2AA4 Assignment 1 Report

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.

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 <u>could actually be rolled on the defined die</u> and also matches the original game rules.

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.

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.

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.

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.

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.

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 easy iterate through the list to access their score values, as well as their win rates.

Fixing Debts:

- Added Player class
    - Used for loops through the ArrayLists to edit all the values and keep track of the scores

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.

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.