

Hasan Al-Habbobi 216315428
Sharujan Rajakumar 216376410
Alain Ballen 216341703
Ahmed Hagi 215043896

TESTING DOCUMENT - GROUP 5

Purpose:

The purpose of this document is to describe the test cases that have been implemented for this application. Additionally this document discusses how these test cases were derived and why they are sufficient which will incorporate discussion of test coverage.

Table of Contents

1.0 = Test Cases

2.0 = Test Cases Implemented

3.0 = Why these Test Cases are sufficient

3.1 - All Aspects are Tested

3.2 - Tests are High Quality

3.3 - Sufficient Statement Test Coverage Metrics

1.0 = Test Cases

The following test cases were implemented (organized by Container, Text and Project categories) along with their descriptions and explanations of how each one was derived.

Unit testing was used to test the essential features of the application which was done using JUnit. GUI testing was done by manually to observe how the application reacts to user input.

#	Test Case	Description	How Test Case was Derived
	CONTAINER		
1	Container Size Test	Tests that both containers/circles increase/decrease in size as the size slider is moved to the left/right respectively. Also tests that when they increase/decrease that they are both treated as one object so that the sizes/space	The Application should allow the ability for users to increase/decrease the size of the containers as they see fit and have it work as intended.

		for each container and the intersection space are consistent ratios no matter the size chosen.	
2	Container Color Test	<p>Tests that each container color can be changed to a random one of the color picker menu options.</p> <p>Also tests that there is enough transparency for the intersection section to be clearly visible.</p>	The Application ensures that users have the ability to change the colors of the containers and have transparency in the colors to allow the intersection section to be clearly visible.
	TEXT		
	<i>Adding Text</i>		
3	Add Text Test	Tests if a new text field can be created and check if the text of that text field is "Empty Value".	Users should be able to add text items into the Venn diagram to be used to compare similarities and differences between other text items.
4	Add Multiple Text Test	Tests if more than one text element can be created.	An application that only lets the user add one text element would not be useful. The test case was derived by the users need to be able to add multiple text elements to make a Venn diagram.
5	Drag Text Test	Tests if a text element can be dragged to any position in the Project Panel.	The test case was derived by the users need to be able to move text wherever they wish to position it in the diagram
	<i>Editing Text</i>		
6	Edit Text Word Test	Tests if a text element's word can be changed from its default.	The test case was derived by the users need to edit text to what they want it to be

			(so that they can represent different ideas/elements, etc)
7	Edit Text Size Test	Test if a text element's size can be changed.	The test case was derived by the users need to be able to fit text elements in the containers and or suit the users style preferences.
8	Edit Text Font Test	Tests if a Text element's font can be changed.	The test case was derived by the users need to be able to change the styling of text elements to suit their preference/theme/style
9	Edit Text Color Test	Tests if a Text element's color can be changed.	The test case was derived by the same reason for "Edit Text Font Test"
	<i>Deleting Text</i>		
10	Delete Text Test	Tests if a text element can be deleted.	The test case was derived by the users need to remove text elements they no longer need/want
11	Clear All (Delete All Text) Test	Tests if all text elements can be deleted at once after "Clear All" and the subsequent "yes" button are clicked	The test case was derived for the user's convenience to get rid of all text elements in two button presses as opposed to deleting each text element one by one.
	PROJECT		
12	Create New Project Test	Tests if New Project can be created (which tests that a project can be named and saved in a file location)	The test case was derived by the users need to create new projects independent of one another
13	Save Test	Tests if a project can be saved such that when it is	The test case was derived by the users

		opened all text elements and their properties.	need to save the project such that they can close the application and work on it later
14	Save As Test	Tests if project can be saved under a new name and a new location	The test cases expands of the derivation of "Save" to incorporate the users need to change the project title or save location
15	Download As A) PNG B) PDF	Tests if Venn Diagram can be downloaded as a PNG or PDF	The test case was derived by the users need to use the Venn Diagram they developed outside of this application/program (whether its sharing it a friend or using it for a presentation, etc)
16	Open Project Tests A) When first opening program B) In the middle of existing project	Tests if the previous project can be responded to (whether it then the application is first opened or in the middle of working on another project in the application in which case it should prompt the user to save their work before doing so.	<p>The test case was derived by the users need to open a project they have saved.</p> <p>This fulfills the users need of being able to close the application and working on a project (that was previously being worked on) later.</p>

2.0 = Test Cases Implemented

We implemented Test Cases 1, 2, and 3 shown in Section 1.0. For the final project we will implement the remaining Test Cases.

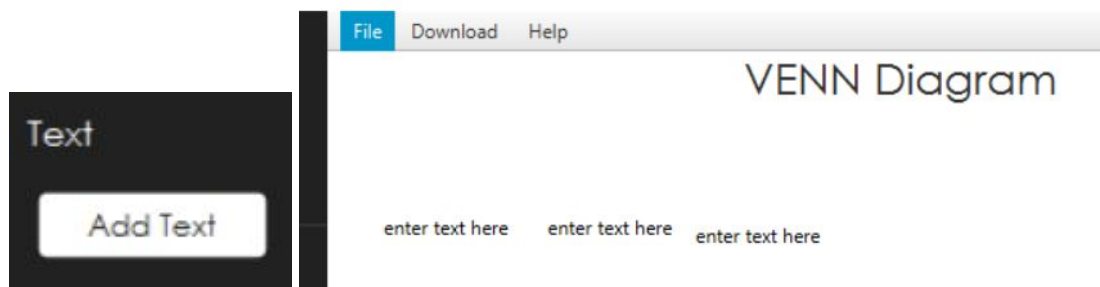
NOTE:

By implementation we mean that we made/implemented junit tests. The other Test Cases were tested for manually (i.e. we checked that each of the test cases in 1.0 worked and satisfied the description of the expected test results).

1 - Manual Testing:

Test Cases 4-16 (as shown in section 1.0) were tested manually. What this means is that we would run the executable jar file version of the program and do the test cases outlined in section 1.0, seeing if the features work. Junit test cases will be implemented for all features in the future.

Let's take Test Case 4 (Adding Multiple Text Elements) as an example to manually test if this case was satisfied we ran the program then clicked the Add Text button, dragged the text to the side and repeated the process. Since we saw that there were multiple text elements added this meant that the expected test results for that case were satisfied (as shown below).



Another example is how we manually tested Test Case 5 (Drag Text Test) by running the program, adding a text element followed by left mouse clicking on that newly created text element while holding down the left mouse key moving the mouse around and letting go of the button. We noticed that the text would move in the direction of the mouse while the left button was clicked. In other words the text could be dragged successfully which means that our expected test results were satisfied.

This testing of test cases 3, 4 and 5 is demonstrated in the following video <https://drive.google.com/file/d/1LRT2QWQweWqm0JNIXb0vXcgzN7HTjayM/view?usp=sharing> (this video is also linked in Section 3.3 of the User Manual).

2 - Junit Testing Implementation:

Test Cases 1, 2, and 3 (as shown in section 1.0) were tested via Junit Tests in the VennTester.java file.

The Junit test cases that we implemented were: getSizeTest(), getColorTest(), getEmptyTextTest() and getTextTest().

For Test Case #1 (Container Size Test)

getSizeTest() - we used Junit tests by setting the radius of the circle and getting the radius of the circle. For this, we needed to implement methods in mainFXMLController such as setRadius() and getRadius(). We used assertEquals to test if the radius set was equivalent to the radius that we got.

For Test Case #2 (Checks for the Container Color Test)

getColorTest() - we used Junit tests to check if the Color RED that we filled with the Circle was satisfied. For the Junit test cases to work, we implemented methods in mainFXMLController such as getColor() and setColor().

For Test Case #3 (Checks for empty text)

getEmptyTextTest() - we used Junit tests to see if the text inputted was equal to null using assertEquals, if so then it's an empty text.

For Test Case #3 (Checks if text is added)

getTextTest() - we used Junit tests by adding a new text and setting the labelCount to be 1 since a new entry has been added. For this, we needed to implement methods in mainFXMLController such as addText() and EditableLabel getText(). Then, we check using assertEquals if the text is equal to the entry added at labelCount (1) and if it is then it returns true (test case is satisfied).

3.0 = Why these Test Cases are sufficient

3.1 - All Aspects of User Interaction are Tested (Quantity)

There are enough test cases to cover all major aspects of the program. Containers (their color and size), Text (their font, color, size) as well as their addition/editing/deleting. Followed by Project/System features like Save, Save As, Open and Create New. In that regard each and every interaction the user can do with the program is tested. Thus all aspects of user Interaction are tested.







3.2 - Tests are High Quality

In addition to there being enough tests to cover all aspects. Each test itself is high quality so that it covers the sub-aspects of each aspect. For example the Editing Text Aspect is made up of 4 sub-categories/sub-aspects (Word, Size, Font and Color); each with their own text. It is also worthy to note that each of these suspect tests have their own sub aspects/details which are tested for. For example Text Color tests for any possible text color at random so that all possible color options (which are sub-aspects of the color aspect category) are tested.

In other words Edit Text Color doesn't just test the color blue, it will test a random color to simulate a user's choice which is unpredictable, ambiguous and random to the program in that respect. This concept applies to each of the Tests outlined in section 1.0 (e.g. Edit Text Font tests a random font, Edit Text Size tests a random size, etc).

The test cases ensures that if bugs do occur in the application that the user is still able to create a very professional Venn Diagrams and developers are able to focus on fixing bugs and improving less essential features of the application to enhance the user experience.

3.3 - Sufficient Statement Code Coverage Metrics

▼ venn.diagram		58.4 %
▶ J mainFXMLController.java		55.7 %
▶ J EditableLabel.java		71.8 %
▶ J VennDiagramTester.java		0.0 %
▶ J SelectionHandler.java		56.8 %
▶ J VennDiagram.java		100.0 %

At first glance the statement code coverage (as seen above) is below what is considered to be good coverage. However these are due to missing features that we still have to fully implement (such as “download to PDF”). In that regard the statement code coverage is sufficient for the features that have been implemented. The Coverage percentage shown above doesn’t omit the features that have yet to be implemented which misleadingly makes it seem insufficient. Furthermore, the VennDiagramTester is not a part of the Java GUI application, which is why it is at zero. Finally, there are a lot of small bugs in the program that need to be refined for the next version (The Final Submission). These missing features and bugs are discussed in Section 2.3 of the User Manual. Adding features like the text focuser feature for instance will allow users to efficiently edit each text element’s properties independently from one another. Thus, adding these features will allow for higher coverage metrics in the final version of the application. In that respect, despite the bugs and missing features, all the important features that are required to have a customizable Venn Diagrams are functional. The application statement coverage is therefore sufficient