DITAA Testing Document

Overview

This document describes tests that were implemented as part of the analysis and extension of DITAA. Various types of testing were implemented and are documented in the following sections.

Our Contributions

- JUnit tests for four additional classes in DITAA
- A total of 14 additional text-file test cases, or ASCII diagrams which DITAA can be invoked on through the command line. This is the format of most of the pre-existing implementation's tests, and they are tested through visual inspection. They include our stress tests, and tests created in the course of identifying and repairing bugs in DITAA.
- Manually-conducted smoke testing and testing of the GUI we implement.

Acceptance Testing

DITAA previously had very few unit tests and no automated method of integration testing. A manual procedure could be developed to perform acceptance testing, but in the case of DITAA there is a single main concern with accepting a release--the correctness of the diagrams that the software generates. Evaluating the correctness of a set of diagrams is easily accomplished through an automated test.

Dataset

Acceptance testing is performed on 30 different inputs of varying complexity. The inputs have been designed to test each entity that DITAA supports on its diagrams, e.g. boxes, lines, dashed boxes, dashed lines, special shapes, arrows, color codes, rounded corners, bulleted lists, free text, etc. The diagrams range from a single shape to a complex diagram of several shapes, arrows, and different colors.

Implementation

The JUnit framework was used to develop an automated test (AcceptanceTestSuite.java) that submits every test text file in the tests/text project directory to DITAA. For each text file that is submitted, a corresponding PNG is also supplied that represents the expected output. These expected outputs have been verified manually. Adding a new test case requires two simple steps: 1) adding the test input file, and 2) adding the expected output diagram file. The automated test is implemented such that it will automatically include the new files when the test suite is run.

For each test input file, DITAA output is generated and stored in a temporary location on the filesystem. An MD5 hash is computed for both the actual and expected output graphic files and the assertion is made that these hashes are equivalent. While MD5 may be vulnerable to exploits, it is extremely unlikely that the hash of two different files will collide unless they are specifically designed to exploit such a vulnerability.

Testing for New Plugins

Additional Color Codes

Test case 32 was created to test 6 new color codes added to DITAA. It was evaluated visually, as most of the test cases are, and based on the results, some changes were made to the color codes' RGB values.

GUI Implementation

The GUI implementation was carefully architected to share the same codebase performing the DITAA ASCII-to-Bitmap with the command line interface. In this regard, all of the above tests are relevant to the GUI implementation with respect to the diagrams that are produced.

Testing the added functionality of the GUI widgets is a difficult task due to the need to mock a large number of widget classes. This is not generally feasible or desirable, so frameworks exist to simulate clicks and typing on the application. However, no such framework was integrated for this project.

If an automated GUI testing technology was to be used for testing the DITAA GUI, there would be only a few main features to test. Since we assume Swing widgets work as advertised and the layout of the application is 'static', i.e. a single window, single view, no menus, etc., the testing is simplified and limited to the following verifications:

1. The GUI starts when
    a. DITAA is run with no arguments
    b. DITAA is run with the --gui or -g argument
2. When no file (or an unreadable file) is selected, the Save As button is disabled.

3. When the Load file button is selected, a file chooser is presented
4. When a file is chosen as an input,
    a. The file contents populate in the left pane
        i. If the file is not ‘valid’, an error is shown instead
    b. Assuming a ‘valid’ input, (within an appropriate amount of time) the output diagram populates in the right pane
    c. Assuming a ‘valid’ input, the Save As button is enabled
5. When the Save As button is selected, a file chooser is presented
6. Choosing a file in the Save As file chooser results in the file being saved to the specified location.

Test Cases that Fail

There are clearly additional bugs in the DITAA codebase as several of the tests currently fail. While it is unfortunate that the cases are failing, it demonstrates some success by our testing efforts in identifying broken functionality within the application.

- In test case 8, a point marker is not rendered.
- In test case 10, a line is not connected to the document it clearly connects to in the ASCII. The document has a special shape (document), which doubtless makes this difficult to ensure.
- In test case 13, there are barely-visible ‘extra lines’ continuing past where the line is supposed to curve into a corner.
- In test case 22, numerous point markers are not rendered.
- In test case 27, an incorrect break appears in a line joining two squares.
- In test case 28 the same incorrect break occurs.
- In case “Bug 10”, one horizontal line is not aligned with other horizontal lines beginning in the same column of the ASCII.
- In “Bug 13”, the same misalignment occurs.
- All of the cases named “OutOfMemoryError”, except for the version named “.edit”, appear to result in indefinite hanging. We are not so sure this is actually an issue of running out of memory, as the stress tests actually result in a complaint from the JVM but these cases do not.
- Case “Garbage” results in a RuntimeException.
- The “Stress Test” and “Less Stress Test” result in OutOfMemory errors.

Stress Testing

To perform stress testing on DITAA requires diagrams with a large number of entities, i.e. a large input file. To accomplish this stress testing, the acceptance test dataset includes three files considered to be abnormally large - tiny_stress_test.txt (30KB), less_stress_test.txt (300KB), and stress_test.txt (29,893KB).

We create the test case ‘Stress Test’ by tiling Test Case 1 to create a file 10,000 times the size of Test Case 1. Attempting to run DITAA with this test input results in an OutOfMemory error from the JVM. We progressively scale down this test, creating ‘Less Stress Test’, and ‘Tiny Stress Test’, the last of which is only 10 times the size of Test Case 1, and successfully runs to completion in 67 seconds on a powerful home workstation.

We conclude that DITAA is not architected around the possibility of large inputs. In particular, the creation of numerous intermediate AbstractionGrids is likely a source of great memory inefficiency, and the execution of repeated flood-fills makes the program inefficient in time on large ASCII art as well.

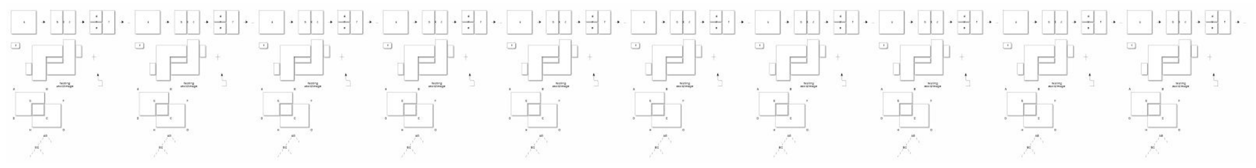
Figure 1: The (correct) rendered output of the Tiny Stress Test, the largest input we successfully process with DITAA

Unit Testing

The DITAA baseline was provided with very few existing unit tests. IDEA generated the following coverage report prior to our modifications and additions:

Package	Class, %	Method, %	Line, %
all classes	27.3% (12/ 44)	16.8% (107/ 637)	17.9% (777/ 4351)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
org.stathissideris.ascii2image.core	5.9% (1/ 17)	0.9% (1/ 113)	1.7% (13/ 781)
org.stathissideris.ascii2image.graphics	0% (0/ 12)	0% (0/ 225)	0% (0/ 1688)
org.stathissideris.ascii2image.test	100% (3/ 3)	100% (21/ 21)	100% (169/ 169)

org.stathissideris.ascii2image.text	66.7% (8/ 12)	30.6% (85/ 278)	34.7% (595/ 1713)

Additions

The pre-existing implementation has some unit tests for the TextGrid, CellSet, and GridPattern classes. We write additional unit tests for:

- DebugUtils
- Pair
- ConversionOptions
- DiagramShape

Of these, DiagramShape is the most significant and complex class, and receives the most detailed unit tests. DiagramShape's non-trivial public methods are all assigned a test.

Results

With our additions to the unit tests, the code coverage was improved greatly. The overall coverage was increased by 2.7 times when counted by class, and more than tripled when counting by method and line. We were able to achieve 75% coverage in the graphics package which previously had no unit test coverage.

The following coverage metrics were generated by IDEA with our added unit tests:

Package	**Class, %**	**Method, %**	**Line, %**
all classes	73.5% (36/ 49)	60.3% (396/ 657)	61.8% (2780/ 4499)

Coverage Breakdown

Package	**Class, %**	**Method, %**	**Line, %**
org.stathissideris.ascii2image.core	47.1% (8/ 17)	20.2% (23/ 114)	9.7% (78/ 806)
org.stathissideris.ascii2image.graphics	75% (9/ 12)	61.7% (140/ 227)	67.1% (1179/ 1756)
org.stathissideris.ascii2image.test	100% (8/ 8)	100% (38/ 38)	100% (221/ 221)

org.stathissideris.ascii2image.text	91.7% (11/ 12)	70.1% (195/ 278)	75.9% (1302/ 1716)

The low coverage for the core package is due in part to unused classes, such as JavadocTaglet and VisualTester, and partially-implemented functionality such as custom-shape config files, which is also unused.

Bug Discovery and Repair

Mixed Shape Bug(s)

During the Design phase, we discovered a possible bug in the Diagram constructor--the handling of mixed shapes may fail under certain circumstances. We introduce new test cases to confirm the existence of this bug and clarify its nature, and in three rounds of testing and code modification, correct the behavior of DITAA by re-designing the aforementioned section of the method.

Our original understanding of the bug is that mixed shapes are decomposed using one of two strategies in the original implementation. They are decomposed using the logic ‘subtract all closed shapes from mixed shapes’, whenever any closed shapes are present, and using an ‘advanced’ algorithm that relies on tracing when there are not. But that may not always be appropriate. We introduce test case 23, which confirms that the ‘advanced algorithm’ logic works under the example circumstance given by the author:

```
+---------+
|         |
|         |
|         |
|       --+----------------------
|         |
|         |
|         |
+---------+
```

And test case 24 confirms that, if a separate closed shape is introduced, the pre-existing implementation fails because the advanced algorithm is not used:

```
+---------+                                    +-+
|         |                             | |
|         |                             +-+
|         |
|       --+----------------------
|         |
|         |
|         |
+---------+
```

In this case, the larger shape in the diagram disappears entirely, because it is never successfully decomposed into a series of open and closed shapes.

We also develop test case 25, which shows the same error occurring even in one large connected component:

```
+---------+------+
|         |      |
|         |      |
|         |      |
|       --+------+---------------
|         |
|         |
|         |
+---------+-------------------------
```

And the slightly different test case 26 demonstrates that when the ‘advanced’ algorithm is triggered by ensuring that no simple closed shapes exist, the shape is handled correctly. This test case passes in the pre-existing implementation:

```
+---------+---+--+
|         |   |  |
|         |   |  |
|         |      |
|       --+------+---------------
|         |
|         |
|         |
+---------+-------------------------
```

Our first notion, then, is that the ‘advanced’ mixed-shape handler appears to work in all cases, and was added after the method of subtracting closed shapes was discovered not to work in the case where no closed shapes were present in the diagram. We believe we can simplify the code by simply removing the branch which subtracts closed shapes and always uses the advanced algorithm.

After making this change, we test the new version of the software. The new test cases that we have devised pass. However, we also run a regression test by running the older, existing test cases--and test case 3 (reproduced here) fails!

```
           +--+
 +-----+-->|  |
 |     |   +--+
 |     |
 |     |
+-+-+ +-+-+
|   | |   |
|   | |   |
+---+ +---+
```

Something about two closed shapes joined by a line appears to cause trouble! We introduce test cases 27, 28, and 29 to clarify this behavior:

```
           +--+
 +-----+
 |     |
 |     |
 |     |
+-+-+ +-+-+
| | | | | |
|   | |   |
+---+ +---+
```

```
           +--+
 +-----+
 |     |
 |     |
 |     |
+-+-+ +-+-+
|   | |   |
|   | |   |
+---+ +---+
```

```
+-------+----+---+
|       |    |   |
|       |        |
+-------+--------+
```

These test cases all fail in the implementation current at this stage. Test case 29 causes significant confusion, as in the implementation at this stage, multiple copies of shape sections are drawn! This will eventually turn out to be caused by a closely-related bug that we will have to fix in the course of repairing our original bug. But initially, we are unaware of this, and create test case 30 to help clarify:

```
+-------+----+---+
|            |   |
|                |
+-------+--------+
```

Test case 30 passes, as well it should. There is something problematic about the existence of extra closed shapes, it seems?

We also have to closely study the implementation of the ‘advanced’ method, CellSet.breakTrulyMixedBoundaries(). This is essentially a tracing method, which pulls out open shapes by starting to trace at ‘dead-ends’ and stopping when it reaches intersections. Test case 3, then, can be explained as a failure because there are multiple closed cells in a mixed shape (the one created by an outside-in fill in Step 2) but only one little dead-end, from which tracing stops before any significant part of the shape can be processed.

There is an immediate impulse to use the subtraction of closed shapes *followed* by the ‘advanced’ algorithm. Indeed, this would cause test case 3 to pass. But we should beware of ‘blindly’ making changes to ensure our existing test cases pass, without fully understanding why. Test cases may not--and most often do not--cover the entire possibility space of inputs to a program, and cannot substitute for careful reasoning.

Indeed, our understanding of the code indicates that this hotfix would not give correct behavior on test case 27 (above). There are no simple closed shapes to be subtracted there.

However, we reason, if the ‘advanced’ algorithm was run *first*, and the closed shapes found as a *result* were subtracted from the remaining cells...then the proper set of closed and open shapes could be found.

We want to do this without breaking cases like case 26 (above), where subtracting one closed shape could destroy another.

Based on this, we believe we can achieve fully-correct mixed shape processing with a modest re-design to this part of the algorithm. We will iterate over both steps of mixed-shape processing in a do-while loop--returning to the start of the loop as long as any mixed shapes remain that haven’t been broken down--and additionally, rather than subtracting all closed shapes willy-nilly, we will ‘subtract’ only closed shapes that are *subsets* of a larger mixed shape under consideration.

(This also allows us to remove the large copy-pasted block of ‘classify all shapes’ logic, so that it exists only in one place, making for cleaner code.)

We cannot think of any diagram that should be processed incorrectly by this design. Any set of cells must have either dead ends or simple closed shapes that can be identified via a flood fill. By continuing to pull out those shapes, we should always be able to break the diagram down into Closed and Open shapes.

Modifying the code as described results in most test cases passing--but, again, not all. Cases 27 and 29, for example, still fail. Upon seeing the output, we develop another intuition and create test case 31 to test it:

```
+-------+----+---+
|       |    |   |
|       |    |   |
|       |        |
|       |        |
+-------+--------+
```

Case 31 passes despite the failure of case 29. We finally realize what’s happening--there is a (separate) bug in the logic of breakTrulyMixedBoundaries() that destroys closed shapes by removing intersection characters--but only if the intersection is attached to a stub exactly 1 character long. By adding an extra check, this is prevented, and at last, all test cases pass!

The Wasteful Flood Fill Bug

In the process of refactoring Diagram by decomposing its large constructor into smaller methods, we began regression testing to ensure that behavior was unchanged and found that test case 1 (the very first!) appeared to result in hanging for an excessive period of time.

This behavior was tracked to findInteriorExteriorBoundaries(), where a comment by the original author indicated that there may be a previously-unfixed bug resulting in findBoundariesExpandFrom() returning empty lists when it was not expected to.

Further inspection revealed that this was, indeed, the source of the problem. The outer loops of findInteriorExteriorBoundaries() are intended to loop over every cell of an AbstractionGrid representing part of the input art, in order to try flood-filling a given shape from all possible locations. However, the bounds of this loop originally used this.width and this.height, which do NOT represent the height and width of that AbstractionGrid in ‘cells’, but the size of the entire Diagram in *pixels*. These loop bounds were vastly higher than they were supposed to be, and leading to huge numbers of unnecessary loop iterations. This is why variable names should give some indication of units!

We replaced these with the correct loop bounds, and the excessive execution time of findInteriorExteriorBoundaries() was no more.

Team Member Journals

Christopher Menart

11/27/20

- Pursuing suspected bug found when describing Diagram constructor
    - It seems like Mixed shapes wouldn’t be handled correctly in all cases
    - Writing test cases to handle this
    - End up going through multiple cycles of tests and code changes documented in Testing

11/28/20

- Need to begin testing! Start test document
- Only thing in it so far is documenting bugfix from the 27th

11/29/20

- Discovering/Fixing Wasteful Flood Fill Bug

11/30/20

- Writing JUnit tests
- Writing stress tests
- Formatting notes on DITAA classes for inclusion in Implementation document

12/1/20

- Finishing JUnit tests
- Starting Final Report Document
- Various report writing

12/2/20

- Re-running acceptance tests

Brad Schneider

11/29/20

- Considering how to test, generated some ideas, e.g. hashing two images to compare graphic output from DITAA

11/30/20

- Discussed possibility of using junit to implement automated regression/acceptance/integration test suite
- Implemented test suite on existing (30+) test files; pass/fail determined by whether the graphic output from test matches the ‘truth’ output in the tests directory
- Ran code coverage reports before/after the addition of all tests our team implemented

12/1/20

- Update test document (proofreading, formatting, etc.)
- Add test steps for new GUI feature
- Add code coverage reports to testing doc