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 31, 2016

Contents

List of Tables

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

The testing schedule of the project is shown by the following Gantt Chart.

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

Note: see 3.1.1.

3.2.1 Look and feel

1. **Test ID #1**
   Type: Structural, Dynamic, Manual

   Initial State: Program is ready to produce one of each graph with random data.

   Input/Condition: Program generates one of each type of graph, with random data inputted

   Output/Result: 70% of users say that the graph is "Visually appealing" rather than "Visually unappealing."

   How test will be performed: Each user in the test group will be presented with a graph produced from random data. They will rate it as "visually appealing" or "visually unappealing." At least 70% of users must rate it as visually appealing for the test to be passed.

3.2.2 Usability

1. **Test ID #2,3,4,5**
   Type: Structural, Dynamic, Manual

   Initial State: Library is available on computer, IDLE is open, a dataset is downloaded on computer. User group has some knowledge of Python.

   Input/Condition: Users asked to create one of each graph without assistance

   Output/Result: 80% of users are able to produce each graph without assistance. They do not have issue with the linguistic content of the documentation. They produce these graphs in under twenty minutes.

   How test will be performed: The test group will be given a computer with IDLE open, and a directory on their desktop containing the documentation for this project and sample datasets. They will be asked to produce one of each graph available in this library and will not be given any assistance by the testers. They will be timed. When error messages are created, the user when surveyed will report 80% of the time that they could understand what the error was.

2. **Test ID #5.1**
   Type: Structural, Dynamic, Manual

   Initial State: Library is available on computer, IDLE is open, a dataset and a series of programs that incorrectly implement the library are downloaded on computer. User group has been acquainted with the library.

   Input/Condition: Users asked to use library in ways that throw each error message.

   Output/Result: Each error message is understood by users 80% of the time

   How test will be performed: Users will be asked to run programs that incorrectly implement the library. They will be asked to identify the error in implementation based on the error message. The correctness of their answers will be recorded.

3.2.3 Performance

1. **Test ID #6,9,10**
   Type: Structural, Dynamic, Automatic

   Initial State: Library is available on computer, minimal other programs are open

   Input/Condition: Script is executed with random datasets, averaging 500 points in a set

   Output/Result: Results written to text file. Graphs never take more than 20 seconds to produce, and never stall out the program

   How test will be performed: A script will repeatedly make a random data set, and time how long it takes to make each graph based on that data set. It will time the generation of the graphs and record this in a text file. After executing many times it will confirm that no graph took more than 20 seconds to produce, and that it did not stall out during execution.

3.2.4 Performance

1. **Test ID #6,9,10**
   Type: Structural, Dynamic, Automatic

   Initial State: Library is available on computer, minimal other programs are open. Automatic testing script is on computer.

   Input/Condition: Script is executed with random datasets, averaging 500 points in a set

   Output/Result: Results written to text file. Graphs never take more than 20 seconds to produce, and never stall out the program

   How test will be performed: A script will repeatedly make a random data set, and time how long it takes to make each graph based on that data set. It will time the generation of the graphs and record this in a text file. After executing many times it will confirm that no graph took more than 20 seconds to produce, and that it did not stall out during execution.

3.2.5 Operational and Environment

1. **Test ID #11,12,13,15**
   Type: Structural, Dynamic, Manual

   Initial State: Test uses Mac, Windows, Linux computers. Python is installed on computer, program that generates each graph with random dataset is on USB stick with the library (stored as zip file)

   Input/Condition: Program is executed on computers.

   Output/Result: All graphs are produced without error.

   How test will be performed: A USB key with a test program will be inserted into computers running Windows 10, Mac OS 10.9+, and the latest Ubuntu LTS. The program will run successfully.

2. **Test ID #11**
   Type: Structural, Dynamic, Manual

   Initial State: Library is available on Mac, Windows, Linux computer. Python is installed on computer, program that generates each graph with random dataset is on USB stick

   Input/Condition: Program is executed on computers.

   Output/Result: All graphs are produced without error.

   How test will be performed: A USB key with a test program will be inserted into computers running Windows 10, Mac OS 10.9+, and the latest Ubuntu LTS. The program will run successfully.

Same as described in 3.1.1.

4 Comparison to Existing Implementation

Edge cases will be tested in this section to ensure the program responds appropriately (determined by existing implementation) in non typical conditions.

Once again, it is important to note that because of limited resources, not every case will be tested and at times, a single test case would be sufficient in determining behaviour.

4.0.6 Area of Testing 1

Negative and Positive infinities as (x, y).

1. **Test ID #1.1**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant. Instantiate axisChart object (jCharts).
   Input: The list [(NEGATIVE_INFINITY, NEGATIVE_INFINITY), (POSITIVE_INFINITY, POSITIVE_INFINITY)]
   Output: Uncaught exception in xPycharts, caught exception in jCharts.
   How test will be performed:

   – The user will instantiate the graph objects from both libraries.
   – The list object will be passed into the plotting method of the APIs.
   – The user will manually verify the behaviors of the 2 libraries.

4.0.7 Area of Testing 2

None object as input.

1. **Test ID #2.1**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant. Instantiate axisChart object (jCharts).
   Input: The list [ ]
   Output: Empty graphs in both xPycharts and jCharts.
   How test will be performed:

   – The user will instantiate the graph objects from both libraries.
   – The list object will be passed into the plotting method of the APIs.
   – The user will manually verify the behaviours of the 2 libraries.

4.0.8 Area of Testing 3

Some (x, y) repeated n times where n is a very large number. This area of testing covers the program's speed and efficiency (i.e a non functional requirement). An efficient library would remove duplicates from the input and would complete this task in O(k) time, where k is the time it takes to remove or ignore the duplicates from the input. In contrast, a non efficient system would take O(n) time.

1. **Test ID #3.1**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant. Instantiate axisChart object (jCharts).
   Input: The list [(1, 1), ? , (1, 1)] with n spaces in between.
   Output: xPycharts would take a considerably long time. jCharts returns caught exception.
   How test will be performed:

   – The user will instantiate the graph objects from both libraries.
   – The list object will be passed into the plotting method of the APIs.
   – The user will manually verify the behaviours of the 2 libraries.

4.0.9 Area of Testing 4

Some (x, y, z) coordinates in input.

1. **Test ID #4.1**
   Type: Functional, Dynamic, Manual
   Initial State: Instantiate Graph(6) object with 6 markings on each quadrant. Instantiate axisChart object (jCharts).
   Input: The list [(1, 1, 1), (2, 2), (3, 3)]
   Output: xPycharts would plot (1, 1), (2, 2), and (3, 3). jCharts returns caught exception.
   How test will be performed:

   – The user will instantiate the graph objects from both libraries.
   – The list object will be passed into the plotting method of the APIs.
   – The user will manually verify the behaviours of the 2 libraries.

5 Unit Testing Plan

Unit testing will be done with unittest, the Python version of JUnit. Unit tests should be written as units are written.

5.1 Unit testing of internal functions

Unit testing for internal functions can be done on methods that return values. We will unit test with inputs that have clear expected outputs and with inputs that generate errors. We will aim for a high degree of code coverage with these unit tests, although the importance of graphics in this project means that there will be some functions that cannot be easily unit tested automatically. We will aim for 70% coverage, although as the project progresses we will revise this goal if it becomes apparent that it is not realistic. Stubs and drivers should not be needed for this project.

5.2 Unit testing of output files

The method described in functional test #6.1 can be employed to unit test output files. Using python scripts and image comparison, we can unit test output files from specific units.

References

6 Appendix

This is where you can place additional information.

6.1 Symbolic Parameters

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

6.2 Usability Survey Questions?

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