Toronto Metropolitan University

Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	**Software Testing and QA**
Course Number:	**coe891**
Semester/Year (e.g.F2016)	**2023**

Instructor:	**Reza Samavi**

Assignment/Lab Number:	*Project*
Assignment/Lab Title:	*Project*

: Submission Date	**April, 6 2023**
Due Date:	**April, 6 2023**

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Faisal	**Mahir**	**500896206**	**4**	**M.F**
Ahmed	**Rehan**	**500896674**	**4**	**R.A**
Ossowski	**Oscar**	**500898545**	**4**	**O.O**
Systerov	**Ivan**	**500773127**	**4**	**I.S**

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: https://www.torontomu.ca/content/dam/senate/policies/pol60.pdf

Jediterm Terminal Widget Testing report

Introduction:

Test plan summary and main points + who is doing what

The open-source application that we chose for our project topic Jediterm is a java terminal application that can be integrated with various jetbrains-created IDEs such as IntelliJ IDEA and Pycharm. With this terminal application, Developers are able to test, compile and debug their code (which is written in a JetBrains IDE of course) with ease and simplicity. However, the main flaw of this application is that it can only be used with a JetBrains IDE. We are saying this with the assumption that since the compatibility level of this application with other IDEs is unknown, it is highly unlikely that this terminal application will work well with any other IDE application. In terms of its features, it's capable of doing everything that a standard linux terminal can do (similar to the MATE terminal on the lab computers) cut/copy/paste, and creating multiple terminal tabs to help keep oneself organized when testing and debugging their code.

Testing plan

The testing was done using 4 main testing abstractions of Input space partitioning, Control flow graphs, Data flow graphs and Logic coverage. The testing work split the features among the group members as shown below.

Xterm emulation – Oscar
Xterm 256 colours – Mahir
Terminal tabs – Mahir
Scrolling – Ivan
Copy/Paste – Ivan
Mouse support – Rehan
Terminal resizing – Client side only – Rehan

Each group member would test 10 methods from their topics and cover each of the 4 testing abstractions at least twice. The below results sections contain 1 example test method from each group member, each in a different testing abstraction.

Testing was done using junit unit testing library and each team member setup and worked on their own environment due to difference in hardware and software which made using a single shared testing environment infeasible.

The maturity level of the project was found to be a level 1 due to lack of proper comments for documentation, repetition of unnecessary code blocks, many unused function inputs and cases of improper documentation on how the code actually works.

Results:

Mahir – ISP, Ivan CFG, Oscar DFG, Rehan Logic coverage

Input Space Partitions

Class: ColorPaletteImpl

Method under test:getForegroundColorByIndex, getBackgroundColorByIndex

Characteristic	b1 (index < 0)	b2 (0 <= index <= 15)	b3 (index > 15)
Colors list index	-1	0, 1, 14, 15, 7	16

Figure 1: ISP choice table

Figures 2, 3, 4, 5, 6, and 7: ISP unit test screen shot

```
43    @Test
44    public void validColorValuesTest(){
45      Assume.assumeTrue( b: type==Type.VALID);
46      int[] actual = {c.getBackgroundByColorIndex(colorVal).getRed(),c.getBackgroundByColorIndex(colorVal).getGreen(),c.getBackgroundByColorIndex(colorVal).getBlue()};
47      assertArrayEquals(expectedList,actual);
48
49    }
50    @Test(expected = ArrayIndexOutOfBoundsException.class)
51    public void invalidColorValuesTest(){
52      Assume.assumeTrue( b: type==Type.INVALID);
53      int[] actual = {c.getBackgroundByColorIndex(colorVal).getRed(),c.getBackgroundByColorIndex(colorVal).getGreen(),c.getBackgroundByColorIndex(colorVal).getBlue()};
54      assertArrayEquals(expectedList,actual);
55
56    }
57  }
```

Figure 2: Background Color by Index Test Methods

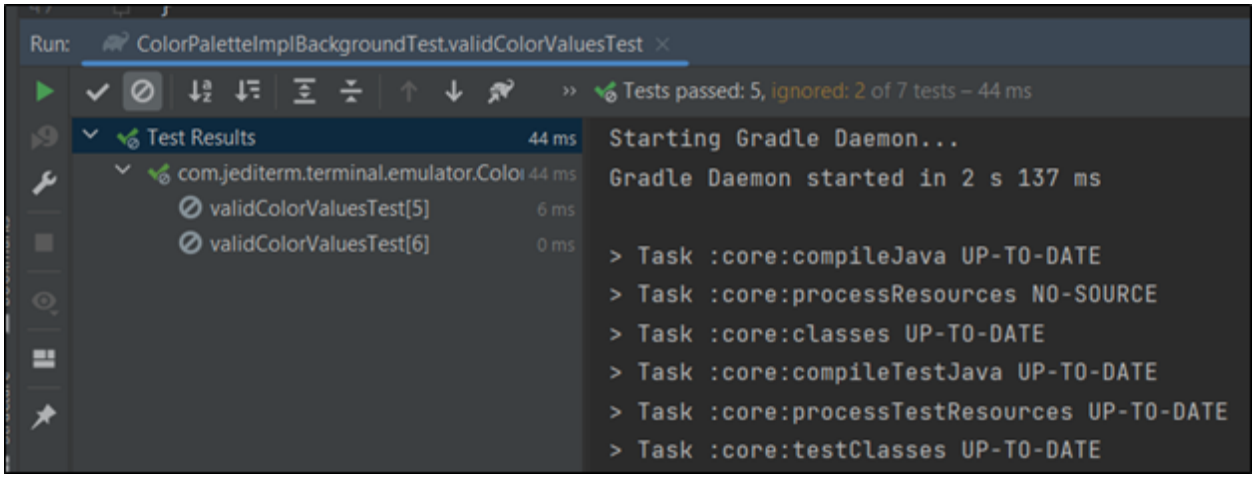

Figure 3: Background Color by Index Test Results For Valid Inputs

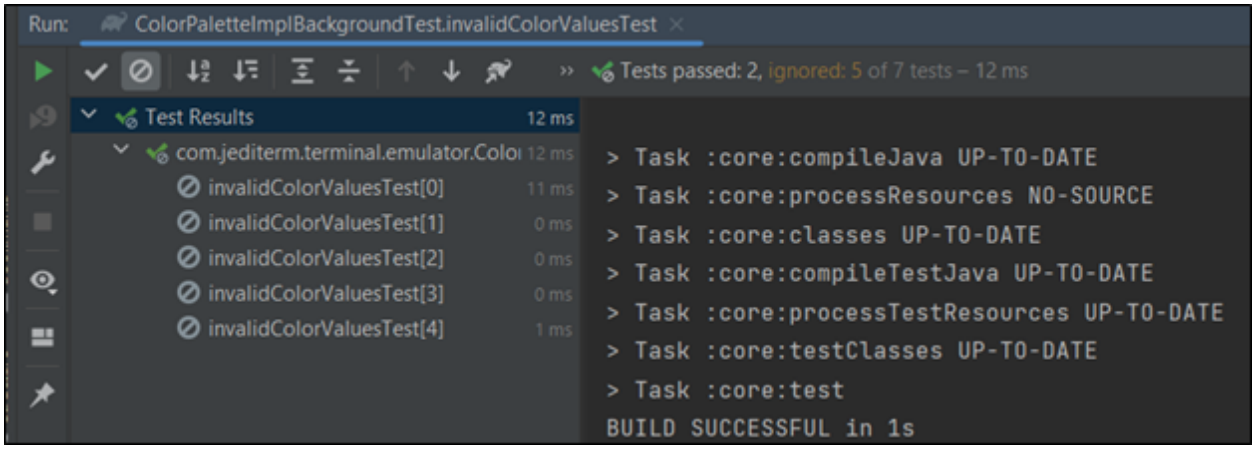

Figure 4: Background Color by Index Test Results For Invalid Inputs

```
@Test
public void validColorValuesTest(){
  Assume.assumeTrue( b: type== Type.VALID);
  int[] actual = {c.getForegroundByColorIndex(colorVal).getRed(),c.getForegroundByColorIndex(colorVal).getGreen(),c.getForegroundByColorIndex(colorVal).getBlue()};
  assertArrayEquals(expectedList,actual);

}
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void invalidColorValuesTest(){
  Assume.assumeTrue( b: type== Type.INVALID);
  int[] actual = {c.getForegroundByColorIndex(colorVal).getRed(),c.getForegroundByColorIndex(colorVal).getGreen(),c.getForegroundByColorIndex(colorVal).getBlue()};
  assertArrayEquals(expectedList,actual);

}
```

Figure 5: Foreground Color by Index Test Methods

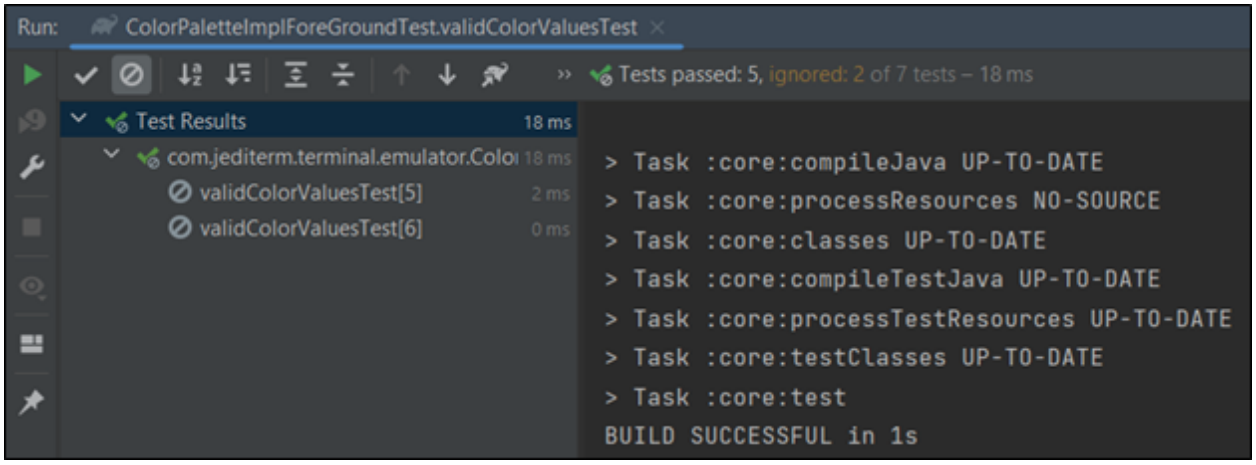

Figure 6: Foreground Color by Index Test Results for Valid Inputs

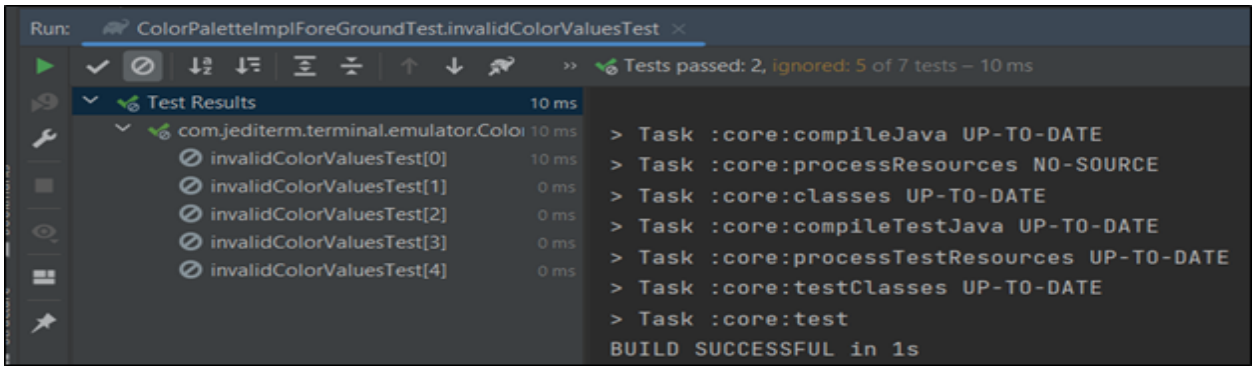

Figure 7: Foreground Color by Index Test Results for Invalid Inputs

The purpose of the Color Palette class is to assign specific color palettes for various available colors that can be applied to and used in the foreground and the background of the jediterm terminal application. The getBackgroundColorByIndex and getForegroundColorByIndex methods were both selected for input space partition testing is because we verify if the method works correctly by returning proper color objects for a given color palette that is being used by either the terminal foreground or the background and prevents incorrect colors or colors that are not a part of the palette to be returned. Furthermore, both methods also satisfy the RIPR as they are easily reachable, infectable, propagatable, and readable. This is thanks to the fact that the methods themselves are being created in a class that implements the colorPalette.java abstract class interface. Additionally, since it is also an implemented class that implements the colorPalette abstract, it can also invoke other methods that are used exclusively by the colorPalette abstract class as well. The ISP approach was the best approach to use for both of these methods because they help us accurately verify if the method is able to function correctly by being able handle valid color object inputs that belong as part of the main color palette as well as being able to handle invalid color objects that do not belong as part of the main color palette.

Control flow graph

Class: getContents

Method under test:

jediterm/ui/src/com/jediterm/terminal/DefaultTerminalCopyPasteHandler.java

Figure 8: CFG graph

```
public String getContents(boolean useSystemSelectionClipboardIfAvailable) {
   if (useSystemSelectionClipboardIfAvailable) {
     Clipboard systemSelectionClipboard = getSystemSelectionClipboard();
     if (systemSelectionClipboard != null) {
       return getClipboardContents(systemSelectionClipboard);
     }
   }
   return getSystemClipboardContents();
 }
```

Figure 9: CFG unit test

CFG:

Nodes	Edges:	Edge Pairs:	Prime Paths
[0]	[0,1]	1. [0,1,2]	1. [3,6,7,9,1,2,4,5]

[1]	[1,2]	2. [1,2,3]	2. [2,4,5,6,7,9,1,2]
[2]	[2,3]	3. [1,2,4]	3. [4,5,6,7,9,1,2,3]
[3]	[3,6]	4. [2,3,6]	4. [1,2,4,5,6,7,9,1]
[6]	[2,4]	5. [3,6,7]	5. [0,1,2,4,5,6,7,8]
[4]	[4,5]	6. [2,4,5]	6. [0,1,2,4,5,6,7,9]
[5]	[5,6]	7. [4,5,6]	7. [4,5,6,7,9,1,2,4]
[7]	[6,7]	8. [5,6,7]	8. [9,1,2,4,5,6,7,8]
[8]	[7,8]	9. [6,7,8]	9. [9,1,2,4,5,6,7,9]
[9]	[7,9]	10. [6,7,9]	10. [7,9,1,2,4,5,6,7]
	[9,1]	11. [7,9,1]	11. [5,6,7,9,1,2,4,5]
		12. [9,1,2]	12. [6,7,9,1,2,4,5,6]
			13. [2,3,6,7,9,1,2]
			14. [3,6,7,9,1,2,3]
			15. [0,1,2,3,6,7,9]
			16. [0,1,2,3,6,7,8]

17. [1,2,3,6,7,9,1]
18. [9,1,2,3,6,7,8]
19. [9,1,2,3,6,7,9]
20. [7,9,1,2,3,6,7]
21. [6,7,9,1,2,3,6]

PPC:

Test Paths	TR
[0,1,2,3,6,7,9,1,2,4,5,6,7,8]	[3,6,7,9,1,2,4,5], [9,1,2,4,5,6,7,8], [7,9,1,2,4,5,6,7], [6,7,9,1,2,4,5,6], [2,3,6,7,9,1,2], [0,1,2,3,6,7,9], [1,2,3,6,7,9,1]
[0,1,2,4,5,6,7,9,1,2,3,6,7,8]	[2,4,5,6,7,9,1,2], [4,5,6,7,9,1,2,3], [1,2,4,5,6,7,9,1], [0,1,2,4,5,6,7,9], [9,1,2,3,6,7,8], [7,9,1,2,3,6,7], [6,7,9,1,2,3,6]
[0,1,2,4,5,6,7,8]	[0,1,2,4,5,6,7,8]
[0,1,2,4,5,6,7,9,1,2,4,5,6,7,8]	[2,4,5,6,7,9,1,2], [1,2,4,5,6,7,9,1], [0,1,2,4,5,6,7,9], [4,5,6,7,9,1,2,4], [9,1,2,4,5,6,7,8], [7,9,1,2,4,5,6,7], [5,6,7,9,1,2,4,5], [6,7,9,1,2,4,5,6]
[0,1,2,3,6,7,9,1,2,4,5,6,7,9,1,2,3,6,7,8]	[3,6,7,9,1,2,4,5], [2,4,5,6,7,9,1,2], [4,5,6,7,9,1,2,3], [1,2,4,5,6,7,9,1], [9,1,2,4,5,6,7,9], [7,9,1,2,4,5,6,7], [6,7,9,1,2,4,5,6], [2,3,6,7,9,1,2], [0,1,2,3,6,7,9], [1,2,3,6,7,9,1], [9,1,2,3,6,7,8], [7,9,1,2,3,6,7], [6,7,9,1,2,3,6]
[0,1,2,3,6,7,8]	[0,1,2,3,6,7,8]

[0,1,2,3,6,7,9,1,2,3,6,7,9,1,2,3,6,7,8]	[2,3,6,7,9,1,2], [3,6,7,9,1,2,3], [0,1,2,3,6,7,9], [1,2,3,6,7,9,1], [9,1,2,3,6,7,8], [9,1,2,3,6,7,9], [7,9,1,2,3,6,7], [6,7,9,1,2,3,6]

EPC:

Test Paths	TR
[0,1,2,4,5,6,7,8]	[0,1,2], [1,2,4], [2,4,5], [4,5,6], [5,6,7], [6,7,8]
[0,1,2,3,6,7,9,1,2,3,6,7,8]	[0,1,2], [1,2,3], [2,3,6], [3,6,7], [6,7,8], [6,7,9], [7,9,1], [9,1,2]

EC/NC:
[0,1,2,4,5,6,7,8]
[0,1,2,3,6,7,9,1,2,3,6,7,8]

```
import org.junit.Test;
import static org.junit.Assert.*;

public class ClipboardTest {
  @Test
  public void testGetContents() {
    DefaultTerminalCopyPasteHandler handler = new DefaultTerminalCopyPasteHandler()
    String expected = "test string";
    handler.setSystemClipboardContents(expected);
    String actual = handler.getContents( useSystemSelectionClipboardIfAvailable: false);
    assertEquals(expected, actual);
  }
```

Data Flow Graph
Class: blob/master/core/src/com/jediterm/terminal/emulator/ControlSequence.java
Method under test: appendToBuffer

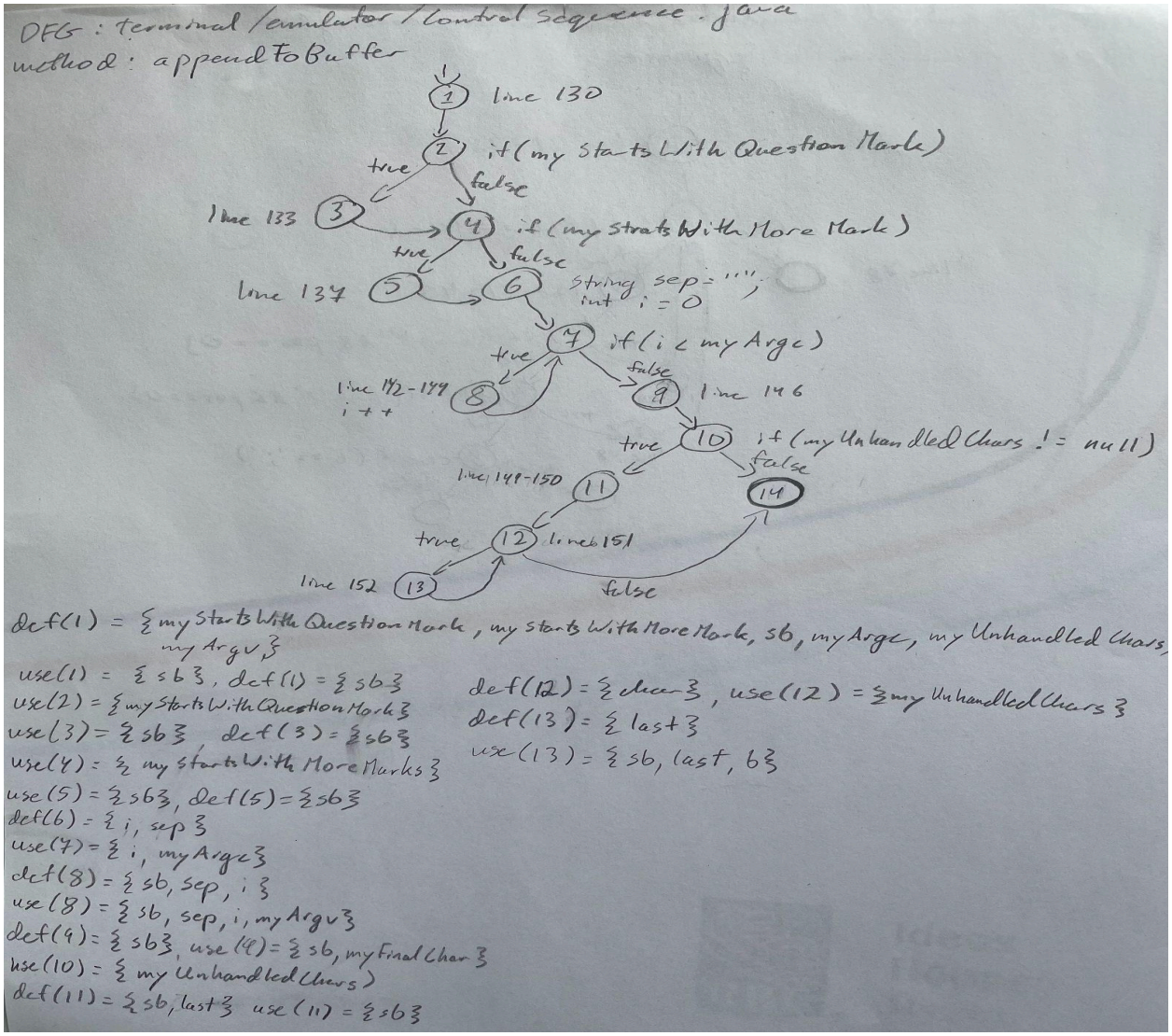

Figure 10: DFG graph

Variable name mapping (for my own sanity when typing it out):

myStartsWithQuestionMark = q
myStartsWithMoreMark = m
myArgc = c
myArgv = v
myUnhandledChars = u
sb = s
sep = p
i = i
myFinalChar = f
last = l
b = b

Du pairs:

q = [[1,2]]
m = [[1,4]]
f = [[1,9]]
v = [[1,8]]
c = [[1,7]]
u = [[1,10], [1,12]]
s = [[1,1], [1,3], [1,5], [1,8], [1,9], [1,11], [1,13], [3,1], [3,3], [3,5], [3,8], [3,9], [3,11], [3,13], [5,1], [5,3], [5,5], [5,8], [5,9], [5,11], [5,13], [8,1], [8,3], [8,5], [8,8], [8,9], [8,11], [8,13], [9,1], [9,3], [9,5], [9,8], [9,9], [9,11], [9,13], [11,1], [11,3], [11,5], [11,8], [11,9], [11,11], [11,13]]
p = [[6,8], [8,8]]
i = [[6,7], [6,8], [8,7], [8,8]]
b = [[12,13]]
l = [[11,13], [13,13]]

Du paths:
q = [[1,2]]
m = [[1,2,4], [1,2,3,4]]
f = [[1,2,4,6,7,9], [1,2,3,4,6,7,9], [1,2,4,5,6,7,9], [1,2,3,4,5,6,7,9]]
v = [[1,2,4,6,7,8], [1,2,3,4,6,7,8], [1,2,4,5,6,7,8], [1,2,3,4,5,6,7,8]]
c = [[1,2,4,6,7], [1,2,3,4,6,7], [1,2,4,5,6,7], [1,2,3,4,5,6,7]]
u = [[1,2,4,6,7,9,10], [1,2,3,4,6,7,9,10], [1,2,4,5,6,7,9,10], [1,2,3,4,5,6,7,9,10], [1,2,4,6,7,9,10,11,12], [1,2,3,4,6,7,9,10,11,12], [1,2,4,5,6,7,9,10,11,12], [1,2,3,4,5,6,7,9,10,11,12]]
s = [[1,2,3], [1,2,4,5], [1,2,4,6,7,9], [1,2,4,6,7,8], [3,4,5], [3,4,6,7,9], [3,4,6,7,8], [5,6,7,9], [5,6,7,8], [8,7,9], [8,7,8], [9,10,11], [11,12,13]]
p = [[6,7,8], [8,7,8]]
i = [[6,7], [6,7,8], [8,7], [8,7,8]]
b = [[12,13]]
l = [[11,12,13], [13,12,13]]
Infeasible paths = [3,4,5], [11,12,14]

All def coverage:
Ts = [[1,2,4,6,7,9,10,14], [1,2,4,6,7,8,7,9,10,14], [1,2,3,4,6,7,9,10,14], [1,2,4,5,6,7,9,10,14], [1,2,4,6,7,9,10,11,12,13,12,14]]]
Inputs = [[t], [1,t], [?,t], [>,t], [:,t]]

All use coverage:
Ts = [[1,2,4,6,7,9,10,14], [1,2,4,6,7,8,7,9,10,14], [1,2,3,4,6,7,9,10,14], [1,2,4,5,6,7,9,10,14], [1,2,4,6,7,8,7,8,7,9,10,14], [1,2,4,6,7,9,10,11,12,13,12,14], [1,2,4,6,7,9,10,11,12,13,12,13,12,14], [1,2,3,4,6,7,8,7,9,10,14], [1,2,4,5,6,7,8,7,9,10,14]]

Inputs = [[t], [1,t], [?,t], [>,t], [:,t], [1,';',1,';',t], [:,:,t], [>,1,t], [?,1,t]]

All du-path coverage:
Ts = [[1,2,4,6,7,9,10,14], [1,2,4,6,7,8,7,9,10,14], [1,2,3,4,6,7,9,10,14], [1,2,4,5,6,7,9,10,14], [1,2,4,6,7,8,7,8,7,9,10,14], [1,2,4,6,7,9,10,11,12,13,12,14], [1,2,4,6,7,9,10,11,12,13,12,13,12,14], [1,2,3,4,6,7,8,7,9,10,14], [1,2,4,5,6,7,8,7,9,10,14], [1,2,3,4,6,7,9,10,11,12,13,12,14], [1,2,3,4,5,6,7,9,10,11,12,13,12,14]]
Inputs = [[t], [1,t], [?,t], [>,t], [:,t], [1,;,1,;,t], [:,:,t], [>,1,t], [?,1,t], [>,:,t],[?,:,t]]

Figure X Below shows the Junit testing implementation for all def coverage over the method using the 5 given test inputs.

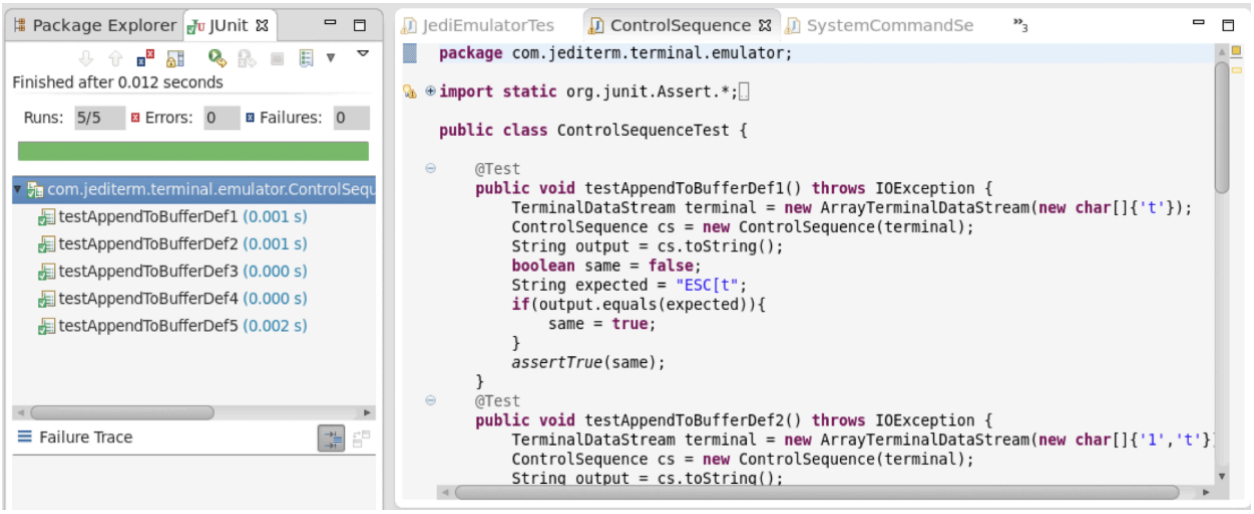

Figure 11: DFG unit test screen shot

The controlSequence class' purpose is to process user input in the terminal application and interpret the input into the proper command structure to allow execution of the user desired functionality. The appendToBuffer method was chosen for test because it fits the entire RIPR model well, being easily reachable, infectible, propagatable, and readable. It deals with all the elements of the controlSequence object, thus testing the method allows for guaranteeing that the input commands are being read properly. The function itself is also used in the toString operation of the object which is integral to the functionality of the application. The DFG format was used because its structure was the best way to model the large amount of modifications done to the StringBuilder object throughout the class, which no other testing approach could properly model.

Logic coverage
Class:blob/master/JediTerm/src/main/java/com/intellij/openapi/ui/ClickListener.java
Method under test:mouseReleased

Logic Coverage was used for the mouseReleased method as it is a vital feature that requires each testing path to be functioning correctly. More specifically, the mouseReleased method allows the JediTerm software to be able to distinguish whether the mouse has a button clicked and without this functionality the software would stop listening to mouse input. Logic Coverage was chosen because the logical statement to determine whether the mouse has clicked a certain part of the screen included multiple boolean operators such as || and && which were used to determine the predicates for this formula efficiently.

A=e.isConsumed()
B=clickedAt == null
C= e.isPopupTrigger()
D=!e.getComponent().contains(e.getPoint())
E=(isWithinEps(releasedAt, clickedAt)
F=onClick(e, clickCount))

P1=A||B ||C||D
P2=E&&F

The reachability based on this:

Line 69-> False
Line 73-> True

The determination predicates based on this include:

P1: A||B ||C||D
A=True-> (True||B||C||D)=True
A=False->(False||B||C||D)=B|C|D
True xor (B||C||D)= !B&&!C&&!D
So A is the determining clause when B, C and D are false

P1: A||B ||C||D
B=True-> (A||True||C||D)=True
B=False->(A||False||C||D)=A||C||D
True xor (A||C||D)= !A&&!C&&!D
So B is the determining clause when A, C and D are false

P1: A||B ||C||D
C=True-> (A||B||True||D)=True
C=False->(A||B||False||D)=A||B||D
True xor (A||B||D)= !A&&!B&&!D
So C is the determining clause when A, B and D are false

P1: A||B ||C||D
D=True-> (A||B||C||True)=True
D=False->(A||B||C||False)=A||B||C
True xor (A||B||C)= !A&&!B&&!C
So D is the determining clause when A, B and C are false

P2:E&&F
E=True->True&&F=True
E=False->False&&F=False
True xor False=true
So E is the determining clause when F is true

P2:E&&F
F=True->E&&True=True
F=False->E&&False=False
True xor False=true
So F is the determining clause when E is true

GACC:
TR=[A=T], [A=F], [B=T], [B=F], [C=T],[C=F],[D=T],[D=F][E=T,F=T],[E=T,F=F],[E=F, F=T]
TS=[A=e.isConsumed()=true], [A=e.Consumed=false], [clickedAt==null=true],
][clickedAt==null=false] [C=e.isPopupTrigger=true],
[C=e.isPopupTrigger=false],[!e.getComponent().contains(e.getPoint())=true]
[!e.getComponent().contains(e.getPoint())=false],[E=(isWithinEps(releasedAt, clickedAt)=true,
F=F=onClick(e, clickCount)=true)].[E=(isWithinEps(releasedAt, clickedAt)=true, F=F=onClick(e,
clickCount)=false)], [E=(isWithinEps(releasedAt, clickedAt)=false, F=F=onClick(e,
clickCount)=true)]

CACC:
TR=[A=T], [A=F], [B=T], [B=F], [C=T],[C=F],[D=T],[D=F][E=T,F=T],[E=T,F=F],[E=F, F=T]
TS=[A=e.isConsumed()=true], [A=e.Consumed=false], [clickedAt==null=true],
][clickedAt==null=false] [C=e.isPopupTrigger=true],
[C=e.isPopupTrigger=false],[!e.getComponent().contains(e.getPoint())=true]
[!e.getComponent().contains(e.getPoint())=false],[E=(isWithinEps(releasedAt, clickedAt)=true,
F=F=onClick(e, clickCount)=true)].[E=(isWithinEps(releasedAt, clickedAt)=true, F=F=onClick(e,
clickCount)=false)], [E=(isWithinEps(releasedAt, clickedAt)=false, F=F=onClick(e,
clickCount)=true)]

RACC:TR=[A=T], [A=F], [B=T], [B=F], [C=T],[C=F],[D=T],[D=F][E=T,F=T],[E=T,F=F],[E=F, F=T]
TS=[A=e.isConsumed()=true], [A=e.Consumed=false], [clickedAt==null=true],
][clickedAt==null=false] [C=e.isPopupTrigger=true],
[C=e.isPopupTrigger=false],[!e.getComponent().contains(e.getPoint())=true]

[!e.getComponent().contains(e.getPoint())=false],[E=(isWithinEps(releasedAt, clickedAt)=true, F=F=onClick(e, clickCount)=true)].[E=(isWithinEps(releasedAt, clickedAt)=true, F=F=onClick(e, clickCount)=false)], [E=(isWithinEps(releasedAt, clickedAt)=false, F=F=onClick(e, clickCount)=true)]

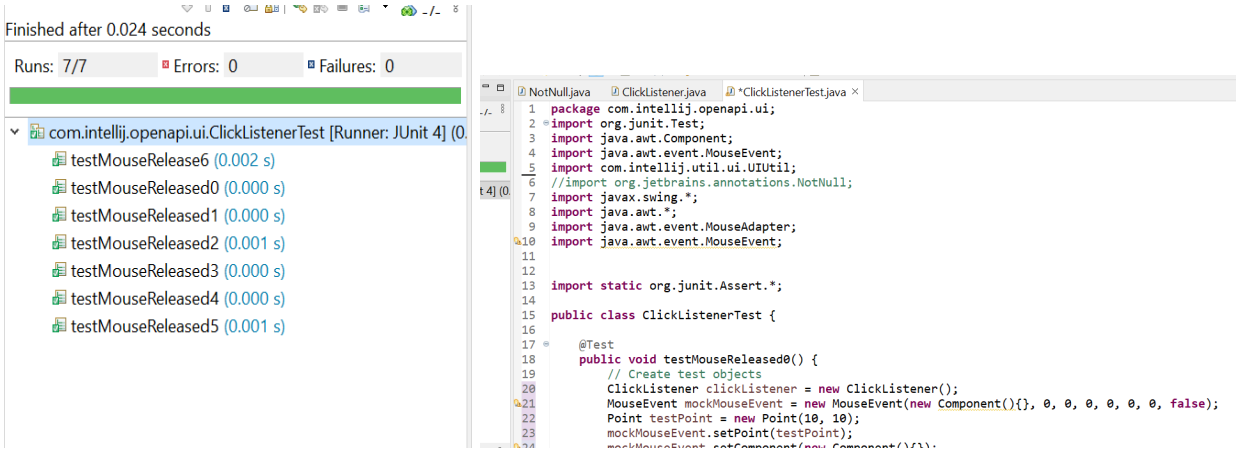

Figure 12: Logic coverage unit test screen shot

Total results

In general the methods were tested for correctness using the unit testing approach on individual methods. The aim was to reach at least 90% correctness for critical features and 75% correctness for all other tested features. With the limited testing done in this all the methods under test passed this condition and were deemed to work correctly. This however does not exclude the existence of bugs in the project, the most relevant of which are mentioned below in the following section

Bugs found

Initially we assumed that the amount of bugs present in an industry level project used in many ide's would be fairly stable and not contain many bugs since they would likely be reported by users. After testing and analyzing the more complex parts of the code though, we found that

many parts of the code are either not reachable, improperly documented, accept unused parameters as function inputs, containing mutually exclusive if statements that are both reachable or contain unnecessary repetition of code segments.

All the found bugs are mentioned in the raw analysis document which contains all of the testing work but some notable examples include: processColorQuery node 9 being unreachable due to never accepting a ps with a value different from 10 or 11 and edge 10-15 being unreachable due to throwing a null pointer exception before ever getting color to be null; or getMapping accepting a graphicSet gr and comment documentation claiming that it is used in getting a mapping but never actually being used.

Issues

The main issue faced in the testing endeavor was environment setup and getting it to work properly with the chosen project. Since each team member had a drastically different working setup, with some even using a different OS entirely, there were difficulties in setting up a single common testing environment. In the end it was impossible and instead each team member set up whatever they were most comfortable using and installed the relevant junit resting plugin.

Another issue was that the project repository was created using a different java version than the java version used by junit in class. This caused a lot of headaches in getting the two to work properly together with some classes ending up untestable due to errors.

A major issue faced in creating unit tests for this project was the lack of comments throughout the code. This made it very difficult to properly trace the requirements and figure out the expected ranges for certain inputs, and instead forced us to trace the code back in order to see how the functions are normally called, sometimes many classes deep, to see what the input format should be. Another test creation issue was that many methods used the default access modifier on their constructor which can only be called from the same package as the class itself, forcing the testing classes to share the package instead of using the standard of a separate test package for all tests.

One of the smaller issues faced was that many methods under test were private or protected access restricted, meaning that the only way to test them was by calling other methods first, and preparing our inputs to not only infect the method but also to reach it. Due to the complex nature of the project itself, oftentimes a large amount of the time spent writing tests was just tracing all the propagation requirements and making sure that the test was actually feasible in the context of the project itself.

What we learned/Reflections

Some of the main takeaways from this testing activity was how to better trace requirements throughout many classes to ensure test propagation. Since the project had limited comments for documentation which is likely very similar to what we may experience in future work, tracing the requirements and classes helped us learn how to better understand code and its execution. This is helpful not only in future testing endeavors but also in any future cooperative development task as it will help us better understand colleague/own code even when lacking documentation. The complexity of this project allowed us to better build up our code reading and traceability skills.

Furthermore this project allowed us the opportunity to get hands-on experience with actually building test cases and inputs based off the testing paths which we previously only studied from a theoretical angle or with simple test methods. This gave us much needed practical experience and showed just how many infeasible paths that are invisible when only looking at the single method alone and not in the context of the entire project.

Furthermore this project helped us develop intuition on how to decide which testing abstraction to use on each method to best cover it. In class we already covered basic ideas like to avoid graph coverage in cases with recursion or that logic coverage can only be used in cases with actual logical expressions. Now however we learned about other restrictions based on reachability or the inner methods called using function calls which may add extra restrictions not present in our mathematical analysis.

Finally we learned first hand the effect of loops and just how much damage they actually cause in any graph based coverage approach. In class most loops were made to be simple to model, however in practice even the simplest loops can greatly balloon the amount of test cases needed for prime path coverage even if the method has a fairly simple concept otherwise.

This testing exercise showed how even the simplest methods are difficult to test fully and gave us a greater appreciation for the hard work done by testing engineers and the patience they have for the suboptimal code given to them to test.

Thank you for the semester!