was implemented and work upwards from there again.

All in all, there really aren't any reason to not deliver the MVP before paying debt as long as the debt is dealt with soon after and not forgotten about. Having something functional is very persuasive when it comes to making you not want to continue working but having this backup checkpoint in your work would really never hurt.

Q3.2) Could you have delivered your MVP with less technical debt? If so, what prevented you from doing so?

The MVP definitely could have been delivered with less debt since the implementation of my solution for the debt was delivered very soon after with no issues at all. The solution was quite straight forward, and I had no hard deadline that was pushing me to quickly deliver the MVP.

However, implementing this improved code structure to manage the various players would have ruined the purpose of creating a *minimal* viable product. Even though the implementation only took a short amount of time, I was effectively able to prototype and test my game before progressing forward and making mistakes later on in the development cycle. I know I would be able to fail *faster* if I delivered my MVP with more technical debt in a shorter time period. This **trade-off** was the main reason that prevented me from undergoing less technical debt.

Q3.3) What are the pros and cons of using logging mechanisms instead of print statements?

Print statements are logged within the terminal itself alongside all of the text that the player must use to navigate through the software. Having logging occur within the console begins to crowd the user's experience with the program with lots of unnecessary information that the player might not need to see at the time. For this reason, it is really useful to have logging mechanisms that can log these outputs to an external file when required.

Another reason is purely the ease of management for these log files when using logging mechanisms. Rather than the user saving terminal outputs, the user can just access and store the

external log files produced by the mechanism with ease. When the software itself closes, these files will continue to exist and can be sorted and revisited at any time in the future.

Q3.4) Introducing Log4J is not a feature. It does not add business value to the product. How could we track the progress of such development activities?

The addition of Log4J was an improvement to the code structure allowing the developer to have more tools of debugging and understanding their code as it continues to grow. This can be compared to the implementation of any solutions to technical debt in the program. These solutions are often large-scale implementations that may take several weeks to implement. Though these solutions do not result in the addition of new features, tracking their integration progress can also be useful. For this reason, I believe having a secondary log that tracks backend improvements for the code such as the payment of technical debts, or for example, the implementation of Log4J could be really useful. This would not be dictated through guidelines posted by the client, rather deadlines instated by the developer.

In the scope of this project, I will use GitHub tags as my method of tracking these backend changes. Previously, the addition of a new tag was introduced when all the observed technical debts were paid, and now another tag was creating for the addition of the logging tool.

Q4.1) Were you able to deliver all of the features during this iteration? If yes, was it challenging? If not, how did you decide which features could be pushed to the next iteration?

With two iteration of 30 minutes, I was able to implement all of my features for this step of the project, and yes, there were parts of it that were terribly challenging.

Some features that were easy to implement were the new scoring system and using this new scoring system in determining who the winner was. Since the structure for the dice and its faces was well defined in an OOP sense, the counting process for each face on the die was quite simple and quick to implement.

The original plan was to implement the combo strategy (F07) and the players use of this new strategy (F08) in this same iteration. However, as I soon realized, due to my poor code structure, adding another dice rolling strategy was a lot more challenging than it should have been. I was able to combine code from my new scoring system function and the random dice roll function to create a new function for this feature. However, determining which player would utilize which strategy was a lot more complicated when they were two different functions for gameplay. For this reason, I decided to push F08 into the next iteration and I knew that it would require some code restructuring where I would break down some of my code and rebuild it.

This push for the feature into the next iteration was quite important for me as it acted like a checkpoint before I began to deconstruct my code which would potentially introduce major bugs.

Moving forward, I would most likely use this same methodology to determine which features should be pushed into the next build iteration. Knowing which features would require code to be dismantled and put back together with high risk should be isolated and dealt with individually.

Q4.2) What is the status of your technical debt?

Bad. As of right now, the strategy implementation for the two types of players is in grave debt. The current system contains two new functions for each strategy, one to manage the dice choosing, and one to simply call the right dice choosing function. This repeats a lot of code and quite inefficient if we were to add even more strategies into the mix.

That being said, I have already generated several solutions to the debt, including the use of lambda statements to classify the dice rolling functions as one type. This would allow for the use of these functions as essentially objects within our code structure, slowly shifting towards a more generic program.

The current system for the user to determine which player strategy they wish to implement into the players is also in debt. The current logic checks each input and creates a new player identity based on the entry. This logic is hard coded in for two players and can easily be replaced with a for loop that reads through all of the inputs. The only issue as of now is that the number of players is still unknown, making it difficult to merge the strategy choosing and trace enabling features together into one.

Q4.3) How did the object orientation of your code support the introduction of new strategy and scoring mechanism?

This question was already touched upon in Q4.1, with the object-oriented approach for the dice allowing for an easy addition for the scoring mechanism. The poor object-oriented design for the players did result in quite some difficulty in adding the new strategies, resulting in some technical debt as well in order to complete the features within the allotted iteration time (soft deadline). The object-oriented approach really provided organization and structure to the code, making it easy to develop upon and improve.

Q5.1) What was the impact of introducing the card deck into your simulator?

Adding the card deck into the simulator made a lot of huge technical changes to the code. Since the card deck made a difference on the players strategies, how the game calculates the final scores, and how the game determines which cards are being used for which rounds, a lot of the original game logic had to be changed in order to implement this new feature. With most of the other features that were added to the game, the features were more so extensions onto the code rather than modifications to existing code, which is why this addition was a large change.

Q5.2) What drove your choices if you had to mix features form the previous step with ones from this step?

Considering that the previous features were already committed in a functional state, I was more willing to modify the object-oriented structure of the old feature's code when it came to implementing the new features. For this reason, I decided to reorganize the scoring system and the player class to include the new card features and was not too hesitant on rewriting some of the code. This allowed for much more modification rather than simple appends.

During this process, I did however make a mistake which lead to the player incorrectly making moves throughout their turn while implementing the "sea battle" strategy. I was simply overcomplicating the process and I had determined an easier, less error prone, method of implementing the feature. For this reason, I went back to my GitHub commits and reinstated the functionality of me previous code and worked upwards again, which finally allowed me to implement the feature without many more problems.

Q5.3) Are you still in technical debt in terms of code quality? If yes, how to reimburse it? If not, how did you avoid it?

With the introduction of the card deck, I have once again gone into technical debt with its constructor method, as well as the general layout for the cards themselves.

In terms of the constructor method, the Card Deck class rigorously adds each card into the depending on the current needs. For example, since 29 NOP cards and 6 sea battle cards are required for this deck, 4 loops are being run: one to add 29 NOP cards, and three to add two sea battles cards each with different target outputs (how many sabers give you a bonus score). This is a really inefficient way of building a card deck, with no real capability for extensions. For this reason, adding parameters that can help determine what kind of card deck is to be built can be really useful for the future.

Another technical debt that was I was fortunately able to avoid was tons of data leakage with getting and settings the cards from the card deck class object. With the use of an "original deck" variable, I was able to keep a copy of the original deck after cards had been drawn and removed from play. This made it easy for the game to rebuild the cards (collect the cards) and shuffle when required during the game. With only one simply draw() function that return a card (and removes it from the deck using pop()), no data from the deck was being spilt over to the main game.

Q5.4) Which object-oriented mechanism are you using to support the switch between the “combo” strategy and “sea battle” one inside your player?

The use of an abstract class and lambda statements allowed me to treat each player strategy as an object that could be passed around from one player to another and changed from one to another within the player themselves. With this implementation, at the start of the game, the user can choose which strategy each player will use and that that player will then be passed the correct strategy method to use during the PlayerMove() call.

Adding new strategies in this structure was also very easy since it only required one new method to be added, outlining the general logic of the strategy, and an if statement that allowed the user to assign this strategy to the various players.

Q6.1) Describe from a coarse-grained perspective how the remaining cards will be implemented in your simulator if you have a couple more iterations. What would the technical difficulty of each one be?

With the dice (gold and diamond), the skull, the captain, and the sorceress cards left to implement and an iteration for each type of card, I would integrate them into the game in the following way:

Dice Cards: The current method of counting the faces from the dice to create a 5-element integer number that represents the number of occurrences for each face. In the case of a gold or diamond card, the corresponding index value would just be increased by 1 before the faces on the dice are counted.

Skulls Cards: These cards would be implemented similarly as the dice cards. When checking for skulls in the CheckSkulls() method, I would pass the player card as a parameter and if the card is a skull card, the corresponding value of skulls (card.target) would be added to the number of skulls in the count.

Captain Cards: The captain card doubles the score of the user which is quite simple to implement as well. When calculating the score in the CalculateScore() function, right before returning the score value, if the card.getFace() (passed in as a parameter) returns CardFace.CAPTAIN, the score would be doubled.

Sorceress Cards: Since sorceress cards allow the user to re-roll a skull. I would first set the target value of the sorceress card to 1, and once a skull is rolled, would decrement and change tis value to a 0. This would allow the card to only be used once. Within the players game logic itself, whenever it encounters a skull, if the card target value is greater than 0, it would re-roll the die and decrement its target value.

Treasure Chest: The treasure chest card is, in my opinion the most technically difficult card to implement. For this card, I would create a new arrayList parameter under the Player class that stores integer values corresponding to the index of each of the eight dice. If the value is not in the list, the respective dice is rollable, but if it is, it would be stored in the "treasure chest." If the player is disqualified, their score for the dice whose index is in the array (foreach index in list) would still be added to their score. The array would then be cleared (clear()). There would also be a separate function similar to the RollDie() function that accepts index values and adds those indices as to the list.

Q6.2) Your client is seriously annoyed that the "Island of Skulls" mechanism is unavailable. What would be the impact of introducing this mechanism in your code?

In order to implement the "Island of Skulls" mechanism into the game, I would have to create a second if statement that checks if the CheckSkulls() method return a value greater than 1 before checking if the CheckSkulls() method was equal to 0 (meaning the player can rolls again). In this if statement, another function would be called where the player keeps rolling skulls in a while loop, each skull adding to a counter. Once the player rolls something that isn't a skull, the game loops through each player in the *players* list and decrements their score by the appropriate value based on the number of skulls removed. (*numSkulls* * 100 [200 for captain card]). This would then end the players turn and the game continues as normal. This addition would really be

an feature where code is simply added and existing code is not modified which means it has a pretty minimal impact with low implementation risk.

Q6.3) Are you still in debt? How does this debt influence the upcoming development?

As I have learned throughout the course of this project, it's almost impossible to not be in at least some debt, and so yes, I am still in debt – the current logging system in the code is quite terrible ☺. The logging system at the moment only exists within the main PiratenKapern class. If any information from outside that class (ex. Player's rollStrategy function), no traces can be made unless the logger and the trace function can be passed through as parameters. As the game continues to grow with the addition of game logic such as the "Island of Skulls" mechanism, logging will become very useful for the user as a lot of decisions will be made. Since the logic really can't exist within the main PiratenKapern class for organization and efficiency reasons, this still needs to be fixed before moving on with upcoming development.

A good solution to this would be to have a class withint a package that could be imported by all other classes (SystemSettings.class) that would contain the *logger*, *trace*, *players* list, *numberOfGames*, etc. values for the simulation. Making these values static would really help keep the game consistent with universal settings that can be followed and checked by all scripts.

Bonus) What was the impact of implementing the bonus extensions to your code?

Due to the code structure that was organized previously, the bonus extensions were not too impactful when implementing as outlined in Q6.1. The addition of PMD did point out many redundant code pieces that could easily be changed. There were lots of modifiers that were unnecessary, such as the public modifier on the interface's abstract methods. This did also help me better understand the relation my objects had within my project. The most change probably came from the addition of the commons CLI command line handler, since it entirely replaced the command line parsing system I had previously created.