**Name: Khazhismel Kumykov** **Partner:**

1. How long did it take to complete the complete task? Do you think this was longer or shorter than you would have completed the assignment if you did not use TDD, but did write tests after you coded?

I think I spent an hour so one day, and then about another hour in a half today. I wasn’t really sure how long it was going to take, so I don’t know if it took longer than expected. I can, however, say, I did get that “suddenly everything is done” you mentioned in class. (Although I did manage to craft a few tests after that moment when trying to break malformed number detection). I think it would have taken me a similar amount of time just going in and writing like normal. My “normal” coding style is trying to implement a solution, then testing if the partial implementation works, and continuing. I think one of the benefits of this was that, as you continue, you have more and more tests proving the robustness of the code, and making sure you didn’t break what was previously working.

1. Did you follow the TDD approach strictly? If not, how did you deviate from the practice?

I tried my best to follow it as closely as possible, but I didn’t exactly go one test at a time. At first, when I was wrapping my head around the problem, I did go one test at a time, but I approached a “local minima” where I had some terrible code that would only get worse as I continued that I didn’t really have a clue how to refactor. I paused and continued another day. On the second day I re-read the standard for the Roman Numerals a split the project up better – rather than going “1 to 10, then 10 to 100” as I was doing, I split it up into the ones places, tens places, hundreds places, and was able to figure the refactoring so once I implemented the ones place, and the “detection” between places, it was all done. Once things got going and I was implementing per “place” I started writing multiple tests for the “place” feature and then implementing to get one of those tests working at a time – which isn’t strictly TDD, but I felt it was a logical grouping and flowed better than writing a test, switching context, writing the implementation, switching context, remembering what part of the feature you didn’t cover yet, writing the test, etc. Also, close to the end I wrote a bunch of tests that didn’t fail in my effort to find something that would break it.

1. Did you work with a partner? If so, how well did pair programming work with TDD?

No.

1. Do you think your code is correct? Are you more confident that it is correct than you were with the pair programming assignment? Explain your answer to support your conclusions.

I am very confident in this code just because of how many tests it has. I am of course worried that there was some possible invalid combination that I would ‘accept’ incorrectly, but in terms of parsing valid Arabic and roman numerals, and handling most invalid combinations, I’m totally confident. I have 58 tests backing me up. ☺ In terms of completeness I am more confident in this program than the pair programming assignment because this program is split into “parts” that are all tested in one way or another, and I know the parts individually work, so for example if I know 1-9 works, and I know 10,20,30 etc. works, since those are the same part just combined, I know that 11-19 etc. works, and so on. Compared to the graph program we just had whatever tests we could happen to think of, and we didn’t really have discrete parts so to say. It was just one big thing.

1. What problems did you encounter with TDD? What, if anything, worked well for you? Provide details.

The whole implement one small thing at a time was a bit frustrating at first. I didn’t like it at all. I wanted to think over the problem and thing of an all-encompassing implementation from the beginning, not focus on getting 1 to work, then 2, then 10. But that was partially just me not exactly understanding what TDD was about, and it was helpful to take a grand overview, identify a way to split it up, and then work on those split features. In other words, at first, I split up the problem incorrectly.

Once it was split up correctly, the writing a test then writing an implementation went pretty smoothly. Writing tests to try and break it was also really good. For example, when writing `spacesInWeirdPlaces` the feature I was going for was “don’t accept spaces in the middle of numbers,” and I honestly fully expected it to work without changes because parseInt wouldn’t like it, but then it turned out that my roman numeral parsing didn’t see anything wrong with it and accepted it anyways. So I learned that if there isn’t a test for that feature, it really does not exist. ☺

1. Would you consider making TDD a regular development practice? Why or why not? Support your conclusion.

If your problem is properly segmented it’s really smooth and works really well. If you have a bad approach to a particular problem (as I did) it’s really easy to get stuck though.

For logic code, yeah I’d probably consider TDD as it just ends up feeling smooth once you get to the final product.

However, there’s a fair amount of stuff for which testing is just not easy, like UI, or going into client-server interactions mocking server responses etc. just gets icky. But that’s not just a strike against TDD, that’s just against automated testing in general. If there are good automated testing tools, I’d consider TDD because it just feels more robust in the end.