Angel Ramirez

For my implementation of DePaCoG, I chose to revolve it around an abstract class called DesignPattern and user input. The abstract class contains attributes and methods I saw all or most design patterns having in common. Such attributes are total functions, a list of subclasses, and a list of parameters received from user input. The main and only abstract method that all design patterns had in common in my implementation was a method to parse out the user input in which all design patterns had to do but all in their own way. The other methods were getters for the user input parameters and the subclasses that were created. The user input part of my implementation is that users of my program are prompt to input the desired design pattern and then asked a series of questions depending on the design pattern. Answers to the questions will then be parsed out to create the files for the desired design pattern if the chosen design pattern is implemented. The implemented design patterns are found in the config file where the design patterns in the program are set to true if implemented. The program notifies the user if the chosen design pattern is not implemented or not. If for whatever reason the user inputs an invalid input while answering the questions for the chosen design pattern, the user is taken to the main prompt, the select design pattern menu. This is achieved through the use of try and catches. Errors and information is logged on the console to specify what was wrong and what key parts went right.

I did not use any outside modules besides the ones we were asked to use to log, set up configurations, and unit test. My program can be broken down as DePaCoGMain which is the main module for this program found in the java directory, the design pattern classes found in the designPattern directory, and finally the Tools and myConstants classes found in my consts directory. Tools and myConstants classes are half of the heavy lifters of this program because they display the correct instructions to users and take care of getting user input. Tools essentially prompts the user to input the values needed to create the design pattern files and myConstants takes in those inputs once parsed out by the chosen design pattern's class and then creates the correct stubs and files. The next level up, the design pattern classes, are the other heavy lifters. They take care of parsing out the user input and using that information to then use methods in myConstants to create the correct stubs and and files. Then at the top is the DePaCoGMain which is set up in a Factory Method style and creates the correct design pattern according to the user's selection. Designing DePaCoGMain in a Factory Method style allowed me to take advantage of the simple way of creating an instance of the chosen object needed, in this case the chosen design pattern, to create the files for the design pattern.

After the completion of my program I am still unsure if my implementation's pros outweigh the cons. The cons of my program is the heavy use of string formatting. I did my best to create methods that will eliminate repeated code or variations of the same code but there were parts in some of the design patterns that did not follow the same format so I could not exactly functionalize the code and as a result I had to write it (mainly talking about the String.format() calls). I tried to use String.format()'s ability to take in an array as the parameters to less the amount of String.format() calls towards the end but ran into some issues that some functions in myConstants required the need for parameters at the moment of calling them and that the inserting calls to the array were so much more than just calling the String.format(). I was

also too into this program to make major modifications to reduce the calls to String.format() and I was also unsure if it would have a major impact on the reduction of code so I continued with the String.format() calls but with a mix of using the array approach for the formatting when it was more appropriate to do so. I chose to format my strings because I thought it would be the best way to organize myself and create a Container little by little. A Container is an object/interface. It contains all necessary information to create the correct file with the right information for the .java file. It holds the information of whether it is a superclass, abstract class, regular class, implements, extends, function amount, attribute amount, and most importantly it has the text that will be written onto a file. In other words it contains the blueprints to the class/interface that is going to be created. The Containers are a pro about my implementations as they are objects that tell methods in myConstants and in the design patterns what to put in each Container's text to then properly create the file. The biggest con though is that my config file is big and takes into account many possibilities of stubs and that in turn resulted in myConstants being bloated with the same amount of static String values which were used for the string formatting throughout my program. At the time I thought this was the best way to take on this program because then that would mean that I could just use those constant stubs in every design pattern without having to write them every time manually and then format. With the constant String values I was able to take advantage of the stubs being prewritten and then just get the user input to place in the spaces left for formatting. With the constants I was also able to just focus on the design pattern classes being in charge of using the input gotten from the Tools methods to sort the parameters out to then just create the proper stubs for the containers for the chosen design pattern. I feel that my plan was well organized but not very well implemented as there is some repeated code. I feel the most disorganized due to the many calls to String.format().

For testing I tested that the design patterns did in fact parse out the data correctly and created the right Containers either as the main object/interface of the design pattern or a subclass. Since my program mainly creates stubs I was unable to write tests to see if the design patterns actually worked. Despite this, I manually tested the created files as if I was an actual user and then put in the necessary information in methods for example and then created instances of them in a separate main and things worked as expected as they should according to the design pattern. Then again this was a manual test which will result in users having to believe that the program creates the design pattern as expected. I was only able to test the main methods of this program which were the parsing and the creation of the files. I did not test to see if the correct strings were in the files but that is expected to work since from the manual tests the results were positive. The parsing tests worked so it could be assumed that the creation of the files are correct based on the mock input parameters and my manual tests backup my reasoning that my program creates the design pattern files correctly.

Overall I feel very accomplished as I was able to catch my mistakes in the beginning and write useful methods and organize my code to the best of my ability. I feel that my code is readable but will probably lose readers with the many calls to String.format(). Despite this, I feel that the organization is good and won't lose readers if they were to want to implement the other design patterns using my current code and modules.