Final Project Part B
Aidan Grimshaw (grimshaa), Khuong Luu (luukh)

1. Methodology Testing (15 points)

1.1. Our general methodology

In general, our methodology combines varieties of methodologies: programming-based testing, partition, and manual testing. Shortly, we partition our input into test data, then use these test data to put into our programming-based test functions. These programming-based test functions then discover test failure. We then write manual test function using the test data that failed in order to address/narrow-down/debug these test failure to find and fix the bug; and repeat the process.

The steps specifically can be described in the following repeated steps:

0. Manually call URLValidator's method with common valid and invalid input to catch early and easy failure.

1. Partition the input to create test data for programming-based testing in step 2
2. From the test data created in step 1, we develop functions that programmatically use those test data to test and discover failed test cases
3. We pick the data that failed in step 2 and create a manual test to narrow down into the bug. If one manual test is not enough to narrow down the bug, we create more manual tests
4. We can use a debugger to find narrowed-down bug
5. We attempt to fix the bug, then re-run start the tests in step 3. If the tests passes, start over from step 2 to discover more failed test cases and repeat.

1.2. Our core test functions

Programming-based Testing
We have 2 core test functions for: **testIsValid** function is for ALLOW_ALL_SCHEME option and **testIsValidScheme** function the other for other scheme options. Both of these functions are inspired and similar to the ones in the final project A, yet we have modified them to fit our ideas and intentions.

Partition Testing

We don't have any particular, dedicated function for partition test. Instead, we present our planned partition as chunks of urls and use them to build complete urls. Each of these urls is a test case for corresponding partitions.

Below are 5 examples of our chunks and complete urls:

http://oregonstate.edu:80/t123?action=edit&mode=up
http://oregonstate.edu:65535/../?action=view
http://microsoft.com:#/test1/../file?action=edit&mode=up
http://0.0.0.0:80/../?action=view/..
http://abc.abc:#/test1?action=$

Manual Testing

We developed functions **testManualXX** where 'XX' is the test number. Whenever we need to check out our assuption, or to narrow down the bug, address the bug and help running the debugger over the failed test case, or just calling out some methods, we write a new testManualXX method (with the 'XX' number incremented)

Below are 6 examples of our manual test function that we used to narrow-down/address the bug

public void testManual1() {
UrlValidator urlVal = new UrlValidator(null, null, UrlValidator.ALLOW_ALL_SCHEMES);
assertEquals(true, urlVal.isValid("http://www.google.com:80/test1/?action=view"));
}

public void testManual2() {
UrlValidator urlVal = new UrlValidator(null, null, UrlValidator.ALLOW_ALL_SCHEMES);
assertEquals(true, urlVal.isValid("http://www.google.com:#/test1?action=view"));
}

public void testManual3() {
UrlValidator urlVal = new UrlValidator(null, null, UrlValidator.ALLOW_ALL_SCHEMES);
assertEquals(false, urlVal.isValid("http://www.google.com/..?action=view"));
}

public void testManual4() {
UrlValidator urlVal = new UrlValidator(null, null, UrlValidator.ALLOW_ALL_SCHEMES);
assertEquals(false, urlVal.isValid("http://256.256.256.256:80/test1?action=view"));
}

public void testManual5() {
UrlValidator urlVal = new UrlValidator(null, null, UrlValidator.ALLOW_ALL_SCHEMES);
assertEquals(true, urlVal.isValid("http://0.0.0.0:80/test1?action=view"));

```
  }
```

```
public void testManual6() {
        String[] schemes = { "http", "gopher" };
        UrlValidator urlVal = new UrlValidator(schemes, 0);
        boolean result = urlVal.isValidScheme("http");
        assertEquals("http", true, result);
    }
```

1.3. Our partitions

Similar to Apache's test suit, we create sub-partition by divide the structure of URL into parts, and the valid and invalid version of each part make a sub-partition. We then permulate sub-partitions together to generate a lot of partitions as test data to our programming-based test case.

Sub-Partition description	**Example(s)**
Valid scheme	http, ftp
Invalid scheme	hkt
Valid IP address/domain	google.com, oregonstate.edu
Invalid IP address/domain	khuongishot.xxx
Valid port	22, 80
Invalid port	-1, 65999999, a, %
Valid structure	http:/ww.google.com, http:///google.com
Invalid structure	http://google.com, www.google.com
Valid path	test1/test2
Invalid path	../.. , ../../
Valid query	?action=true
Invalid query	#$%-99

2. Coverage

2.1. Screenshot of the final coverage

We covered **77% of instructions and 55% of branches**

UrlValidator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
isValidAuthority(String)		79%		67%	7	15	7	30	0	1
isValid(String)		77%		55%	7	11	7	22	0	1
isValidPath(String)		86%		75%	3	7	4	15	0	1
UrlValidator(long)		0%		n/a	1	1	2	2	1	1
UrlValidator(RegexValidator, long)		0%		n/a	1	1	2	2	1	1
matchURL(String)		0%		n/a	1	1	1	1	1	1
getInstance()		0%		n/a	1	1	1	1	1	1
UrlValidator(String[], RegexValidator, long)		100%		100%	0	4	0	12	0	1
static {...}		100%		n/a	0	1	0	7	0	1
isValidScheme(String)		100%		100%	0	5	0	7	0	1
countToken(String, String)		100%		100%	0	3	0	8	0	1
isOn(long)		100%		100%	0	2	0	1	0	1
isOff(long)		100%		100%	0	2	0	1	0	1
isValidQuery(String)		100%		100%	0	2	0	3	0	1
isValidFragment(String)		100%		100%	0	2	0	3	0	1
UrlValidator(String[], long)		100%		n/a	0	1	0	2	0	1
UrlValidator(String[])		100%		n/a	0	1	0	2	0	1
UrlValidator()		100%		n/a	0	1	0	2	0	1
Total	61 of 417	85%	21 of 86	75%	21	61	24	121	4	18

2.2. Which tests had higher coverage, and why?

Although we used a mixed of all methods, we used programming-based method the most and it gives us the highest coverage. It would still give us the highest coverage had we decided to write most of our tests manually because through this method, we are able to inject a lot of sub-case/sub-partition and permulate them to build test case/test partition which obviously can accomplish more tests than writing manually urls one by one, which is also very ineffective in this program.

2.3. Point out some code, if any, that your “final” test cases do not cover. Why?

2.4. Which tests had better fault detection capability? (Be detailed and thorough!)

Since the programming-based method has not only highest coverage but also deepest coverage since it both robust in tern of amount of test data and also deep and sharp in term of the ability to test edge case just by putting data into the pool. So, again, had we decided to write random testing or other method, programming-based testing would still has better fault detection capability.

3. Bug Report

Note: Because of our programming-based testing methodology described above, all of the bugs have been caught by just several assertion positions. Thus, content of our second part of each bug report is certainly very similar to each other but enough to give us huge hints about the bugs.

Bug #1: Always return opposite test data logic

Failure
The URLValidator return false for valid urls and return true for invalid urls
How we found it/Test case that caught it
finalprojectB.UrlValidatorTest.testIsValid(finalprojectB.UrlValidatorTest) junit.framework.AssertionFailedError: **http://www.google.com/test1?action=view** at junit.framework.Assert.fail(Assert.java:57) at junit.framework.Assert.failNotEquals(Assert.java:329) at junit.framework.Assert.assertEquals(Assert.java:78) at junit.framework.Assert.assertEquals(Assert.java:174) at junit.framework.TestCase.assertEquals(TestCase.java:333) **at finalprojectB.UrlValidatorTest.testIsValid(UrlValidatorTest.java:150)**
Cause of failure
expected:<false> but was:<true>

Which code causes it?
Turn out there is a bug in ResultPair class contructor.
public ResultPair(String item, boolean valid) {
this.item = item;
this.valid = !valid; //Weather the individual part of url is valid.
}
 	
The bad code is
this.valid = !valid;
which should be
this.valid = valid;

Bug #2: Fail on http url that contain port and True on urls that don't

Failure
The URLValidator return false for URLs that contain port

How we found it/Test case that caught it
finalprojectB.UrlValidatorTest.testIsValid(finalprojectB.UrlValidatorTest) junit.framework.AssertionFailedError: **www.google.com:80** at junit.framework.Assert.fail(Assert.java:57) at junit.framework.Assert.failNotEquals(Assert.java:329) at junit.framework.Assert.assertEquals(Assert.java:78) at junit.framework.Assert.assertEquals(Assert.java:174) at junit.framework.TestCase.assertEquals(TestCase.java:333) **at finalprojectB.UrlValidatorTest.testIsValid(UrlValidatorTest.java:150)**
Cause of failure
expected:<true> but was:<false>
Which code causes it?

```
if ("http".equals(scheme)) {// Special case - file: allows an empty authority
          if (authority != null) {
             if (authority.contains(":")) { // but cannot allow trailing :
                return false;
             }
          }
          // drop through to continue validation
       } else { // not file:
          // Validate the authority
          if (!isValidAuthority(authority)) {
             return false;
          }
       }
```

The bad code is
if ("http".equals(scheme))
which should have been:
if ("file".equals(scheme))

Bug #3: Fail on valid domain names and True on invalid domain names

Failure
The URLValidator return false for URLs that contain valid domain names and return true for invalid domain names
How we found it/Test case that caught it
testIsValid(finalprojectB.UrlValidatorTest) Time elapsed: 0.002 sec <<< FAILURE! junit.framework.AssertionFailedError: http://www.google.com:80/test1?action=view

expected:<true> but was:<false> at junit.framework.Assert.fail(Assert.java:57) at junit.framework.Assert.failNotEquals(Assert.java:329) at junit.framework.Assert.assertEquals(Assert.java:78) at junit.framework.Assert.assertEquals(Assert.java:174) at junit.framework.TestCase.assertEquals(TestCase.java:333) at finalprojectB.UrlValidatorTest.testIsValid(UrlValidatorTest.java:173)
Cause of failure
UrlValidatorTest.testIsValid:173 http://www.google.com:80/test1?action=view expected:<true> but was:<false>

Which code causes it?
The failure is in method isValid of class DomainValidator
if (groups != null && groups.length > 0) {
return isValidTld(groups[0]);
}
 	
The bad code is
return !isValidTld(groups[0]);
which should have been:
return isValidTld(groups[0]);

Bug #4: Fail on URLs that contain queries

Failure
The URLValidator return false for URLs whose paths contain queries
How we found it/Test case that caught it
junit.framework.AssertionFailedError: **http://www.google.com:80/test1/?action=view** at junit.framework.Assert.fail(Assert.java:57) at junit.framework.Assert.failNotEquals(Assert.java:329) at junit.framework.Assert.assertEquals(Assert.java:78) at junit.framework.Assert.assertEquals(Assert.java:174) at junit.framework.TestCase.assertEquals(TestCase.java:333) at finalprojectB.UrlValidatorTest.testIsValid(UrlValidatorTest.java:174)
Cause of failure
expected:<true> but was:<false>

Which code causes it?
In URLValidator class: The bad code is **private static final String PATH_REGEX = "^(/[-\\w:@&?=+,.!*'%$_;\\(\\)]*)?$";** which should have been: **private static final String PATH_REGEX = "^(/[-\\w:@&?=+,.!/~*'%$_;\\(\\)]*)?$";**

Bug #5: Throw NoClassDefFound exception

Failure
The URLValidator throw NoClassDefFound exception
How we found it/Test case that caught it
http://www.google.com:80/test1/?action=view finalprojectB.UrlValidatorTest. testManual(finalprojectB.UrlValidatorTest) Time elapsed: 0.011 sec <<< ERROR! java.lang.ExceptionInInitializerError: null at finalprojectB.RegexValidator.<init>(RegexValidator.java:121) at finalprojectB.RegexValidator.<init>(RegexValidator.java:96) at finalprojectB.RegexValidator.<init>(RegexValidator.java:83) at finalprojectB.DomainValidator.<init>(DomainValidator.java:108) at finalprojectB.DomainValidator.<clinit>(DomainValidator.java:96) at finalprojectB.UrlValidator.isValidAuthority(UrlValidator.java:394) at finalprojectB.UrlValidator.isValid(UrlValidator.java:328) at finalprojectB.UrlValidatorTest.testManual(UrlValidatorTest.java:155) testIsValid(finalprojectB.UrlValidatorTest) Time elapsed: 0 sec <<< ERROR! java.lang.NoClassDefFoundError: Could not initialize class finalprojectB.DomainValidator at finalprojectB.UrlValidator.isValidAuthority(UrlValidator.java:394) at finalprojectB.UrlValidator.isValid(UrlValidator.java:328) at finalprojectB.UrlValidatorTest.testIsValid(UrlValidatorTest.java:172) at junit.framework.Assert.assertEquals(Assert.java:78) at junit.framework.Assert.assertEquals(Assert.java:174) at junit.framework.TestCase.assertEquals(TestCase.java:333) at finalprojectB.UrlValidatorTest.testIsValid(UrlValidatorTest.java:174)
Cause of failure
Exception thrown: NoClassDefFound
Which code causes it?

The constructor of RegexValidator has failed to be constructed which led the RegexValidator class failed to be initiated, which leds to ExceptionInInitializerError exception, which leds to NoClassDefFoundError in the class that called it.
The bad code is
if (regexs != null || regexs.length == 0)
which should have been:
if (regexs == null || regexs.length == 0) {

Bug #6: Throw NullPointerException exception

Failure
The URLValidator throw NullPointerException exception
How we found it/Test case that caught it
http://www.google.com:80/test1/?action=view testManual(finalprojectB.UrlValidatorTest) Time elapsed: 0.014 sec <<< ERROR! java.lang.NullPointerException: null at finalprojectB.RegexValidator.match(RegexValidator.java:165) at finalprojectB.DomainValidator.isValid(DomainValidator.java:164) at finalprojectB.UrlValidator.isValidAuthority(UrlValidator.java:413) at finalprojectB.UrlValidator.isValid(UrlValidator.java:328) at finalprojectB.UrlValidatorTest.testManual(UrlValidatorTest.java:155) testIsValid(finalprojectB.UrlValidatorTest) Time elapsed: 0 sec <<< ERROR! java.lang.NullPointerException: null at finalprojectB.RegexValidator.match(RegexValidator.java:165) at finalprojectB.DomainValidator.isValid(DomainValidator.java:164) at finalprojectB.UrlValidator.isValidAuthority(UrlValidator.java:413) at finalprojectB.UrlValidator.isValid(UrlValidator.java:328) at finalprojectB.UrlValidatorTest.testIsValid(UrlValidatorTest.java:172)
Cause of failure
Exception thrown: NullPointerException
Which code causes it? The constructor of RegexValidator has failed to be constructed which led the RegexValidator class failed to be initiated, which leds to ExceptionInInitializerError exception, which leds to NoClassDefFoundError in the class that called it. The bad code is **for (int i = 0; i < regexs.length-1; i++)**

which should have been: **for (int i = 0; i < regexs.length; i++)**

Bug #7: Throw NullPointer error (not exception)

Failure
The URLValidator throw NullPointer error (not exception)
How we found it/Test case that caught it
http://www.google.com:80/test1/?action=view testIsValid(finalprojectB.UrlValidatorTest) Time elapsed: 0.06 sec <<< ERROR! java.lang.NullPointerException: null at finalprojectB.UrlValidator.isValidAuthority(UrlValidator.java:416) at finalprojectB.UrlValidator.isValid(UrlValidator.java:328) at finalprojectB.UrlValidatorTest.testIsValid(UrlValidatorTest.java:171)
Cause of failure
Tests in error: UrlValidatorTest.testIsValid:171 » NullPointer
Which code causes it? In InetAddressValidator, line 68 The bad code is **return null;** which should have been: **return VALIDATOR;**

Bug #8: False the valid ip4 addresses

Failure
The URLValidator false the valid ip4 addresses
How we found it/Test case that caught it
http://0.0.0.0:80/test1?action=view testIsValid(finalprojectB.UrlValidatorTest) Time elapsed: 0.063 sec <<< FAILURE! junit.framework.AssertionFailedError: http://0.0.0.0:80/test1?action=view expected:<true> but was:<false>

at junit.framework.Assert.fail(Assert.java:57) at junit.framework.Assert.failNotEquals(Assert.java:329) at junit.framework.Assert.assertEquals(Assert.java:78) at junit.framework.Assert.assertEquals(Assert.java:174) at junit.framework.TestCase.assertEquals(TestCase.java:333) at finalprojectB.UrlValidatorTest.testIsValid(UrlValidatorTest.java:173)
Cause of failure
expected **http://0.0.0.0:80/test1?action=view** to be <true> but was:<false>
Which code causes it? In InetAddressValidator, line 88 The bad code is **if (groups != null) {** which should have been: **if (groups == null) {**

Bug #9: Mistake urls whose octals bigger than 255 as valid

Failure
The URLValidato mistakes urls whose octals bigger than 255 as valid
How we found it/Test case that caught it
http://256.256.256.256:80/test1?action=view testIsValid(finalprojectB.UrlValidatorTest) Time elapsed: 0.099 sec <<< FAILURE! junit.framework.AssertionFailedError: http://256.256.256.256:80/test1?action=view expected:<false> but was:<true> at junit.framework.Assert.fail(Assert.java:57) at junit.framework.Assert.failNotEquals(Assert.java:329) at junit.framework.Assert.assertEquals(Assert.java:78) at junit.framework.Assert.assertEquals(Assert.java:174) at junit.framework.TestCase.assertEquals(TestCase.java:333) at finalprojectB.UrlValidatorTest.testIsValid(UrlValidatorTest.java:173)
Cause of failure
expected **http://256.256.256.256:80/test1?action=view** to be <false> but was:<true>
Which code causes it?

In InetAddressValidator, line 107

The bad code is

```
if (iIpSegment > IPV4_MAX_OCTET_VALUE) {
                        return true;
}
```

which should have been:

```
if (iIpSegment > IPV4_MAX_OCTET_VALUE) {
                        return false;
}
```

Bug #10: Transform scheme to uppercase instead of lowercase

Failure
The URLValidator transforms scheme to uppercase instead of lowercase
How we found it/Test case that caught it
testIsValidScheme(finalprojectB.UrlValidatorTest) Time elapsed: 0.008 sec <<< FAILURE! junit.framework.AssertionFailedError: http expected:<true> but was:<false> at junit.framework.Assert.fail(Assert.java:57) at junit.framework.Assert.failNotEquals(Assert.java:329) at junit.framework.Assert.assertEquals(Assert.java:78) at junit.framework.Assert.assertEquals(Assert.java:174) at junit.framework.TestCase.assertEquals(TestCase.java:333) at finalprojectB.UrlValidatorTest.testIsValidScheme(UrlValidatorTest.java:187) testManual6(finalprojectB.UrlValidatorTest) Time elapsed: 0.008 sec <<< FAILURE! junit.framework.AssertionFailedError: http expected:<true> but was:<false> at junit.framework.Assert.fail(Assert.java:57) at junit.framework.Assert.failNotEquals(Assert.java:329) at junit.framework.Assert.assertEquals(Assert.java:78) at junit.framework.Assert.assertEquals(Assert.java:174) at junit.framework.TestCase.assertEquals(TestCase.java:333) at finalprojectB.UrlValidatorTest.testManual6(UrlValidatorTest.java:223)
Cause of failure
UrlValidatorTest.testIsValidScheme:187 http expected:<true> but was:<false>
Which code causes it?
In URLValidator, line 283 The bad code is

allowedSchemes.add(schemes[i].toUpperCase(Locale.ENGLISH));
which should have been:
allowedSchemes.add(schemes[i].toLowerCase(Locale.ENGLISH));

4. Debugging

4.1. Method of Debugging

When we found the failed test case, we need a way to find the bad code (the root cause that need to be fixed). Below is how we **debug the application by debugging the test class with IntelliJ IDEA debugger**

As you can see from section (3) above, most of the bugs have been caught by test cases with the test url **http://www.google.com:80/test1/?action=view**. So we wrote a manual test:

```
public void testManual1() {
        UrlValidator urlVal = new UrlValidator(null, null, UrlValidator.ALLOW_ALL_SCHEMES);
        assertEquals(true, urlVal.isValid("http://www.google.com:80/test1/?action=view"));
    }
```

and put it in the main function:

```
public static void main(String[] argv) {
        UrlValidatorTest fct = new UrlValidatorTest("url test");
        fct.testManual();
        //fct.setUp();
        //fct.testIsValid();
        //fct.setUp();
        //fct.testIsValidScheme();
    }
```

… so that we can use IntelliJ IDEA to debug the test.

Similarly, for each failed url, we create a manual test with test data as that url. Then we call that manual test in the main function and put a breakpoint at that line. After that, we ran the IntelliJ IDEA debugger and use the “Step In”, “Step Over”, “Force Step into”, and “Smart step out” to localize the bug.

4.2. Agan's rules we applied: For every bug, we applied Agan's rules to every debugging session to help us localize the bug and fix the bug effectively. In general, here are the Agan's rules that we used:

- We applied Agan's Rule #1 - Understand the system: The first things we did at very first was to read and understand the code in the final project A. Thank to the understanding we gained from the final project part A, we was able to partition the test data and debug the failed test cases very effectively in this final project part B.

- We applied Agan's Rule #2 - Make it failed: As can be seen in the previous sections of this report, we wrote our tests so that the program failed reliablity and reproducibly.

- We applied Agan's Rule #3 - Quit Thinking and Look: We let our test cases print out all neccessary information and don't hold back like the original version of Apache in the final project part A code base. Also, as we already mentioned and demonstrated, we used debugger to localize where it return true/false or throw error instead of merely looking at the code.

- We applied Agan's Rule #4 - Divide and Conquer: The fact that we used the debugger and partitioned test cases, we have applied this rule.

- We applied Agan's Rule #5 - Change One Thing at a Time: Each time we were certain and decided to make a fix, we changed one thing at a time. Luckily, in every bugging session below, every our first change fixed the bug.

- We applied Agan's Rule #6 - Keep an Audit Trail: We kept a journal file at src/finalProjectB/java/main/test/journal.md**.** Except the bug that involve bad regular expression, we didn't have many chance to write "failed attempt" into that journal file - only successfull attempts since the bugs in this proejct were fairly easy to find and fix. It was also because we were able to debug all the bugs we found in a single testing by the two of us together (again, since the bugs were fairly easy), we didn't have much to write down to remember for later session in the other days.

4.3. Debugging Details

Below is what we recalled that we thought while debugging each bug:

Bug #	Debugging details/journals/logs
1	This is the bug that both cost us most of the time and taught us a valuable lesson. When our test failed, we thought it was the component of url was incorrectly labelled. For example, http:// is true but we might have labelled it false. We spent nearly 45 minutes re-checking all of our test data and found nothing.

	So we applied Agan's rule #3: We started making the programming-based test function print all the necessary information it has; and it turned out that our test data were fine as we have rechecked. Next, we wrote a manual test with that failed url and start placing breakpoint here and there, step in and step out, until we found the bad code that negate the validity of the test data in the class ResultPair. We had never thought it would be here and thus didn't look at it since the main focus was in the URLValidator.isValid() method. We were wrong. The lesson learned is that all classes related to that class and its method must be tested, too. It's even more important to test ResultPair because our test class actually use it to build the programatic test cases. This lesson helped us a lot in debugging following bugs.
2	This was also a not-so-easy bug. Applying the same localization methodology, we found that the isValidAuthority function mistakenly failed every url we put in. But looking a little closer at the failed test cases, they only failed with http whose path contain port. So we put in similar url in manual test and debug it and quickly found that the scheme that should have been checked for "colon" was "file", not "http"
3	As usualy, the failed test case didn't give us much specific information besides the stack trace and where the program failed. We again pick a failed url and put it in a manual test to debug. We step in, step out, and restart the debugging session over and over again and finally found that the validity result was clearly negated in DomainValidator class. Bug fixed.
4	This is the bug that we would not have been able to fix if we hadn't look at the old code. We still don't know why the changed regular expression failed while the new one doesn't in which the only different is "\~" addition to the Regex pattern. The fact that this addition fixed the bug doesn't make sense to use since the "~" symbol doesn't have anything to do with match urls that contain query in its path. Eventually, we must fix this bug to continue on other bugs.
5	This bug taught us to catch any exception early to avoid the program throw a wild non-sense exception itself. The NoClassDefFound initially looks like an exception throw by not having a correct java path set up. But after roughly 20 minutes we couldn't find anything wrong with our program set up or importing statements. Thus, again, we followed the debugger into the class RegexValidator and found that an if-statement hasn't been handle correctly which leds to weird exception thrown out by the program.
6	This bug is fairly easy since the NullPointerException is very common in Java. We use debugger the localize the bug at a loop in RegexValidator class again. This time, the upperbound of the loop is bad - it was off-by-one loop which leds to invalid memory access, which caused NullPointerException
7	The NullPointer hinted us that the program might have used a Null pointer to do something. Using debugger, we located the bug at InetAddressValidator, then at its constructor. In this one, the value returned should have been VALIDATOR instead of

	null - just as we had suspected
8	This bug is similar to bug #1, which we were then familiar with. We quickly started debugging the InetAddressValidator since only URL with valid IP address failed. And we were right: There was negated if-statement that should have been groups == null instead of groups != null. Again, the flipping boolean login hinted us a lot this time.
9	We noticed only URL with octals IP greater than 255 failed. We immediately checked the InetAddressValidator again and this time, without using debugger, we found a very obvious bug: the validity checking function return true instead of false for octal IP greater than max value (255). Easy peasy!
10	We ran testValidScheme with varieties of scheme and restriction and found that most of the scheme failed except the empty one regardless of whether they are restricted or not. The first thing we thought of was there must be some correctness issue with the constructor of the URLValidator class. So we put a breakpoints in each of its constructor and found that the last constructor, which is responsible for passing the scheme to other validator, mistakenly transformed the scheme to uppercase instead of lowercase, which is required by other helper validator - otherwise they will fail the url, which they did. So we quickly changed the lowercase and this fixed the bug.

5. Team Work

Initially, we divided the work between partitioning the test (Aidan) and writing the programming-based tests (Khuong). In the stage we collaborate via Github and communicate via Google Hangout.

However, we later changed our approach and in our new approach, we realized that the test partitioning has to be done before we are able to start developing programming-based tests, we decided to divide the work but partitions by half and half (each of us took a half); then we worked closely together face-to-face in library to do pair-programming the programming-based test functions and find-fix the bugs together.

Contribution
Each of us planned and partitioned approximately half of our test partitions. Then we worked together to find and fix the bugs.

6. Citation

Our test suite is heavily inspired and inherited from Apache's programming-based test suit that we were introduced in the final project part A. Though we borrowed a bunch of (amazing) ideas

from the Apache implementation, we have modified, restructured, reformated, and partition the test data by our own ideas.