SE 3XA3: Test Plan
Title of Project

Team #, Team Name
Student 1 name and macid
Student 2 name and macid
Student 3 name and macid

October 30, 2016

Contents

1 General Information **1**
- 1.1 Purpose . . . 1
- 1.2 Scope . . . 1
- 1.3 Acronyms, Abbreviations, and Symbols . . . 1
- 1.4 Overview of Document . . . 2

2 Plan **2**
- 2.1 Software Description . . . 2
- 2.2 Test Team . . . 2
- 2.3 Automated Testing Approach . . . 2
- 2.4 Testing Tools . . . 2
- 2.5 Testing Schedule . . . 2

3 System Test Description **2**
- 3.1 Tests for Functional Requirements . . . 2
  - 3.1.1 Area of Testing 1 . . . 2
  - 3.1.2 Area of Testing 2 . . . 4
  - 3.1.3 Area of Testing 3 . . . 5
  - 3.1.4 Area of Testing 4 . . . 6
  - 3.1.5 Area of Testing 5 . . . 8
  - 3.1.6 Area of Testing 6 . . . 8
- 3.2 Tests for Nonfunctional Requirements . . . 9
  - 3.2.1 Area of Testing2 . . . 9

4 Tests for Proof of Concept **10**
- 4.1 Area of Testing1 . . . 10
- 4.2 Area of Testing2 . . . 10

5 Comparison to Existing Implementation **10**

6 Unit Testing Plan **10**
- 6.1 Unit testing of internal functions . . . 10
- 6.2 Unit testing of output files . . . 10

7 Appendix **11**
- 7.1 Symbolic Parameters . . . 11
- 7.2 Usability Survey Questions? . . . 11

List of Tables

1 Revision History ii
2 Table of Abbreviations 1
3 Table of Definitions 1

List of Figures

Table 1: **Revision History**

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

This document ...

1 General Information

1.1 Purpose

1.2 Scope

1.3 Acronyms, Abbreviations, and Symbols

Table 2: **Table of Abbreviations**

Abbreviation	**Definition**
Abbreviation1	Definition1
Abbreviation2	Definition2

Table 3: **Table of Definitions**

Term	**Definition**
Term1	Definition1
Term2	Definition2

1.4 Overview of Document

2 Plan

2.1 Software Description

2.2 Test Team

2.3 Automated Testing Approach

2.4 Testing Tools

2.5 Testing Schedule

See Gantt Chart at the following url ...

3 System Test Description

It is important to note that because of the lack of resources (mainly time), the testing will not be able to cover a wide array of test cases. At times, single positive and negative test cases will be enough to ensure a program works as expected.

3.1 Tests for Functional Requirements

3.1.1 Area of Testing 1

Requirement #1: The software shall read data given to it.
Requirements #3: The software will plot all the data points.

1. **Test ID #1.1**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
   Input: The list object: [(1, 1), (2, 2), (3, 3), (4, 4)]
   Output: A window depicting a graph with plotted points at (1, 1), (2, 2), (3, 3), and (4, 4).
   How test will be performed:
   - The user will instantiate the graph object.

- The list object will be passed into the plotting method in the API.
- The program will run to completion.
- The user will manually verify the points plotted correspond to the ones entered.

2. **Test ID #1.2**
   Type: Functional, Dynamic, Manual
   Initial State: None Object.
   Input: Instantiate Graph(6, data = [(1, 1), (2, 2), (3, 3), (4, 4)])
   Output: A window depicting a graph with plotted points at (1, 1), (2, 2), (3, 3), and (4, 4).
   How test will be performed:
   - The user will instantiate the graph object with data immediately.
   - The program will run to completion.
   - The user will manually verify the points plotted correspond to the ones entered.

3. **Test ID #1.3**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
   Input: The list object: [ ].
   Output: A window depicting a graph with no plotted points.
   How test will be performed:
   - The user will instantiate the graph object.
   - The list object will be passed into the plotting method in the API.
   - The program will run to completion.
   - The user will manually verify that no points were plotted.

4. **Test ID #1.4**
   Type: Functional, Dynamic, Manual
   Initial State: None Object.
   Input: Instantiate Graph(6, data = [ ])
   Output: A window depicting a graph with no plotted points.
   How test will be performed:

- The user will instantiate the graph object with data immediately.
- The program will run to completion.
- The user will manually verify that no points were plotted.

After this point, it should be also verified that instantiating a Graph object with data versus instantiating a Graph object and then setting the data should have no difference in the output. Therefore, after this point all test cases will (decided arbitrarily) initialize a Graph object and then set the data if the situation arises. This will be done purely to save time, which is a limitation.

3.1.2 Area of Testing 2

Requirement #2: The software will raise an exception if the data format cannot be plotted, and stop the program.

1. **Test ID #2.1**
   Type: Functional, Dynamic, Manual
   Initial State: A testing script that imports the method that validates data.
   Input: The list object: [(1, 1), (2, 2), (3, 3), (4, 4)]
   Output: A safe end to the execution (i.e, no exception raised).
   How test will be performed:
   - The user will call the method with the list object.
   - The user will verify the method completed execution without raising an exception.

2. **Test ID #2.2**
   Type: Functional, Dynamic, Manual
   Initial State: A testing script that imports the method that validates data.
   Input: The list object: [(1, 1), (2, 2), (3, 3), (4)]
   Output: The program raised an exception.
   How test will be performed:
   - The user will call the method with the list object.
   - The user will verify the method raised an exception.

3. **Test ID #2.3**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant..
   Input: Instantiate Graph(6, data = [(1, 1), (2, 2), (3, 3), (4)])
   Output: The list object: [(1, 1), (2, 2), (3, 3), (4)]
   How test will be performed:
   - The user will instantiate the graph object.
   - The list object will be passed into the plotting method in the API.
   - The user will manually verify an exception was raised.

2.1 and 2.2 directly test the method that is in charge of raising the exception. 2.3 tests whether it is used correctly in the library. We do not need to again test the positive scenario of 2.3 as this will have been tested with Area of Testing 1.

3.1.3 Area of Testing 3

Requirement #3: The software will construct a coordinate system that will fit all the data points.

1. **Test ID #3.1**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
   Input: The list object: [(1, 1), (2, 2), (3, 3), (17, 77)]
   Output: A window depicting a graph with max x axis value to be 17 and -17, and y axis to be 77 and -77.
   How test will be performed:
   - The user will instantiate the graph object.
   - The list object will be passed into the plotting method in the API.
   - The program will run to completion.
   - The user will manually verify the axes are appropriate.
2. **Test ID #3.2**
   Type: Functional, Dynamic, Manual

Initial State: Instantiate Graph(6) object with 6 markings on each quadrant.
Input: The list object: [(1, 1), (2, 2), (3, 3), (-17, -77)]
Output: A window depicting a graph with max x axis value to be 17 and -17, and y axis to be 77 and -77.
How test will be performed:

- The user will instantiate the graph object.
- The list object will be passed into the plotting method in the API.
- The program will run to completion.
- The user will manually verify the axes are appropriate.

3.1.4 Area of Testing 4

Requirement #4: The software will connect a line that passes through all the data points if the data points are a function of x.

This functionality is composed of three stages that were required to complete it:

I. The ability to plot points.

II. The ability to graph functions.

With the completion of the above 2, and through a method that determines a polynomial (which is a function) that passes through all the data points, we come to the third stage, which is the essence of this functionality:

III. A function that passes through all the data points.

The first stage is assumed to have been tested in Area of Testing 1, and will not be repeated here. Furthermore, because Stage II is an intermediary stage and not an official requirement (at least, not yet), it will be given one test case here, to save time.

1. **Test ID #4.1**
   Type: Functional, Dynamic, Manual
   Initial State: An instantiated Graph object.

Input: The default math.sin() function in python.
Output: A window depicting the sin() graph.
How test will be performed:

- The user will instantiate the graph object.
- The user will call the plot function method in the API with [Math.]sin() as the parameter.
- The program will run to completion.
- The user will manually verify that the program plotted the function correctly.

2. **Test ID #4.2**
Type: Functional, Dynamic, Manual
Initial State: An instantiated Graph object.
Input: The list object: [(1, 1), (2, 2), (3, 3), (4, 4)]
Output: A window depicting a graph with the points plotted, and a line is connecting all points.
How test will be performed:

- The user will instantiate the graph object.
- The user will call the appropriate method in the API with the data set.
- The program will run to completion.
- The user will manually verify that the program plotted the points, and a line is connecting all points.

3. **Test ID #4.3**
Type: Functional, Dynamic, Manual
Initial State: An instantiated Graph object.
Input: The list object: [(1, 1), (2, 2), (3, 3), (3, 4)]
Output: An exception will be raised.
How test will be performed:

- The user will instantiate the graph object.
- The user will call the appropriate method in the API with the data set.
- The program will run to completion.
- The user will manually verify that an exception was raised.

3.1.5 Area of Testing 5

Testing of source code.

1. **Test ID #5.1**
   Type: Functional, Static, Automated **Initial State:** Completed source code.
   Input: Source code.
   Output: Review of source code.
   How test will be performed:

   – Pylint (a python static testing framework) will be used to analyze the source code and determine if there are syntax errors, faulty naming conventions, unused variables, poor quality etc. This testing will be automated.

3.1.6 Area of Testing 6

Verifying correctness of output using automated testing.

The challenges in this form of automated testing lie because of the following tasks:

– There needs to be a comparison image that is also generated in an automated fashion.

– A comparison tool needs to exist that can compare 2 images and determine the difference on some scale.

– The difference scale needs to be normalized to account for static differences that are always present, such as in the way the axes are draw (line thickness for example), the size of the points, the spacing between one point to the next, etc.

1. **Test ID #6.1**
   Type: Functional, Dynamic, Automated **Initial State:** Completed source code.
   Input: An image from xPycharts, and a comparison image generated using the same dataset.?Input: The 2 images.
   Output: A numerical value denoting the difference between these two

images, and a pass/fail indicator.
How test will be performed:

- Using a python library such as PIL, an image of a graph generated by xPycharts will be saved.
- Using an external graphing library (matplotlib), a comparison image will be created.
- The script will then proceed to use these 2 images in the comparison tool denoted here.
- The program will output a value that will then be normalized by a ratio determined prior.
- The program will display the value, and indicate whether it met the threshold for a pass, or failed.

3.2 Tests for Nonfunctional Requirements

Title for Test

1. test-id1
   Type:

   Initial State:

   Input/Condition:

   Output/Result:

   How test will be performed:

2. test-id2
   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input:

   Output:

   How test will be performed:

3.2.1 Area of Testing2

...

4 Tests for Proof of Concept

4.1 Area of Testing1

Title for Test

1. test-id1
   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input:

   Output:

   How test will be performed:

2. test-id2
   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input:

   Output:

   How test will be performed:

4.2 Area of Testing2

...

5 Comparison to Existing Implementation

6 Unit Testing Plan

6.1 Unit testing of internal functions

6.2 Unit testing of output files

7 Appendix

This is where you can place additional information.

7.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

7.2 Usability Survey Questions?

This is a section that would be appropriate for some teams.