CS 362-400 SOFTWARE ENGINEERING II
Winter 2018
FINAL PROJECT
Jonathan Grocott (grocottj), Zheng Zhu (zhuzhe),
Drew Wolfe (wolfedr)

Final Project: Part B

Methodology Testing

Manual Testing

Manual testing was our initial opportunity to outline a testing strategy for UrlValidator.java. Because it tends to be the most labor intensive approach with the lowest coverage, it seemed to make sense, at the very least, to use it as an opportunity to break the otherwise large problem into more manageable pieces. We thought of the ***URL scheme*** and the **`UrlValidator()`** ***options*** as being the two multifaceted entities that essentially needed to be tested against one another. The *URL scheme* could be thought to be composed-of or including an overarching SCHEME, an AUTHORITY, a PORT, a PATH, and a QUERY. While there are a multitude of `UrlValidator()` *options*, we chose to focus on ALLOW_2_SLASHES, ALLOW_ALL_SCHEMES, NO_FRAGMENTS, and ALLOW_LOCAL_URLS.

The strategic combination of *URL scheme* and `UrlValidator()` *options* resulted in the following tactical construct for all manual tests:

```
// <URL SCHEME COMPONENT/OPTION #1>
	public static void testManualTest_<OPTION>(PrintWriter printResult) {

		ArrayList<Pair<String, Boolean>> testURLs = new ArrayList<Pair<String, Boolean>>();
		// VALID-OUTPUT TESTS
		testURLs.add(new Pair("<KNOWN VALID URL UNDER TEST>", new Boolean(true)));
		testURLs.add(new Pair("<KNOWN VALID URL UNDER TEST>", new Boolean(true)));
		testURLs.add(new Pair("<KNOWN VALID URL UNDER TEST>", new Boolean(true)));
		// INVALID-OUTPUT TESTS
		testURLs.add(new Pair("<KNOWN INVALID URL UNDER TEST>", new Boolean(false)));
		testURLs.add(new Pair("<KNOWN INVALID URL UNDER TEST>", new Boolean(false)));
		testURLs.add(new Pair("<KNOWN INVALID URL UNDER TEST>", new Boolean(false)));
		testURLs.add(new Pair("<KNOWN INVALID URL UNDER TEST>", new Boolean(false)));

		UrlValidator urlVal = new UrlValidator(UrlValidator.<OPTION>);

		printResult.println("testManualTest_<OPTION> STARTS\n");

		for (int i=0; i<testURLs.size(); i++) {
			String testURL = testURLs.get(i).getKey();
			printResult.println("\t" + testURL);

			boolean shouldBeValid = testURLs.get(i).getValue().booleanValue();

			try {
				boolean testValid = urlVal.isValid(testURL);
				printResult.println("\t\tShould be " + shouldBeValid
					+ "; Test result: " + testValid);
				if (shouldBeValid != testValid) {
```

```
                        printResult.println("\t\tError!\n");
                    }
                    else {
                        printResult.println();
                    }
                } catch (IllegalArgumentException | ExceptionInInitializerError
                | NoClassDefFoundError e) {
                    printResult.println("\t\tRuntime error: " + e + "\n");
                }
            }
            printResult.println("testManualTest_<OPTION> ENDS\n");
        }
```

The above construct continues to include all 5 URL scheme options and is subsequently reused for the remaining `UrlValidator()` options *B* through D, totalling at minimum of:
((5 *URL schemes* * 4 `UrlValidator()` *options*) * (≥ 1 VALID-OUTPUT TEST + ≥ 1 INVALID-OUTPUT TEST)) = ≥ 40 manual tests

While the sum of the manual tests' useful output is relatively low, especially in comparison to the subsequently employed methods, it helped to familiarize us with the general structure of the test's probable relationship with the CUT moving forward. Additionally it helped define a relatively sound foundation from which to approach additional, more efficient test methodologies.

Input Partitioning Testing
Input partitioning significantly benefited from the strategic approach defined during the development of our manual tests. The bottleneck associated with the manual testing is that the input domain consists of an enormous number of possible test cases. Manual testing attempted to provide coverage for what seemed to be the *most likely* use cases. However, it became clear through the definition of the input domain by parsing a URL into its respective scheme components as well as utilizing the `UrlValidator()` *options*, that manual test coverage would be woefully devoid of almost all edge-cases because of the tremendous labor involved with writing each test.

What we needed in order to dramatically increase our capability to cover the input domain was an automated means to generate individual URL scheme-components and a subsequent means to generate entire URLs from the discrete components. We began by developing between 10 and 30 basic cases for each component. Next came the powerhouse method `generateComponentList()`. This method accepts as parameters a component (for example, a top-level-domain list or `tld`) and an integer to define the number of URL scheme sections (so `(tld, 1)` could be `.com` while `(tld, 2)` could be `.academy.ca`). The URL is ultimately divided into six (6) sections. The AUTHORITY component is further divided into `hostWithoutTLD` (e.g. `www.google`) and the aforementioned `tld`. `generateComponentList()` utilizes the component base cases as models to be replicated using uppercase and lowercase letters, digits, special characters, and escaped special characters. The method adds new URL strings to the component list. Finally we tested the generated component lists with existing methods in the `UrlValidator` class such as `isValidScheme()` and `isValidPath()`.

While the advantage to this approach is the enormous increase in coverage it provides, both in the generation of VALID and INVALID URLs for the individual component-testing `isValidxxxxx()` methods, unfortunately it cannot test the `isValid()` method. That said, through the far greater coverage the approach inherently provides, it becomes much easier, using explicit failures, to essentially *triangulate* the location of an *implicit* bug through its dependencies.

Programming-based Unit Testing

The third testing methodology was driven off of the results found and lessons learned in manual and input partitioning testing. Strategically we chose to base our tests in this section solely on *ALLOW_ALL_SCHEMES* as the test case for the public constant in our construction of the UrlValidator. We began by generating the base cases for component lists from the valid URL components in our partition testing. Lists were generated from these components for each of the following; scheme, authority (hostWithoutTLD and TLD), port, path and query. Each of the hard coded value lists were selected to represent individual valid components so that when they were combined in nested loops, they would report the completed URL address results for each combination. Nesting each component in a lower level `for()` loop allowed a single test run to represent the full range of value combinations in the component lists as a combined address. These address strings were fed to their respective methods in the `isValid()` code base.

If a test failed to match the expected outcome the result would be printed along with the invalid address. We chose to only print and record the invalid results as the number of test iterations grew. We also included some exception handling in the test to report and avoid fault crashing. These included an `OutOfMemoryError` exception when generating the component lists, and the runtime exceptions: `IllegalArgumentException`, `ExceptionInInitializerError` and `NoClassDefFoundError` when feeding the combined component addresses to `isValid()`. If these exceptions were reached they would be added to another list, printed to the console and/or recorded in the output. Only one exception per type was output in detail, but each instance was counted and recorded. The last record is the bulk test results from this section in the form of errors per test performed.

Test Functions	**Methodology**
`public static void testManualTest_2_SLASHES(PrintWriter printResult)`	Manual Testing
`public static void testManualTest_ALL_SCHEMES(PrintWriter printResult)`	Manual Testing
`public static void testManualTest_NO_FRAGMENTS(PrintWriter printResult)`	Manual Testing
`public static void testManualTest_LOCAL_URLS(PrintWriter printResult)`	Manual Testing
`public void testScheme(PrintWriter printResult)`	Input Partitioning
`public void testAuthority(PrintWriter printResult)`	Input Partitioning
`public void testPath(PrintWriter printResult)`	Input Partitioning

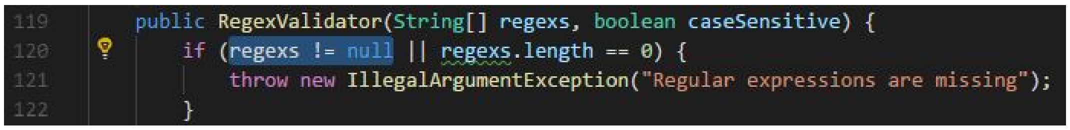
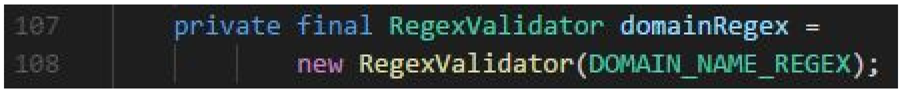
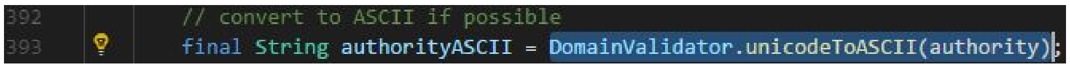
`public void testQuery(PrintWriter printResult)`	Input Partitioning
`public void testIsValid(PrintWriter printResult)`	Programming-Based Unit Testing

Bug Reports

BUG ID	BUG-01
BUG NAME	`isValidAuthority()` THROWS UNCAUGHT EXCEPTIONS FOR ALL TEST CASES
FAILURE DESCRIPTION	There is a failure in `isValidAuthority()`, potentially caused by a failure in the initialization of the `DomainValidator` class. Whenever `isValidAuthority()` is called with either valid or invalid authority strings, an uncaught `ExceptionInInitializerError` or an uncaught `NoClassDefFoundError` is thrown which crashes the program.

TEST CODE FOR REPRODUCING THE BUG	`UrlValidator urlVal = new UrlValidator();`
	
	`urlVal.isValidAuthority("amazon.com");`
	`// Exception in thread "main" java.lang.ExceptionInInitializerError`
	
	`urlVal.isValidAuthority("");`
	`// Exception in thread "main" java.lang.ExceptionInInitializerError`
	
	`UrlValidator urlVal = new UrlValidator(UrlValidator.ALLOW_2_SLASHES);`
	
	`urlVal.isValidAuthority("amazon.com");`
	`// Exception in thread "main" java.lang.ExceptionInInitializerError`
	
	`urlVal.isValidAuthority("");`
	`// Exception in thread "main" java.lang.ExceptionInInitializerError`
	
	`UrlValidator urlVal = new UrlValidator(UrlValidator.ALLOW_ALL_SCHEMES);`
	
	`urlVal.isValidAuthority("amazon.com");`
	`// Exception in thread "main" java.lang.ExceptionInInitializerError`
	
	`urlVal.isValidAuthority("");`
	`// Exception in thread "main" java.lang.ExceptionInInitializerError`

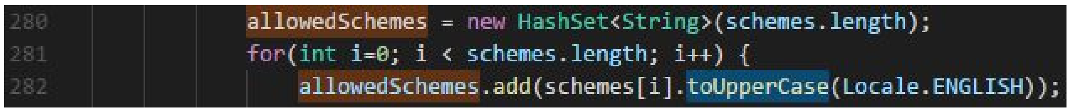
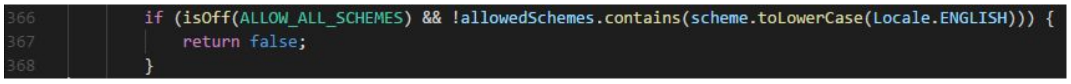
CAUSE OF THE BUG	The source of this bug lies on lines 120-122 in the `RegexValidator.java` file, which is part of a constructor for the `RegexValidator` class that incorrectly throws an `IllegalArgumentException`("Regular expressions are missing") when the input regexs is not null.

	
	`public RegexValidator(String[] regexs, boolean caseSensitive) {`
	`    if (regexs != null || regexs.length == 0) {`
	`        throw new IllegalArgumentException("Regular expressions are missing");`
	`    }`
	The `DomainValidator` class has a class variable `domainRegex` of type `RegexValidator` on lines 107-108 of the `DomainValidator.java` file, which is initialized by calling the `RegexValidator` constructor with a non-null `regexs`. As the `RegexValidator` constructor throws the `IllegalArgumentException` and `RegexValidator` does not have code to catch that exception, the `domainRegex` variable fails to be initialized, and thus the `DomainValidator` class cannot be initialized.
	
	`107     private final RegexValidator domainRegex =`
	`108             new RegexValidator(DOMAIN_NAME_REGEX);`
	The `isValidAuthority()` method calls the `DomainValidator.unicodeToASCII()` static method on Line 393 in the `UrlValidator.java` file. However, as the `DomainValidator` class cannot be initialized, this method call is doomed to fail.
	
	`392         // convert to ASCII if possible`
	`393         final String authorityASCII = DomainValidator.unicodeToASCII(authority);`
	As a result, an `ExceptionInInitializerError` or a `NoClassDefFoundError` is thrown and crashes the program.

BUG ID	BUG-02
BUG NAME	`isValidScheme()` RETURNS FALSE FOR VALID SCHEMES WHEN `ALLOW_ALL_SCHEMES` IS OFF
FAILURE DESCRIPTION	When a `UrlValidator` variable is initialized without the `ALLOW_ALL_SCHEMES` option, `isValidScheme()` returns FALSE for valid scheme inputs. For example, “http” and “ftp” are deemed as invalid schemes.

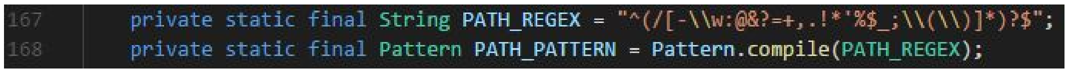
TEST CODE FOR REPRODUCING THE BUG	`UrlValidator urlVal = new UrlValidator();`
	`urlVal.isValidScheme("http");    // returns FALSE - should be TRUE`
	`urlVal.isValidScheme("ftp");    // returns FALSE - should be TRUE`
	
	`UrlValidator urlVal = new UrlValidator(UrlValidator.ALLOW_2_SLASHES);`
	`urlVal.isValidScheme("http");    // returns FALSE - should be TRUE`
	`urlVal.isValidScheme("ftp");    // returns FALSE - should be TRUE`
	
	`UrlValidator urlVal = new UrlValidator(UrlValidator.ALLOW_ALL_SCHEMES);`
	`urlVal.isValidScheme("http");    // returns TRUE`
	`urlVal.isValidScheme("ftp");    // returns TRUE`

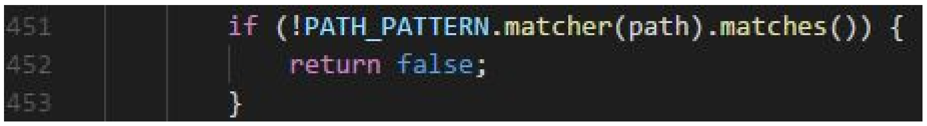
CAUSE OF THE BUG	The source of this bug lies on line 282 of the `UrlValidator.java` file, which is inside a constructor of the `UrlValidator` class. This line is executed when ALLOW_ALL_SCHEMES is not passed into the constructor options. It **mistakenly** converts all the items in the schemes array to UPPER cases and add them into the `allowedSchemes` set.

	 `280 allowedSchemes = new HashSet<String>(schemes.length);`
	`281 for(int i=0; i < schemes.length; i++) {`
	`282 allowedSchemes.add(schemes[i].toUpperCase(Locale.ENGLISH));`
	Then, on line 366 of the same file, when the method `isValidScheme()` is called with a valid scheme and without the ALLOW_ALL_SCHEMES option, `allowedSchemes.contains(scheme.toLowerCase(Locale.ENGLISH)` always returns false, since `allowedSchemes` contains only upper-case strings and cannot possibly contain any lower-case strings passed to the contains() method. Therefore, the if statement is true and `isValidScheme()` returns false.
	 `366 if (isOff(ALLOW_ALL_SCHEMES) && !allowedSchemes.contains(scheme.toLowerCase(Locale.ENGLISH))) {`
	`367 return false;`
	`368 }`

BUG ID	BUG-03
BUG NAME	`isValidPath()` RETURNS FALSE FOR ANY VALID TEST PATH THAT HAS MORE THAN ONE FORWARD SLASH
FAILURE DESCRIPTION	`isValidPath()` returns FALSE for valid paths with more than one forward slash, e.g. "`/test1/test2`".

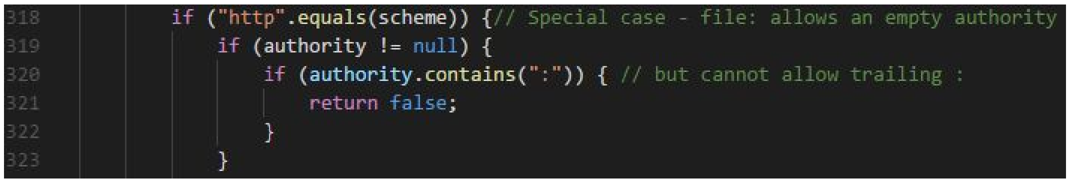
TEST CODE FOR REPRODUCING THE BUG	`UrlValidator urlVal = new UrlValidator();`
	`urlVal.isValidPath("/file1/file2");    // returns FALSE - should be TRUE`
	`urlVal.isValidScheme("/test1/");    // returns FALSE - should be TRUE`
	
	`UrlValidator urlVal = new UrlValidator(UrlValidator.ALLOW_2_SLASHES);`
	`urlVal.isValidScheme("/file1/file2");    // returns FALSE - should be TRUE`
	`urlVal.isValidScheme("/test1/");    // returns FALSE - should be TRUE`
	`urlVal.isValidScheme("//test1");    // returns FALSE - should be TRUE`
	
	`UrlValidator urlVal = new UrlValidator(UrlValidator.ALLOW_ALL_SCHEMES);`
	`urlVal.isValidScheme("/file1/file2");    // returns FALSE - should be TRUE`
	`urlVal.isValidScheme("/test1/");    // returns FALSE - should be TRUE`

CAUSE OF THE BUG	The source of this bug lies on line 167 of the `UrlValidator.java` file, which defines the regular expression for valid path strings: `^(/[-\\w:@&?=+,.!*'%$_;\\(\\)]*)?$`. It only allows one forward slash character which must be in the leading position of the test string and does not allow any more forward slashes.  `167 private static final String PATH_REGEX = "^(/[-\\w:@&?=+,.!*'%$_;\\(\\)]*)?$";` `168 private static final Pattern PATH_PATTERN = Pattern.compile(PATH_REGEX);` On line 451 of the same file, the `isValidPath()` method which compares the input string with the pattern represented by the above regular expression and returns false if they don't match. As a result, it returns false for all valid paths that have more than one forward slash.

	 `if (!PATH_PATTERN.matcher(path).matches()) {`
	`    return false;`
	`}`

BUG ID	BUG-04
BUG NAME	`isValid()` RETURNS TRUE FOR INVALID URLS WITH THE “`http`” SCHEME AND INVALID AUTHORITY SECTIONS WHEN `ALLOW_ALL_SCHEMES` IS ON
FAILURE DESCRIPTION	When a `UrlValidator` variable is initialized with the ALLOW_ALL_SCHEMES option, `isValid()` skips the testing for the validity of the AUTHORITY part of any URL candidate that has “`http`” as the scheme and returns TRUE if it finds no problems in any of the other parts of the URL. Examples include “`http://.ca.test,+;`”, “`http://.ninja.test,=;/`”, “`http://./80`”, “`http://co/`”, “`http://ac/file1`”, etc., all of which are invalid URLS with “`http`” as the scheme and an invalid AUTHORITY part, but `isValid()` returns TRUE for the above test strings.

TEST CODE FOR REPRODUCING THE BUG	`UrlValidator urlVal = new UrlValidator();`
	`urlVal.isValid("http://.ca.test,+;");    // returns FALSE`
	`urlVal.isValid("http://co/");    // returns FALSE`
	
	`UrlValidator urlVal = new UrlValidator(UrlValidator.ALLOW_2_SLASHES);`
	`urlVal.isValid("http://.ca.test,+;");    // returns FALSE`
	`urlVal.isValid("http://co/");    // returns FALSE`
	
	`UrlValidator urlVal = new UrlValidator(UrlValidator.ALLOW_ALL_SCHEMES);`
	`urlVal.isValid("http://.ca.test,+;");    // returns TRUE - should be FALSE`
	`urlVal.isValid("http://co/");    // returns TRUE - should be`

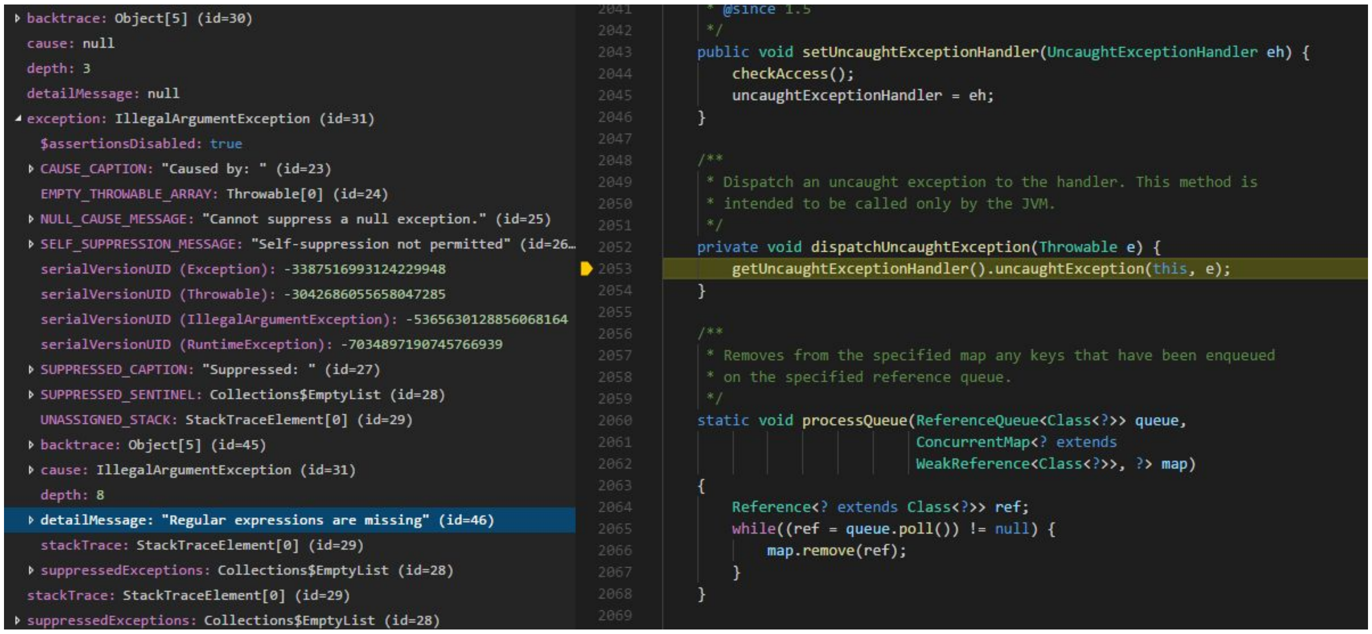
CAUSE OF THE BUG	The source of this bug lies on line 318 of the `UrlValidator.java` file, which is inside the `isValid()` method that we are testing. It sets the “`http`” scheme as a special case and skips calling `isValidAuthority()` to test those URLs with this scheme. Inside the if-block, it only returns FALSE for those URLs that have “`:`” in the AUTHORITY part, and there is no further testing for the AUTHORITY part. As a result, invalid URLs with “`http`” as the scheme and invalid AUTHORITY part may pass this test by mistake.
	
	`318        if ("http".equals(scheme)) {// Special case - file: allows an empty authority`
	`319            if (authority != null) {`
	`320                if (authority.contains(":")) { // but cannot allow trailing :`
	`321                    return false;`
	`322                }`
	`323            }`

Debugging

BUG-01

The collective output of all *manual tests* unfortunately does not provide much clarity on the root-cause of any single found bug… But, the output did provide hints. The *manual test* output, as it pertains to the "`isValidAuthority() THROWS UNCAUGHT EXCEPTIONS…`" bug, was surfaced via failures relating to the ALLOW_ALL_SCHEMES option. Upon further inspection of the exhaustive *input partitioning test* output, it became clear that ALL of the tests for the AUTHORITY parts failed and uncaught exceptions were thrown for all the cases when calling the `isValidAuthority()` method.

We utilized the `debuggingHelper()` function implemented in the `UrlValidatorTest` class by setting DEBUG_HELPER = `true` on line 18 and changing line 29 to `debuggingHelper("authority", 0, "google.com");`, which called the `isValidAuthority()` method with `"google.com"`as the passed argument. We then used the Java debugging tool of *Microsoft Visual Studio Code* to set a breakpoint on line 384 of the `UrlValidator.java` file, which was the first line inside the `isValidAuthority()` method. After the test program started running, it paused on the above line. We clicked "Step Over" to examine the result of executing each line and check where the exception would be thrown. It didn't take long to find that the exception was thrown right after line 393 was executed. Investigating the details of the exception variable on the left panel of the *VS Code* window revealed that the `ExceptionInInitializerError` was caused by another `IllegalArgumentException` that carried a message "Regular expressions are missing" (see the screenshot below). This was crucial information in the debugging process, as searching for this message among all the files of the project directly pointed to line 121 of the `RegexValidator.java` file. Further, as line 393 of the `UrlValidator.java` file called the `DomainValidator.unicodeToASCII()` static method and according to the type of exception thrown (`ExceptionInInitializerError`), it was not difficult to find that the `DomainValidator` class failed to be initialized due to the initialization error of one of its class variable `domainRegex`, which was caused by the unhandled exception thrown by the `RegexValidator` class discovered above. Therefore, we have traced to the source of this bug as described in the "Cause of the Bug" entry in the Bug Reports section.



```
▸ backtrace: Object[5] (id=30)
  cause: null
  depth: 3
  detailMessage: null
◂ exception: IllegalArgumentException (id=31)
    $assertionsDisabled: true
  ▸ CAUSE_CAPTION: "Caused by: " (id=23)
    EMPTY_THROWABLE_ARRAY: Throwable[0] (id=24)
  ▸ NULL_CAUSE_MESSAGE: "Cannot suppress a null exception." (id=25)
  ▸ SELF_SUPPRESSION_MESSAGE: "Self-suppression not permitted" (id=26…
    serialVersionUID (Exception): -3387516993124229948
    serialVersionUID (Throwable): -3042686055658047285
    serialVersionUID (IllegalArgumentException): -5365630128856068164
    serialVersionUID (RuntimeException): -7034897190745766939
  ▸ SUPPRESSED_CAPTION: "Suppressed: " (id=27)
  ▸ SUPPRESSED_SENTINEL: Collections$EmptyList (id=28)
    UNASSIGNED_STACK: StackTraceElement[0] (id=29)
  ▸ backtrace: Object[5] (id=45)
  ▸ cause: IllegalArgumentException (id=31)
    depth: 8
  ▸ detailMessage: "Regular expressions are missing" (id=46)
    stackTrace: StackTraceElement[0] (id=29)
  ▸ suppressedExceptions: Collections$EmptyList (id=28)
  stackTrace: StackTraceElement[0] (id=29)
▸ suppressedExceptions: Collections$EmptyList (id=28)
```

```java
     * @since 1.5
     */
    public void setUncaughtExceptionHandler(UncaughtExceptionHandler eh) {
        checkAccess();
        uncaughtExceptionHandler = eh;
    }

    /**
     * Dispatch an uncaught exception to the handler. This method is
     * intended to be called only by the JVM.
     */
    private void dispatchUncaughtException(Throwable e) {
        getUncaughtExceptionHandler().uncaughtException(this, e);
    }

    /**
     * Removes from the specified map any keys that have been enqueued
     * on the specified reference queue.
     */
    static void processQueue(ReferenceQueue<Class<?>> queue,
                             ConcurrentMap<? extends
                             WeakReference<Class<?>>, ?> map)
    {
        Reference<? extends Class<?>> ref;
        while((ref = queue.poll()) != null) {
            map.remove(ref);
        }
    }
```

BUG-02

After examining the results obtained from the Input Partitioning tests, we found that `isValidScheme()` always returned false if not supplied the `ALLOW_ALL_SCHEMES` option. Debugging was performed using strategic breakpoint inserted at the beginning isValidScheme() method in the `UrlValidator` class. Running through the test code that reproduced the bug, it was quickly apparent that the isValidScheme() method returned false for valid schemes after line 366 of the `UrlValidator.java` file was executed. Analyzing the conditions in the if statement yielded the conclusion that allowedSchemes did not contain the valid tested scheme in lower-case letters. Tracing the item-adding procedure of the allowedSchemes set gave us the answer that the faulty line of code lay on line 282 inside the constructor of the `UrlValidator` class. This line mistakenly converted all the items in the schemes array to upper-cases and added them into the `allowedSchemes` set. Once all of the scheme entries were converted to upper cases, hitting line 366 of the same file meant that the `isValidScheme()` method would always return false (see BUG-02, CAUSE OF THE BUG above).

BUG-03

Locating the culprit for this bug was easier than that for the above two bugs. After setting a breakpoint on at the beginning of the `isValidPath()` method on line 447 of the `UrlValidator.java` file and running the test code that reproduced the bug in *debugging mode*, we soon discovered that the condition of the if-statement on line 451 was not met, resulting in the method returning FALSE for any valid path that had more than one forward slash. This lead us to examine the condition in the if-statement and search for the definition of the PATH_PATTERN variable. We then quickly found that the regular expression on line 167 that defined PATH_PATTERN was insufficient to cover all the valid paths and clearly excluded those with more than one forward slash.

Team Work

1. Write about how did you work in the team?
    a. How did you divide your work?

    There are three team members, so we decided that we would initially divide each section, *Methodology*, *Bug Reports*, and *Debugging* into three parts essentially corresponding to a bug found by the methodology we employed. Of course this approach wasn't perfect. While the presence of bugs could be determined by manual testing, it was difficult to pinpoint the culprit method until more data was available, essentially through input-partitioning test output. We, for the most part, attempted to remain responsible for our own sections, but we would be remiss in saying that the workload was even… Zheng definitely pulled more than his share of the work by taking on both the input-partitioning test work as well as the integration of the other methodology tests into the final product. This involved some non-trivial refactoring as the output format (to-console vs. to-file) involved the encapsulation of tests within a previously non-existent `try` code block.

    b. How did you collaborate?

    The team uses a private channel (*6remlin-hunt3rs*) within a *Slack* workspace shared by a small group of peers. The team communicates approaches towards problems, scheduling, and the broad strokes of division of labor within the channel.

2. The contribution of each of your team members.

- **Jonathan Grocott**: Base testing report write-up. Test function to methodology table. BUG-02 writeup.
- **Zheng Zhu**: Implemented the input-partitioning tests and programming-based unit testing, helped Jonathan and Drew get their tests up to speed, provided the technical meat to all bug reports, and essentially wrote the debugging section.
- **Drew Wolfe**: Attempted to hold up his end of the work through the development of the manual tests, Bug Report for BUG-01, and Debugging BUG-01 and some of BUG-03. Of course he readily admits that he couldn't have done it without pretty significant help/insight from his teammates.