CS4218 Milestone 2 Report

Team_15

CAO SHENGZE
KIM HYUNG JON
YAP HAN CHIANG

1. Implementation Details and Assumptions

- **Some assumptions in the whole project**
    - In all applications, the arguments in the run() method do not include the command name.
    - Our test environment is windows, which uses \r\n as newlines in files. Our project aims to behave as similarly as Linux shell commands as possible. Hence, we use \n as newline char in this project. For some applications, we would just replace \r\n with \n when we need to output a \n to the outputStream. Therefore, sometimes the output would have both \r\n and \n. We think it is fine as we just regard the \r as a meaningless char. Therefore, some of our outputs may differ from other groups that do not take this difference between operating systems into consideration. This is not a bug.
    - Any arg that begins with ‘-’ is considered an option, even if it is a valid file name.
    - For most interfaces, even though we do not change the interface itself, but the meaning of the interfaces functions as well as the usage of it may change a lot. The details would be in the following description of each certain application.
    - As of Milestone 2, it seems our program may fail several test cases under Ubuntu because of some file name problem, as we all use windows, we have not tested under Ubuntu. But surely all test cases pass under our windows system.
    - The test cases for TDD for each application is inside <name>ApplicationTest: e.g. “CdApplicationTest”.
    - The test cases for more test cases as well as the integration test is inside Extra<name>ApplicationNameTest: e.g. ExtraGrepApplicationTest.

- **main**
    - The role of the main function is to read inputs, process each input line to obtain an array of string, and pass each string in the array to the parseAndEvaluate function.
    - Notice that the semicolon operator is evaluated inside the main function. Therefore, the test for the semicolon operator need to relay on the static function instead of using the parseAndEvaluate.

- **parseAndEvaluate**
    - Here, each string is processed for pipes and transformed into an array of string, and each string is passed to CallCommand.
    - The redirection functionality would also be evaluated here.
    - parseAndEvaluate is the function mostly used in test cases. The behavior of the shell can be performed with the parseAndEvaluate function.
    - The parseAndEvaluate function is also responsible for the processing of the pipeline.

- **CallCommand** helper class
  - We have a CallCommand helper class to parse each line into their individual components, which are the application name, the arguments, input file redirection and output file redirection.
  - CallCommand also do the processing of all three kinds of Quotes: The double quotes, the single quotes as well as the back quotes, which is the command substitution.
  - The globbing function would be implemented here.
  - CallCommands will call the runApp function of ShellImpl to let the shell implement the parsed command.

- **CatApplication**
  - Can use both stdin and the file reader to read the file.

- **CdApplication**
  - Changes the global variable Environment.currentDirectory
  - Does not deal with going to home directory (“cd”) or absolute paths. Does support going to parent directory (“cd ..”)

- **EchoApplication**
  - Can read from stdin only. Cannot read from files directly.

- **HeadApplication**
  - Similar to cat.
  - Only read the first several rows found in the file/stdin. Ignore the rest content.
  - Empty lines are still counted as lines.
  - Does not handle multiple files.

- **TailApplication**
  - Similar to head.
  - Count the number of lines of the input first, then decide which lines to print and which lines not.
  - Does not handle multiple files.

- **WcApplication**
  - **Interface changed:** original requirement is that each interface is given the whole command arguments as single string. For our implementation, the interfaces are invoked with the contents of the file or stdin as parameters.
  - One line is printed as output for each file in arguments. For valid files, the output line format is “[chars] [words] [lines] filename”, separated by single whitespaces. For invalid files, the output line format is “wc: filename: No such file”. The output lines do not begin with leading whitespaces.
  - Throws exception for invalid options. Does not throw exceptions for invalid files: as stated above, the error message should be part of ordinary output.

- Order of args (options and filenames) does not matter.
- Spaces(\s) and newlines(\n) are all counted as characters for option -m. Carriage returns(\r) are not counted. Lines that are empty or contain only whitespaces are counted as lines for option -l. Spaces and empty lines are not counted as words for option -w. This is following the logic of wc in Linux.
- When counting lines, the number of "lines" is counted, not the number of newline('\n') characters. Also, while Windows stores new lines as "\r\n", to simulate the behavior in Linux we drop the carriage return('\r'). This makes the char count less than if simply the length of the file bytes was returned. This is an intended behavior and not a bug.

- **DateApplication**
  - Invokes the java.util.Date class and prints it out using SimpleDateFormat class with the regex "EEE MMM dd HH:mm:ss z yyyy", allowing the date to be printed in format given in requirements. The time is printed in 24-hour format.

- **Semicolon Functionalities:**
  - For semicolon, what we do is before we parse the command line, we look for unquoted semicolon. If found ,then the command would be split to different commands and executed one by one.
  - This is done before the parseAndEvaluate function. All the commands get from processing the semicolon would then be passed to parseAndEvaluate one by one. We want to make sure the parseAndEvaluate would only focus on single command.

- **Quotes Functionalities:**
  - Similar to Semicolon, first look for all special symbols, then for the ones not quoted by other quotes(for example, back quotes would not work when it is inside a pair of single quotes.

- **Pipe Functionalities:**
  - Before passing the commands to the CallCommand function, the parseAndEvaluate functions would detect the valid | and then get a list of commands. With the help of pipeStream, we can easily use the output from one command become the input to the next command.

- **Globbing Functionalities:**
  - During the callCommand parse, all the * which is not instead a quote would be evaluated and replaced by the certain file names.

- **Redirection Functionalities:**
  - Inside the parseAndEvaluate, the redirection input and output would be found. If there is redir input or redir output, it would cover the pipeline inputStream and outputStream

    - It can be covered by the given file. In other words, only no files provided, the redirection would work.

- **CommandSubsititution:**
    - For command substitution, we observed that since the output of the first application is used as arguments for the second. Since the second application would usually expect the arguments to be filenames, it will output/throw exception that file is invalid. This is okay, but we created some files that contain other file names, broken by newlines. When these files are read using cat, head or tail, the output will contain the filename which the second app can actually understand and process. This allowed us to try tests such as “sort `cat cmdSubFile.txt` “ which produce proper outputs.

- **SortApplication**:
    - For efficiency, we implemented a single sorting algorithm that works regardless of which types of characters are found in the text, which all 15 interfaces invoke. The run() method identifies the char types and invokes the correct interface. Therefore, we did not implement input validation in interfaces to check whether the passed file actually contains the supposed char types (e.g. sortSimpleNumbers checking if the file contains only lower-case alphabets and digits) because 1) it does not matter in actual execution since all interfaces use the same algorithm and 2) since run() does the input checking, an interface being passed a file that does not contain the exact char types it is supposed to process is an infeasible execution, and hence not worth testing.
    - SortApplication is implemented to follow the given interface. The methods do not take stdin as parameter. Therefore, stdin is obtained from the Application’s global variable stdin (called as this.stdin). Hence, prior to testing an interface in isolation the global stdin should be set by calling the method run() or setStdin().
    - If multiple files are given, the contents are put together and sorted.
    - If any file is invalid an exception is thrown.
    - If sorting numerically, only the starting sequence of numbers (if any) will be treated numerically. For example, “line 30” is split to [l] [i] [n] [e] [ ] [3] [0], while “21th” is split to [21] [t] [h].

- **GrepApplication**:
    - The args in the given interface refers to the content from the input file, instead of the command itself. It would make this application easier to implement.
    - When no matches found, the output would be empty. No new line there.
    - According to the project description, it does not state it is necessary to do the reorder of the arguments, therefore, the first argument must be the pattern.
    - As multiple files provided, the grep would not throw an exception when a single file fails, instead, it would write a no file message to the outputStream.

    - Only the first parameter would be recognized as pattern, the rest would be regarded as filename automatically.

- **SedApplication**:
    - We have not changed the interfaces in this function. So all the interface methods are remaining the same. The command would be parsed twice: The first time run function would decide which certain api should be used (read from stdin or file, all or the first one). The second time of parsing is implemented inside each certain interface function and generate the result from the interface function.
    - As the design for the sed takes use of the interface cannot throw exception, therefore, sometimes the error message would just be generated as text instead of throw an exception.
    - For first match, there would always be a new line at the last line. But for the global one, it would not generate a newline at the end. If the original file does not contain a newline, it would just remain that thing.
    - For other certain assumptions, can check our test cases provided.

- **CalApplication**:
    - Only accepts “cal” command in lowercase
    - Month is entered as a number, not word (e.g. 6 instead of june)
    - Months 1 to 9 do not have a leading zero.
    - Interface for **printCalForMonthYear** and **printCalForMonthYearMondayFirst** receives 2 arguments: month and year instead of just a string argument

2. Test-Driven Development Plan

2.1 First iteration of Unit test cases

- For developing the EF1 features, we started with the tests we had written on Milestone 1.
- Like in Milestone 1, we used Linux virtual machine to try out the original versions of the commands we are trying to implement, and used the results as expected values for our assertions, while keeping in mind the difference between the actual commands in Linux and our implementation.. We divided the tests (including those mentioned above) into a few groups, based on the order of complexity required for the application under test. For example, Sort would be tested initially with strings of similar length and character type composition (e.g. “line 1\nline 2\nline 3\nline 4”). Then, as Sort is developed, it would be tested with more and more complex tests such as empty strings, strings that are slight variations of the other(e.g. “Abcd”, “aBcD”), strings that would be ordered differently under numerical sort and non-numerical sort (e.g. “3rd”, “10th”), strings which are identical until one ends first (e.g. “hello”, “hello, world!”), number strings that have the same numerical value but different string values (e.g. “21”, “0021”), etc.

2.2 First generation of code

- Once a reasonable set of tests was obtained, we started developing the application. First we programmed the core function, so that the application is able to read the input and produce an output that we expect to be correct. Then the application was tested with the first (=simplest) test sets, and if it failed we identified what was causing the failure, fixed the bug and tested again. We also sometimes identified bits of code that were redundant or excessively complex as well, by observing the statement coverage. Once the first set passed, we moved on to the next, repeated the process, and continued until all tests pass.
- After pass all the testcases written by ourselves, we come to generate more test cases to increase the code coverage rate. We also used the tests downloaded from IVLE, with many changes to fit our implementation. For apps like wc for example, we did not implement the interfaces in the same way as the provided test assumed. Some tests like WcApplicationTest had different logic for counting characters and lines as well. We changed the expected values to fit our implementation. These tests were put into separate test files, with names prefixed with “Extra”. (e.g. ExtraSortApplicationTest) Unit tests testing individual interface methods and some of the integration test cases were also put into these files.

2.3 Iteratively generate test cases (both unit and integration) & fix bugs

- For the new bugs found, fix bugs until it passes all the new generated test cases as well.
- After making sure all the until tests succeed, we proceeded to generate Integration test to try to test the interactions between different parts.

- After integration test, we carried out some system test by running the main method of the ShellImpl. If bugs were found, the steps were recorded new test cases written that can reveal the bug.

2.4 Randoop help with more regression test

- After all the above steps finish, we use Randoop to generate more test cases. Fix bugs found in error-revealing test and add some regression tests for our applications.

2.5 Code Coverage

- Code coverage is something quite important here. We use the code coverage tool to help us find out which statements have not be covered. We have try to maximize the code coverage. For some applications such like the Grep and Sed, the coverage rate is already 100%.
- For some other applications which is not 100 percent, here is the reason that prevent us from reaching 100%:
    - An exception that cannot happen. Even though we have a potential exception but this exception has been checked by previous statements. For example, the given interface structure that requires combining the args together back into cmd and re-parsing it forces file validation to be done twice. Even though the files have been validated in the first parsing stage and can never throw exception the second time unless some very unexpected thing happens, we have to put try-catch blocks again. This reduces coverage.
    - The main function inside the ShellImpl. Main function cannot be tested using JUnit.
    - Some exceptions our Application throws that are quite hard to happen: A previously valid OutputStream crash when use in next time. Although we did try to emulate situations like this, it did not go very well and hence we left them uncovered.
- Besides the above three situations, we have try to write test cases to cover all statements.

3. Integration Testing

3.1 Plan

We drew some tables like the below one:

CS: Command Subsititution PIPE: pipeline RE: redirection of Input/Output Quo:Quote

CS	PIPE	RE	Quo	Sed	Wc	Sort	Grep	Head	Tail	Cat	Exception
T	T						T			T	T
T	T						T	T			
T	T		T				T				
	T		T				T				T

And we fill in the table when we do the integration test. Each row is a test case and each entry represents whether an application or command function is involved inside a test case.

We do not want to write all combinations as the number is too large. And with some functionalities such like pipe, there are infinitely many combinations we can generate actually. Therefore, we try to do it pairwise. We try to let each pair appears at least once.

3.2 Execution

In order to make it easy to generate the integration test cases, we write integration test for each application inside the Extra<name>ApplicationTest. For example, for the GrepApplication, we have test cases in ExtraGrepApplicationTest. In order to reach the requirements:

1. Chain of interactions integrate applications using command substitution (≥10 tests) Example: sort `cat files.txt | head -n 1`
2. Chain of interactions integrate applications using piping (≥10 tests) Example: cat text1.txt | sort
3. Chain of applications connected with at least two pipes (≥4 tests)
4. Chain of applications connected with at least two command substitutions (≥4 tests)
5. For each case above, at least one corresponding negative scenario e.g., file does not exist or execution of one application fails - what happens to the other application? e.g., piping or command substitution to an application or command that does not exist?

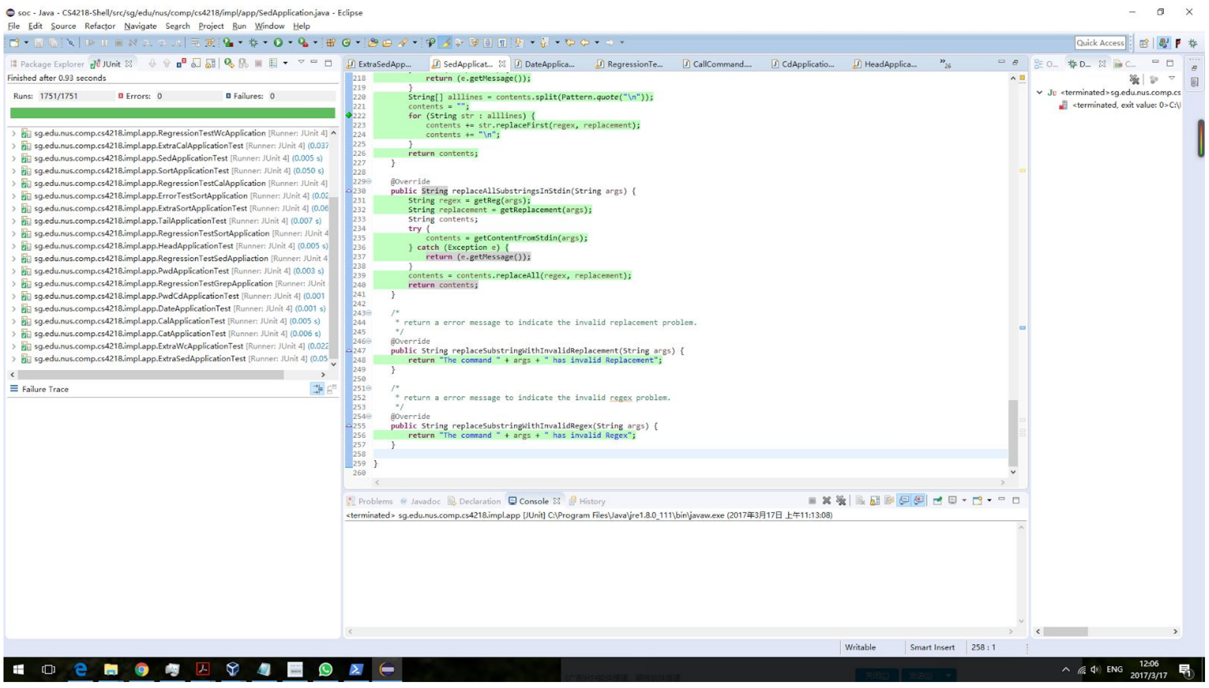
For the 1-4 requirements, we have 2-3 positive test cases and 1 test case which corresponding to the negative scenario as well. As we have tested among all applications, the total number is obviously larger than 10 for each requirements.

For each application under test, we tried piping and code substitution testing with various other applications to cover as many pairs as possible. Some applications that are normally not meant to interact with other applications, such as cd, pwd, cal, were tested less extensively for integration testing.

Besides the above four, we also have test cases regarding the redirection, quote and globbing.

The integration tests are always at the bottom of the Extra<name>ApplicationTest, for ease of finding.

3.3 Final Tests Result


soc - Java - CS4218-Shell/src/sg/edu/nus/comp/cs4218/impl/app/SedApplication.java - Eclipse

```
Finished after 0.93 seconds
Runs: 1751/1751    Errors: 0    Failures: 0

sg.edu.nus.comp.cs4218.impl.app.RegressionTestWcApplication [Runner: JUnit 4]
sg.edu.nus.comp.cs4218.impl.app.ExtraCalApplicationTest [Runner: JUnit 4] (0.037
sg.edu.nus.comp.cs4218.impl.app.SedApplicationTest [Runner: JUnit 4] (0.005 s)
sg.edu.nus.comp.cs4218.impl.app.SortApplicationTest [Runner: JUnit 4] (0.050 s)
sg.edu.nus.comp.cs4218.impl.app.RegressionTestCalApplication [Runner: JUnit 4]
sg.edu.nus.comp.cs4218.impl.app.ErrorTestSortApplication [Runner: JUnit 4] (0.02
sg.edu.nus.comp.cs4218.impl.app.ExtraSortApplicationTest [Runner: JUnit 4] (0.06
sg.edu.nus.comp.cs4218.impl.app.TailApplicationTest [Runner: JUnit 4] (0.007 s)
sg.edu.nus.comp.cs4218.impl.app.RegressionTestSortApplication [Runner: JUnit 4
sg.edu.nus.comp.cs4218.impl.app.HeadApplicationTest [Runner: JUnit 4] (0.005 s)
sg.edu.nus.comp.cs4218.impl.app.RegressionTestSedAppliaction [Runner: JUnit 4
sg.edu.nus.comp.cs4218.impl.app.PwdApplicationTest [Runner: JUnit 4] (0.003 s)
sg.edu.nus.comp.cs4218.impl.app.RegressionTestGrepApplication [Runner: JUnit
sg.edu.nus.comp.cs4218.impl.app.PwdCdApplicationTest [Runner: JUnit 4] (0.001
sg.edu.nus.comp.cs4218.impl.app.DateApplicationTest [Runner: JUnit 4] (0.001 s)
sg.edu.nus.comp.cs4218.impl.app.CalApplicationTest [Runner: JUnit 4] (0.005 s)
sg.edu.nus.comp.cs4218.impl.app.CatApplicationTest [Runner: JUnit 4] (0.006 s)
sg.edu.nus.comp.cs4218.impl.app.ExtraWcApplicationTest [Runner: JUnit 4] (0.022
sg.edu.nus.comp.cs4218.impl.app.ExtraSedApplicationTest [Runner: JUnit 4] (0.05
```

```java
                return (e.getMessage());
            }
            String[] alllines = contents.split(Pattern.quote("\n"));
            contents = "";
            for (String str : alllines) {
                contents += str.replaceFirst(regex, replacement);
                contents += "\n";
            }
            return contents;
        }

        @Override
        public String replaceAllSubstringsInStdin(String args) {
            String regex = getReg(args);
            String replacement = getReplacement(args);
            String contents;
            try {
                contents = getContentFromStdin(args);
            } catch (Exception e) {
                return (e.getMessage());
            }
            contents = contents.replaceAll(regex, replacement);
            return contents;
        }

        /*
         * return a error message to indicate the invalid replacement problem.
         */
        @Override
        public String replaceSubstringWithInvalidReplacement(String args) {
            return "The command " + args + " has invalid Replacement";
        }

        /*
         * return a error message to indicate the invalid regex problem.
         */
        @Override
        public String replaceSubstringWithInvalidRegex(String args) {
            return "The command " + args + " has invalid Regex";
        }

}
```

```
<terminated> sg.edu.nus.comp.cs4218.impl.app [JUnit] C:\Program Files\Java\jre1.8.0_111\bin\javaw.exe (2017年3月17日 上午11:13:08)
```

4. Code Quality: PMD, Eclipse Formatter

4.1 PMD

We tried to make our code pass the PMD check as much as possible. Such as:

- In assertion, we always followed the format “assertEquals(expected, result)”.
- We made sure that all if-elseif-else statements are used with curly brackets.
- In if-else statements, the if condition is always a positive comparison (== or isEquals, instead of != or !val.isEquals)
- We tried to keep methods short, and use variable names of reasonable lengths.

However at some points exceptions had to be made:

- We threw exceptions in catch blocks.
- Although PMD recommends against long methods, some methods, especially the run() methods of most Applications, could not be shortened much. If we felt that refactoring and hence breaking the code logic into bits would actually make the code harder to read, we did not follow PMD.
- Somehow, even though we added the locale setting “java.util.Locale.US” to DateFormat declarations, PMD still insists that locale should be set. We were unable to bypass this.
- PMD recommends avoiding short variable names, but we still followed some common practices like “InputStream is”, “OutputStream os”, “BufferedReader br”.
- And a few other exceptions, where we decided to favor code simplicity.

4.2 Eclipse Formatter

We ran Eclipse Formatter a few times during development as well.

Although not part of code quality requirement, a practice we followed for testing is that we always put assertions outside try-catch blocks. This is because if we put the assertion in the try block but an unexpected exception was thrown so the execution went into the catch block, then no error will be detected and the test will pass falsely. In the case we expect an exception to be thrown, we declared an Exception object exc beforehand, and in the catch block assigned the thrown exception to the Exception object using the statement “exc = e”. Then after the try-catch block we would compare exc.getMessage() with the expected error message. This allowed us to eliminate false positives (i.e. test passing when it should not) in our tests.

5. Randoop for Regression Test

5.1 Error-Revealing Test

The first thing that Randoop can help is to reveal potential errors inside the project.

According to the documentation of Randoop:

> When Randoop calls a method that creates an object, Randoop verifies that the object is well-formed. Currently, Randoop checks for the following contracts:
>
> - Contracts over Object.equals():
>   - Reflexivity: o.equals(o) == true
>   - Symmetry: o1.equals(o2) == o2.equals(o1)
>   - Transitivity: o1.equals(o2) && o2.equals(o3) ⇒ o1.equals(o3)
>   - Equals to null: o.equals(null) == false
>   - it does not throw an exception
> - Contracts over Object.hashCode():
>   - Equals and hashcode are consistent: If o1.equals(o2)==true, then o1.hashCode() == o2.hashCode()
>   - it does not throw an exception
> - Contracts over Object.clone():
>   - it does not throw an exception, including CloneNotSupportedException
> - Contracts over Object.toString():
>   - it does not throw an exception
>   - it does not return null
> - Contracts over Comparable.compareTo() and Comparator.compare():
>   - Reflexivity: o.compareTo(o) == 0 (implied by anti-symmetry)
>   - Anti-symmetry: sgn(o1.compareTo(o2)) == -sgn(o2.compareTo(o1))
>   - Transitivity: o1.compareTo(o2)>0 && o2.compareTo(o3)>0 ⇒ o1.compareTo(o3)>0
>   - Substitutability of equals: x.compareTo(y)==0 ⇒ sgn(x.compareTo(z)) == sgn(y.compareTo(z))
>   - Consistency with equals(): (x.compareTo(y)==0) == x.equals(y) (this contract can be disabled)
>   - it does not throw an exception
> - Contracts over checkRep() (that is, any nullary method annotated with @CheckRep):
>   - if its return type is boolean, it returns true
>   - it does not throw an exception

Hence, randoop is powerful for error revealing when the tested object is mainly a new object instead of a set of functions. For our Shell Application, most object behaves like a set of functions, therefore, the randoop is not quite helpful here. It even cannot find out the following error:

```
        if (args == null) {
                        throw new GrepException("No pattern found");
        }
        check(args[0]);
```

The above is part of the code from the GrepApplication. It thinks the args[0] is always existent as args is not null. But the args might be an array with length 0. In that case, the

OutOfBoundsException would be generated, which is not expected. However, the Randoop would not even regard this as an error, it generates the test case like following:

```
        try {
            grepApplication0.run(str_array23, inputStream24, outputStream25);
          org.junit.Assert.fail("Expected exception of type
    java.lang.ArrayIndexOutOfBoundsException");
        } catch (java.lang.ArrayIndexOutOfBoundsException e) {
            // Expected exception.
        }
```

In this case, it finds out the situation when the args is a length 0 array, but it regards the ArrayIndexOutOfBoundsException as an expected exception.

But this error still can be found, as in GrepApplication, it never handles ArrayIndexOutOfBoundsException, therefore, we can go through the generated Regression test and try to see whether there are any unexpected exceptions.

Besides this indirect way, the Randoop also helps us find a bug by the real error-revealing test.

(1) generated test:

```
    @Test
    public void test01() throws Throwable {

        if (debug) {
            System.out.format("%n%s%n", "ErrorTest0.test01");
        }

        sg.edu.nus.comp.cs4218.impl.app.SortApplication sortApplication0 = new
sg.edu.nus.comp.cs4218.impl.app.SortApplication();
        // during test generation this statement threw an exception of type
        // java.lang.NullPointerException in error
        java.util.ArrayList<java.lang.String> arraylist_str1 =
sortApplication0.getStdinContents();

    }
```

(2) buggy code fragment,

```
    public ArrayList<String> getStdinContents() {
        InputStream inputStream = new ByteArrayInputStream(baos.toByteArray());
        ….
    }
```

(3) applied fix,

```
    public ArrayList<String> getStdinContents() {
        if (baos == null) {
                // not supposed to happen in normal executon (i.e. infeasible path)
                // but can happen if for some reason
                // this method is called in isolation. Deals with it by creating a
                // new empty baos.
                baos = new ByteArrayOutputStream();
        }
        InputStream inputStream = new ByteArrayInputStream(baos.toByteArray());
        ….
    }
```

(4) command line options and other resources used to generate Randoop

```
    gentests --testclass=sg.edu.nus.comp.cs4218.impl.app.SortApplication --timelimit=5
--junit-output-dir=src-test
```

Inside the SortApplication, the function getStdinContents would use baos without checking whether baos is null or not. This would not happen in the normal execution, as the getStdinContents method should be used after the baos has been set. Therefore, in our test case, this error is not found. However, as a public method, it can be called in all kinds of scenarios. Therefore, it should deal with this situation even though it might be trivial.

We use Randoop for all of EF1 funcs and EF2 funcs, and the above is the only error-revealing test that we find. But this is really helpful.

For the error revealing test, we do not delete it after passing it. We leave it there with name SolveErrorTestSort as a regression test file.

5.2 Regression Test

Besides error revealing, Randoop is also helpful for regression test. But it is needed to be noticed that fail the Regression Test does not mean the new code has error! Most likely, it could be an error inside the previous code.

The Randoop would always record the current behaviors of the code, even the behavior has potential problems. For example, for this code:

```
    if (args == null) {
                throw new GrepException("No pattern found");
    }
    check(args[0]);
```

It is wrong as it does not deal with the situation what args is an empty array. If I generate RegressionTest and then change the code to

```
    if (args == null && args.length() != 0) {
                throw new GrepException("No pattern found");
```

```
        }
        check(args[0]);
```

It would fail the RegressionTest as in the RegressionTest, the OutofBound Exception is expected.

Therefore, the RegressionTest is quite helpful for finding the differences between the new code and the old one. But just notice fail RegressionTest does not mean the code is wrong.

Another interesting fact about the RegressionTest is the RegressionTest cannot work for DateApplication.

The Date would return current time but the RegressionTest would always fail as the behavior of Date always change.

Finally, we have RegressionTest for other 5 applications in EF1 and EF2:
RegressionTestCalApplication
RegressionTestSedApplication
RegressionTestWcApplication
RegressionTestSortApplication
RegressionTestGrepApplication

And these tests would help find the changes in the updates.