IDT Program Documentation

The program can be run via command line using java -jar [args]. The program takes in the required types of inputs necessary for exploratory black-box testing.
In order to search the large search space, the program implements a genetic algorithm to pursue the greatest paths of interest for a user.
The following classes compose the algorithm:

1. AdvancedTester.java
    a. AdvancedTester.java is the entry point into the Genetic Algorithm. The initial search space is created with a population size of 50 consisting of random bounded/unbounded Strings, integers, and doubles. The parameters for the initial search space were generated using ParameterFactory.java. The initial search space is additionally fuzzed (while maintaining parameter bounds) so that it is completely randomized.
2. CodeCoverage.java
    a. CodeCoverage.java is essentially a wrapper class that handles coming up with a "fitness" scores given a Collection<IClassCoverage> instance. It does this by going through and checking which lines had how much instruction and branch coverage, adding to an overall array of coverage by line. A large fitness bonus is provided for any coverage on a line with previously no coverage, otherwise a small penalty (scaled quadratically) is assigned based on how many times that line has already been executed; this serves to punish stagnation in the "gene pool". It also provides a large fitness bonus for discovering "new" errors (checking if the StdErr output has already been seen before)
3. Gene.java
    a. Gene instances contain a list of ParameterWrappers and a CodeCoverage instance, it is primarily a struct used for keeping these things together in the Genetic Algorithm.
4. GeneticAlgorithm.java
    a. GeneticAlgorithm.java contains the main algorithm that we use; it's a fairly standard genetic algorithm implementation with a set population size. It works by progressively performing one iteration on the current population to yield a new one. Each iteration consists of finding the minimum and maximum fitness scores, scaling the others to these, then filling all the slots in the new population by selecting members of the old population based on their fitness scores (percentage of the total sum becomes selection probability). Then, to "breed" two members, there is a set chance that they "crossover", which involves swapping their parameters, and then they "mutate", which is simply fuzzing the parameters.

5. IOHandler.java
    a. IOHandler.java handles all user input (by using CommandLineParser) and YAML output (by using the YAMLBeans library). IOHandler.java also runs basic tests, through a Tester object, as well as security tests, through an AdvancedTester object, which ultimately runs the Genetic Algorithm.
6. Main.java
    a. Main.java is the entry point into our program. The Main class instantiates an IOHandler object which handles basic tests and security tests. The Main class also has a static long to keep track of the start time of the program so that the Genetic Algorithm can be time constrained.
7. ParameterWrapper.java
    a. ParameterWrapper.java is a wrapper class that contains a Parameter object and a current value for that object (whatever's currently being used, this gets manipulated by the Genetic Algorithm). In the case of a formatted string, it contains a List<Object> representing all the values to be replaced into the format string, in the case of any other parameter type it contains an Object which contains a unbounded value (i.e String, double, integer). The wrapper class was created to facilitate fuzzing and manipulation of parameters, while keeping track of what the last thing we tried was (necessary for a Genetic Algorithm).
8. Tester.java
    a. Tester.java runs basic tests initially extracted from the black-box jar through reflection. It is called by an IOHandler object.
9. TesterHelper.java
    a. TesterHelper.java is a class containing various parameters to be evaluated for testing purposes. It is instantiated in Tester.java for tests to be carried out.

Requirements Met:

1. Our solution uses Java 7, and is compatible with Java 8 as well.
2. Our solution tests Java jar file executables via the command line.
3. Our solution increases code coverage of the supplied Software Under Test (SUT) via using a Genetic Algorithm for exploratory black box testing, we tested this on the provided sample jar executables and found that it did a satisfactory job of increasing code coverage.
4. We found that our solution does increase the number of unique exceptions generated by the SUT via exploratory black box testing through using the Genetic Algorithm.
5. Our solution does accept the path of the executable jar to test as a required command line argument in the form of: -jarToTestPath <jar to test path here> as handled by IOHandler using a CommandLineParser.

6. Our solution accepts the path to the directory where jacoco will generate output files as a required command line argument in the form of: -jacocoOutputPath <jacoco output dir path here> as handled by IOHandler using a CommandLineParser.
7. Our solution accepts the path of the jacoco agent jar in the form of: -jacocoAgentJarPath <jacoco agent jar path here> as handled by IOHandler using a CommandLineParser.
8. Our solution executes the specified number of exploratory black box test iterations (default 1000) against the SUT as is bounded by the run method in GeneticAlgorithm.
9. The solution runs additional exploratory black box test iterations, if the specified number of exploratory black box test iterations completes within a specified test time goal (default 5 minutes) as handled by the IOHandler which uses a CommandLineParser. The solution ensures that the Genetic Algorithm is time constrained by keeping track of the current time, as well as the start time for the program.
10. The solution accepts the number of exploratory black box tests to run as an optional command line argument in the form of: -bbTests <number of tests here> as handled by the IOHandler which uses a CommandLineParser.
11. The solution accepts the test time goal as an optional command line argument in the form of: - timeGoal <number of minutes here> as handled by the IOHandler which uses a CommandLineParser.
12. The solution supports applications that take a fixed number of arguments as handled by ParameterFactory and the initial search space generated in AdvancedTester.
13. The solution supports applications that take a variable number of arguments as handled by ParameterFactory and the initial search space generated in AdvancedTester.
14. The solution supports integer arguments through extraction from ParameterFactory as well as having a wrapper class so that fuzzing also supports integer arguments.
15. The solution supports double arguments through extraction from ParameterFactory as well as having a wrapper class so that fuzzing also supports double arguments.
16. The solution supports bounded String arguments through extraction from ParameterFactory as well as having a wrapper class so that fuzzing also supports bounded String arguments..
17. The solution supports unbounded String arguments through extraction from ParameterFactory as well as having a wrapper class so that fuzzing also supports unbounded String arguments.
18. The solution records unique exceptions or crashes seen during exploratory testing by keeping a Set<Exceptions> object in Genetic Algorithm.
19. The solution prints to stdout a YAML report at the end of testing that is in the following format:
    - Total predefined tests run:
    - Number of predefined tests that passed:
    - Number of predefined tests that failed:

- Total code coverage percentage:
- Unique error count:
- Errors seen:

20. The solution accepts an optional command line argument of the form "-toolChain" that will limit the output to contain only the parseable YAML indicated above as handled by the IOHandler which uses a CommandLineParser.
21. The solution is delivered as an executable jar with the name com.idtus.contest.winter2017.framework.jar.
22. The solution is also delivered as an Maven Project in Eclipse (zipped up) with the name com.idtus.contest.winter2017.framework.zip.