# SE 3XA3: Test Plan
# FlagGenerator

Team #2, Team Jakriel
Akram Hannoufa, hannoufa
Ganghoon (James) Park, parkg10
Nathaniel Hu, hun4

March 11, 2022

# Contents

# List of Tables

Table 1: **Revision History**

| **Date** | **Version** | **Notes** |
|---|---|---|
| March 3, 2022 | 1.0 | Initial Document |
| March 3, 2022 | 1.1 | Added initial draft of Tests for Proof of Concept section |
| March 8/9, 2022 | 1.2 | Made edits to draft of Tests for Proof of Concept section; Added initial draft of System Test Description section |
| March 10, 2022 | 1.3 | Made edits, added to draft of Tests for Proof of Concept section; Added to draft of System Test Description section |
| March 11, 2022 | 1.4 | Added Comparison to Existing Implementation, Unit Testing Plan and Appendix content |

This document describes the plan for software testing of Random Flag Generator.

# 1 General Information

## 1.1 Purpose

The purpose of this test plan is to describe the testing and validation process of The Random Flag Generator in detail. This test plan will properly access the functional and non-functional requirements of the software as well as performance and usability. The test cases will address each individual unit of source code and can be referred or updated with future implementations. The test plan aims to minimize the probability of error when used by the user.

## 1.2 Scope

The test plan considers and tests all functional requirements and nonfunctional requirements through different testing procedures. Test for proof of concept is also demonstrated. In addition, comparison to existing implementation is discussed to compare its functionality and improve the design. Unit testing plan for internal functions and output files is outlined. Finally, the usability survey questions will evaluate user experience. The test plan will be conducted accordingly, and the document will be revised with more development of the project.

## 1.3 Acronyms, Abbreviations, and Symbols

Table 2: **Table of Abbreviations**

| **Abbreviation** | **Definition** |
|---|---|
| GUI | Graphical User Interface |
| UI | User Interface |
| FR | Functional Requirement |
| RBG | Red Blue Green |
| LF | Look and Feel test |
| UH | Usability and Humanity test |
| PE | Performance test |
| SRS | Software Requirements Specification |
| PAGAN | **Python Avatar Generator for Absolute Nerds** |

Table 3: **Table of Definitions**

| **Term** | **Definition** |
|---|---|
| Python | The programming language used in this project. |
| Input String | The input of type string from the user. |
| Gallery | Collection of previously generated flags. |
| Hashing | Algorithm that converts input data to fixed-size value. Using a hash function to return an output that is usually a string or hexadecimal. |
| User | Person who uses or operates a computer program. |
| User Interface | Interactions between machines and humans occur |
| Graphical User Interface | A form of UI that allows users to use electronic devices to interact with graphics.. |
| System/Program | Collection of instructions or components that tell a computer how to operate. |
| Software Requirements Specification | A document that details what the prgoram/software will do and how it will accomplish the expected performance |
| Tester | An individual testing the software via the user interface or the code. |

## 1.4 Overview of Document

The test plan how the team is going to test the software and the automated testing tools. All requirements included in the SRS will be tested. Tests for proof of concept as well as comparison to existing implementation, unit testing, and usability survey questions are described.

# 2 Plan

## 2.1 Software Description

The Random Flag Generator is a user-friendly software that allows users to uniquely create a personalized random flag based on the entered input string. It is built using python and is inspired from PAGAN, an open source that generates an avatar from an input string.

## 2.2 Test Team

All members of group 2 are responsible for creating, executing, and documenting tests. The testing team is comprised of:

- Akram Hannoufa
- Ganghoon Park
- Nathaniel Hu

Upon completing the final stage of development, TAs, students, and other volunteers would be asked to provide feedback using the usability survey questions.

## 2.3 Automated Testing Approach

To review and validate the software product, automated testing for each output testing. It will be performed automatically by running the test program. The hash functions along with the flag generator functions will be tested automatically to ensure that the input string gives the expected output string and that it is repeatable.

## 2.4 Testing Tools

The main software testing framework that will be used is pytest, which will allow testers to write test cases in python and report the results. It is a simple, efficient, and effective way to handle increasingly complex testing requirements.

## 2.5 Testing Schedule

Please refer to project Gantt chart: ../ProjectSchedule/3XA3Ganttchart.pdf

# 3 System Test Description

## 3.1 Tests for Functional Requirements

### 3.1.1 User Interface Testing

**User Interface Test**

1. FR-01

   Type: Functional, Dynamic, Manual

   Initial State: UI is uninitialized

   Input: command to initilize and run the UI

   Output: UI is initialized

   How test will be performed: a command will be entered at the command line (by the user). The FlagGenerator main menu UI should open and allow the user to enter text into the input string field. The main menu should also present buttons for the instructions, flag gallery and settings menus.

2. FR-02

   Type: Functional, Dynamic, Manual

   Initial State: UI is initialized and running

   Input: mouse clicks on the instructions, flag gallery or settings menu buttons, or the appropriate keyboard strokes assigned to open each menu

   Output: the instructions, flag gallery or settings menu will be pulled up and can be viewed in its entirety

   How test will be performed: The user will have the FlagGenerator main menu UI already running, and then either click on the buttons or press the appropriate keyboard strokes to pull up each menu.

3. FR-03

   Type: Functional, Dynamic, Manual

   Initial State: UI is initialized and running

   Input: keyboard strokes to (re)enter text into the input string field

   Output: text appears in the input string field

   How test will be performed: The user will have the FlagGenerator main menu UI already running, and then (re)enter text into the input string field.

4. FR-04-08

   Type: Functional, Dynamic, Manual

   Initial State: UI is initialized and running

   Input: (text in input string field and) mouse click on generate flag button

   Output: flag will be generated using the input string and saved on the local machine (in generated_flags directory)

   How test will be performed: The user will have the FlagGenerator main menu UI already running, and then click the generate flag button to start flag generation.

5. FR-05-09

   Type: Functional, Dynamic, Manual

   Initial State: UI is initialized and running, a flag has already finished generating

   Input: (text in the input string field and) mouse click on the display flag button.

   Output: generated flag with be displayed to the user through the UI

   How test will be performed: The user will have the FlagGenerator main menu UI already running and already have a flag generated, and then click the display flag button to display the flag in the UI.

6. FR-06-10

   Type: Functional, Dynamic, Manual

   Initial State: UI is initialized and running, a flag has already finished generating

   Input: (text in the input string field,) saved changes to the settings and mouse click on the display flag button

   Output: flag will be regenerated using the input string and changed settings, and saved on the local machine (in generated_flags directory)

   How test will be performed: The user will have the FlagGenerator main menu UI already running and already have a flag generated (and displayed). The user then makes and saves some changes to the settings, and reruns the flag generation with the new settings in effect.

7. FR-07-11

   Type: Functional, Dynamic, Manual

   Initial State: UI is initialized and running

   Input: (text in the input string field and) mouse click on the generate flag button.

   Output: flag will be generated using the input string and saved on the local machine (in generated_flags directory) using the input string as its (default) name

   How test will be performed: The user will have the FlagGenerator main menu UI already running, and then click the generate flag button to start flag generation. The user will then open up the flag gallery menu, and the new generated flag should be displayed there.

8. FR-12-13

   Type: Functional, Dynamic, Manual

   Initial State: UI is initialized and running

   Input: mouse click on the instructions button, or the appropriate keyboard stroke assigned to open the instructions menu

Output: the instructions menu will be pulled up and can be viewed in its entirety, and a button will be displayed that the user can click on to close the instructions menu and return to the main menu

How test will be performed: The user will have the FlagGenerator main menu UI already running, and then either click on the instructions button or press the appropriate keyboard stroke to pull up the instructions menu. After viewing the instructions menu, the user will then click on the return to main menu button to close the instructions window and return to the main menu.

9. FR-14

   Type: Functional, Dynamic, Manual

   Initial State: UI is initialized and running

   Input: mouse click on the settings button, or the appropriate keyboard stroke assigned to open the settings menu, and clicks/dragging to change settings

   Output: the settings menu will be pulled up where settings can be changed

   How test will be performed: The user will have the FlagGenerator main menu UI already running, and then either click on the settings button or press the appropriate keyboard stroke to pull up the settings menu and change settings.

10. FR-15

    Type: Functional, Dynamic, Manual

    Initial State: UI is initialized and running

    Input: mouse click on the settings button, or the appropriate keyboard stroke assigned to open the settings menu

    Output: the settings menu will be pulled up where the flag generator version will be displayed

    How test will be performed: The user will have the FlagGenerator main menu UI already running, and then either click on the settings button

or press the appropriate keyboard stroke to pull up the settings menu to confirm that the flag generator version number is displayed properly.

11. FR-16

    Type: Functional, Dynamic, Manual

    Initial State: UI is initialized and running

    Input: mouse click on the settings button, or the appropriate keyboard stroke assigned to open the settings menu

    Output: the settings menu will be pulled up where a return to main menu button will be displayed that the user can click on to close the settings menu and return to the main menu

    How test will be performed: The user will have the FlagGenerator main menu UI already running, and then either click on the settings button or press the appropriate keyboard stroke to pull up the settings menu. After viewing the settings menu, the user will then click on the return to main menu button to close the instructions window and return to the main menu.

12. FR-17

    Type: Functional, Dynamic, Manual

    Initial State: UI is initialized and running

    Input: mouse click on the flag gallery button, or the appropriate keyboard stroke assigned to open the flag gallery menu

    Output: the flag gallery menu will be pulled up where a list of all flags and the input strings used to generate them will be displayed

    How test will be performed: The user will have the FlagGenerator main menu UI already running, and then either click on the flag gallery button or press the appropriate keyboard stroke to pull up the flag gallery menu for viewing.

13. FR-18

    Type: Functional, Dynamic, Manual

Initial State: UI is initialized and running

Input: mouse click on the flag gallery button, or the appropriate keyboard stroke assigned to open the flag gallery menu

Output: the flag gallery menu will be pulled up where an input string field will allow the user to search for flags by input string

How test will be performed: The user will have the FlagGenerator main menu UI already running, and then either click on the flag gallery button or press the appropriate keyboard stroke to pull up the flag gallery menu. The user will then input text in the input string field to search for flags by input string.

14. FR-19

    Type: Functional, Dynamic, Manual

    Initial State: UI is initialized and running

    Input: mouse click on the flag gallery button, or the appropriate keyboard stroke assigned to open the flag gallery menu

    Output: the flag gallery menu will be pulled up where a button will be displayed that the user can click on to close the flag gallery menu and return to the main menu

    How test will be performed: The user will have the FlagGenerator main menu UI already running, and then either click on the flag gallery button or press the appropriate keyboard stroke to pull up the flag gallery menu. After viewing the flag gallery menu, the user will then click on the return to main menu button to close the flag gallery window and return to the main menu.

### 3.1.2 Output Testing

**Output Test**

1. test_get_hash_algo

    Type: Functional, Dynamic, Automated

    Initial State:

Input: input string of hashing algorithm name

Output: hashlib hashing algorithm

How test will be performed: automatically by running the test program containing the test set

2. test_get_hash_hex

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: input string to be used to generate flag, hashlib hashing algorithm

   Output: hash digest of input string in hexadecimal form

   How test will be performed: automatically by running the test program containing the test set

3. test_hash_generator

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: input string to be used to generate flag

   Output: hash digest of input string in hexadecimal form

   How test will be performed: automatically by running the test program containing the test set

4. test_pad_hashcode

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: output hash string to be used to generate flag

   Output: output hash string padded to minimum hash length

   How test will be performed: automatically by running the test program containing the test set

5. test_grind_hash_for_colors

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: output hash string to be used to generate flag

   Output: list of five colors' RGB values (R, G, B)

   How test will be performed: automatically by running the test program containing the test set

6. test_grind_hash_for_stripe_style

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: output hash string to be used to generate flag

   Output: output string of stripe style to be used to generate flag

   How test will be performed: automatically by running the test program containing the test set

7. test_grind_hash_for_stripe_number

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: output hash string to be used to generate flag

   Output: output string of stripe number to be used to generate flag

   How test will be performed: automatically by running the test program containing the test set

8. test_grind_hash_for_symbol_locations

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: output hash string to be used to generate flag

Output: output string of symbol location to be used to generate flag

How test will be performed: automatically by running the test program containing the test set

9. test_grind_hash_for_symbol_number

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: output hash string to be used to generate flag

   Output: output string of symbol number to be used to generate flag

   How test will be performed: automatically by running the test program containing the test set

10. test_grind_hash_for_symbol_types

    Type: Functional, Dynamic, Automated

    Initial State:

    Input: output hash string to be used to generate flag

    Output: output string of symbol type to be used to generate flag

    How test will be performed: automatically by running the test program containing the test set

11. test_hex2rgb

    Type: Functional, Dynamic, Automated

    Initial State:

    Input: output hexidecimal color code string to be used to generate flag

    Output: output tuple of RGB values for color derived from heximdecimal color code string to be used to generate flag

    How test will be performed: automatically by running the test program containing the test set

12. test_generate_flag

    Type: Functional, Dynamic, Manual

    Initial State:

    Input: input string, hashing algorithm name string to be used to generate flag

    Output: generated flag image file

    How test will be performed: manually by running the test program containing the test set, and then doing a manual visual comparison of the generated flag with its expected generated flag output

13. test_generate_flag_data

    Type: Functional, Dynamic, Automated

    Initial State:

    Input: input string, hashing algorithm name string to be used to generate flag

    Output: tuple consisting of list of 5 tuples of RGB values, tuple of stripe style, number and tuple of symbol location, number and type

    How test will be performed: automatically by running the test program containing the test set

## 3.2 Tests for Nonfunctional Requirements

### 3.2.1 Look and Feel Testing

**Appearance Test**

1. LF-01

   Type: Manual, Nonfunctional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification of UI simplicity, intuitivity, and clutterlessness

How test will be performed: The FlagGenerator UI will be analyzed for how simple, intuitive and non-cluttered it is for users.

**Style Test**

1. LF-02

   Type: Manual, Functional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition: input string, hashing algorithm name string to generate a flag

   Output/Result: flag generated using input string and hashing algorithm

   How test will be performed: The user will enter an input string and hashing algorithm string name to generate a flag. Then user will then review the generated flag for image quality and aesthetics.

2. LF-03

   Type: Manual, Functional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition: input string, hashing algorithm name string to generate a flag

   Output/Result: flag generated using input string and hashing algorithm

   How test will be performed: The user will enter an input string and hashing algorithm string name to generate a flag. Then user will then review the generated flag to ensure the colours used are within certain colour ranges.

3. LF-04

   Type: Manual, Functional, Dynamic

   Initial State: UI is initialized and running

Input/Condition: input string, hashing algorithm name string to generate a flag

Output/Result: flag generated using input string and hashing algorithm

How test will be performed: The user will enter an input string and hashing algorithm string name to generate a flag. Then user will then review the generated flag to ensure all flag components are visible in the generated image.

### 3.2.2 Usability and Humanity Testing

**Ease of Use Test**

1. UH-01

   Type: Manual, Nonfunctional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification of the logical flow of the UI components

   How test will be performed: The FlagGenerator UI will be analyzed for the logical flow of the UI components

2. UH-02

   Type: Manual, Functional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification that all tasks can be done within one interface

   How test will be performed: Various tasks will be performed (e.g. reading the instructions, changing the settings) to ensure all tasks can be done within one interface, and that no tasks need interactions with more than one interface to be completed.

3. UH-03

   Type: Manual, Nonfunctional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification that FlagGenerator is easy to use for people aged 7 or older

   How test will be performed: Various tasks will be performed (e.g. reading the instructions, changing the settings) to ensure all tasks can be easily understood and completed by people aged 7 or older.

4. UH-04

   Type: Manual, Nonfunctional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification that all keyboard inputs to perform actions are intuitive for users

   How test will be performed: Various tasks will be performed (e.g. reading the instructions, changing the settings) to ensure all tasks can be easily understood and completed using keyboard inputs that are intuitive and easy to remember and use.

**Personalization Test**

1. UH-05

   Type: Manual, Functional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification of the modifiability of output specifications for flag generation (e.g. type of hashing, image file, etc.)

How test will be performed: The FlagGenerator UI will be analyzed to ensure the output specifications for flag generation can (easily) be modified.

2. UH-06

   Type: Manual, Functional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification of the modifiability of the flag gallery

   How test will be performed: The FlagGenerator UI will be analyzed to ensure the flag gallery can (easily) be modified (i.e. add, delete flags).

**Learning Test**

1. UH-07

   Type: Manual, Nonfunctional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification of the learnability of the FlagGenerator program

   How test will be performed: The FlagGenerator UI will be analyzed to ensure the user can learn to use the program with no prior experience. A sample group of users with no prior experience will also attempt to use the peogram (and 85% of users should be able to with ease).

2. UH-08

   Type: Manual, Nonfunctional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification of the accessibility of the instructions menu

How test will be performed: The FlagGenerator UI will be analyzed to ensure the user can (easily) access the instruction menu for help. A sample group of users will also attempt to use the program (and 85% of users should be able to pull up the instructions menu within 5 seconds).

3. UH-09

   Type: Manual, Nonfunctional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification of the intuitivity/informativeness of the main UI

   How test will be performed: The FlagGenerator UI will be analyzed to ensure the user can (easily) use the program by following the main UI instructions. A sample group of users will also attempt to use the program (and 85% of users should be able to do so just using the main UI instructions).

**Understandability Test**

1. UH-10

   Type: Manual, Nonfunctional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification of consistent language use throughout the program

   How test will be performed: The FlagGenerator UI will be analyzed to ensure that consistent language is used throughout the program.

2. UH-11

   Type: Manual, Nonfunctional, Dynamic

   Initial State: UI is initialized and running

Input/Condition:

Output/Result: verification of simplified terminology being used wherever possible

How test will be performed: The FlagGenerator UI will be analyzed to ensure that simplified terminology is used wherever possible in the program.

**Accessibility Test**

1. UH-12

   Type: Manual, Nonfunctional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification of easy-to-read fonts and font sizes being used throughout the program

   How test will be performed: The FlagGenerator UI will be analyzed and compared with the web accessibility standards to ensure that easy-to-read fonts and font sizes are used throughout the program.

2. UH-13

   Type: Manual, Nonfunctional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition:

   Output/Result: verification of a colour-blind friendly FlagGenerator UI

   How test will be performed: The FlagGenerator UI will be analyzed to ensure that a colour-blind friendly UI is provided.

### 3.2.3 Performance Testing

**Speed Test**

1. PE-01-02

   Type: Manual, Functional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition: input string and hashing algorithm name string to generate flag, mouse click

   Output/Result: output generated flag image

   How test will be performed: The FlagGenerator program will be given an input string and hashing algorithm name string, and the time it takes to generate/ download a flag image shall be timed. The generation and download time shall not exceed 5 seconds.

2. PE-03

   Type: Manual, Functional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition: mouse click

   Output/Result: user's flag gallery should be loaded in the UI

   How test will be performed: The FlagGenerator UI program will open the user's flag gallery and the duration shall be timed. The loading time shall not exceed 5 seconds.

**Precision or Accuracy Test**

1. PE-04

   Type: Manual, Functional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition: input string and hashing algorithm name string to generate flag

   Output/Result: output hexadecimal hash digest (per the selected hashing algorithm)

   How test will be performed: The FlagGenerator program will be given an input string and hashing algorithm name string, and the output

hexadecimal hash digest (per the selected hashing algorithm) will be compared with the hash digest generated with the input string using an external hashing program.

2. PE-05

   Type: Manual, Functional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition: input string and hashing algorithm name string to generate flag, mouse click

   Output/Result: output generated flag image

   How test will be performed: The FlagGenerator program will be given an input string and hashing algorithm name string, and the generated flag image will be analyzed to ensure the various components are accurately placed (i.e. they match the template files).

3. PE-06

   Type: Manual, Functional, Dynamic

   Initial State: UI is initialized and running

   Input/Condition: input string and hashing algorithm name string to generate flag, mouse click

   Output/Result: output generated flag image

   How test will be performed: The FlagGenerator program will be given an input string and hashing algorithm name string, and the generated flag image will be analyzed to ensure the various colours used match the hexadecimal RGB colour values derived from the hash digest. The RGB volour values will be run through an external colour tool, and those colours will be compared with the generated flag's colours to ensure they match.

**Reliability and Availability Test**

1. PE-07

Type: Manual, Nonfunctional, Dynamic

Initial State: GUI is uninitialized

Input/Condition: command to initialize and run the GUI

Output/Result: GUI is initialized and running

How test will be performed: The FlagGenerator program will be initialized and run from the command line at several different times during the day to ensure its availability.

**Capacity Test**

1. PE-08

   Type: Manual, Functional, Dynamic

   Initial State: UI is initialized and running, maximum number of generated flags currently stored in user's gallery

   Input/Condition: input string and hashing algorithm name string to generate flag, mouse click

   Output/Result: program displays an error message telling the user the maximum number of generated flags has been reached, and to delete flags in the gallery to make more space for new flags.

   How test will be performed: The FlagGenerator program will be running and the user will attempt to generate a new flag despite the gallery being full (i.e. the flag generation should fail and an error message should pop up).

2. PE-09

   Type: Manual, Nonfunctional, Dynamic

   Initial State: GUI is uninitialized

   Input/Condition: command to initialize and run the GUI

   Output/Result: GUI is initialized and running

   How test will be performed: The FlagGenerator program will be initialized and run from the command line, and only one user is initialized per machine (i.e. all generated flags are stored in the same gallery/directory on the machine).

**Scalability or Extensibility Test**

1. PE-10

   Type: Manual, Nonfunctional, Dynamic

   Initial State: GUI is uninitialized

   Input/Condition: a new hashing algorithm/function option is added, command to initialize and run the GUI

   Output/Result: program should be able to generate new flags using the new hashing algorithm/function without any changes needing to be made to the existing code.

   How test will be performed: A new hashing algorithm/function will be added to the FlagGenerator program, and then the GUI will be initialized. The user will attempt to generate a new flag using the new hashing algorithm and an input string, and the generated flag should be different from all of the other flags generated using that input string and all other existing hashing algorithms.

2. PE-11

   Type: Manual, Nonfunctional, Dynamic

   Initial State: GUI is uninitialized

   Input/Condition: new flag components are added (i.e. new .jka flag asset files), command to initialize and run the GUI

   Output/Result: program should be able to generate new flags using the new flag asset files with only minor to no changes being made to the existing code.

   How test will be performed: A new (or several) flag asset(s) will be added to the FlagGenerator program, and then the GUI will be initialized. The user will attempt to generate a new flag using the new flag asset(s) with specific input strings (determining by trial and error), and the generated flag should use the new flag asset(s).

### 3.2.4 Operational and Environmental Testing

**Expected Physical Environment Test**

1. PE-13

   Type: Manual, Nonfunctional, Dynamic

   Initial State: GUI is uninitialized

   Input/Condition: local machine has Wi-Fi disabled, command to initialize and run the GUI

   Output/Result: GUI is initialized and running

   How test will be performed: The FlagGenerator program will be initialized and run from the command line on a local machine with the Wi-Fi disabled. This program should still function as it would even with the Wi-Fi enabled.

2. PE-14

   Type: Manual, Nonfunctional, Dynamic

   Initial State: GUI is uninitialized

   Input/Condition: local machine can support (running) the Python language, command to initialize and run the GUI

   Output/Result: GUI is initialized and running

   How test will be performed: The FlagGenerator program will be initialized and run from the command line on several different local machines that can support (running) the Python language. This program should still function the same on all the different local machines.

### 3.2.5 Interfacing with Adjacent Systems Testing

**Interfacing with Adjacent Systems Test**

1. PE-15

   Type: Manual, Nonfunctional, Dynamic

   Initial State: GUI is uninitialized

Input/Condition: command to initialize and run the GUI, user generates some flag(s)

Output/Result: GUI is initialized and running, new flag(s) generated and saved to the user's local machine

How test will be performed: The FlagGenerator program will be initialized and run from the command line, and the user will attempt to generate some flag(s). The user will then check that the program has only created/altered files within its working directory (i.e. generated_flags directory), and no where else on the local machine.

## 3.3 Traceability Between Test Cases and Requirements

In terms of traceability, most test cases and requirements have a one to one relationship, with exceptions and special cases noted. Some functional requirements are covered using the same tests, because they are essentially the same requirements, but are a part of different business events. See the following traceability matrices for more details.

Table 4: **Traceability Matrix for Functional Requirements I**

| | Requirements | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Test Cases | FR1 | FR2 | FR3 | FR4 | FR5 | FR6 | FR7 | FR8 | FR9 | FR10 | FR11 | FR12 | FR13 | FR14 |
| FR-01 | X | | | | | | | | | | | | | |
| FR-02 | | X | | | | | | | | | | | | |
| FR-03 | | | X | | | | | | | | | | | |
| FR-04-08 | | | | X | | | | X | | | | | | |
| FR-05-09 | | | | | X | | | | X | | | | | |
| FR-06-10 | | | | | | X | | | | X | | | | |
| FR-07-11 | | | | | | | X | | | | X | | | |
| FR-12-13 | | | | | | | | | | | | X | X | |
| FR-14 | | | | | | | | | | | | | | X |

Table 5: **Traceability Matrix for Functional Requirements II/Non-Functional Requirements I**

| | Requirements | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Test Cases | FR15 | FR16 | FR17 | FR18 | FR19 | LF1 | LF2 | LF3 | LF4 | UH1 | UH2 | UH3 | UH4 | UH5 |
| FR-15 | X | | | | | | | | | | | | | |
| FR-16 | | X | | | | | | | | | | | | |
| FR-17 | | | X | | | | | | | | | | | |
| FR-18 | | | | X | | | | | | | | | | |
| FR-19 | | | | | X | | | | | | | | | |
| LF-01 | | | | | | X | | | | | | | | |
| LF-02 | | | | | | | X | | | | | | | |
| LF-03 | | | | | | | | X | | | | | | |
| LF-04 | | | | | | | | | X | | | | | |
| UH-01 | | | | | | | | | | X | | | | |
| UH-02 | | | | | | | | | | | X | | | |
| UH-03 | | | | | | | | | | | | X | | |
| UH-04 | | | | | | | | | | | | | X | |
| UH-05 | | | | | | | | | | | | | | X |

Table 6: **Traceability Matrix for Non-Functional Requirements II**

| | Requirements | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Test Cases | UH6 | UH7 | UH8 | UH9 | UH10 | UH11 | UH12 | UH13 | PE1 | PE2 | PE3 | PE4 | PE5 | PE6 |
| UH-06 | X | | | | | | | | | | | | | |
| UH-07 | | X | | | | | | | | | | | | |
| UH-08 | | | X | | | | | | | | | | | |
| UH-09 | | | | X | | | | | | | | | | |
| UH-10 | | | | | X | | | | | | | | | |
| UH-11 | | | | | | X | | | | | | | | |
| UH-12 | | | | | | | X | | | | | | | |
| UH-13 | | | | | | | | X | | | | | | |
| PE-01-02 | | | | | | | | | X | X | | | | |
| PE-03 | | | | | | | | | | | X | | | |
| PE-04 | | | | | | | | | | | | X | | |
| PE-05 | | | | | | | | | | | | | X | |
| PE-06 | | | | | | | | | | | | | | X |

Table 7: **Traceability Matrix for Non-Functional Requirements III**

| | Requirements | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Test Cases | PE7 | PE8 | PE9 | PE10 | PE11 | PE13 | PE14 | PE15 |
| PE-07 | X | | | | | | | |
| PE-08 | | X | | | | | | |
| PE-09 | | | X | | | | | |
| PE-10 | | | | X | | | | |
| PE-11 | | | | | X | | | |
| PE-13 | | | | | | X | | |
| PE-14 | | | | | | | X | |
| PE-15 | | | | | | | | X |

# 4 Tests for Proof of Concept

## 4.1 Hash Generator

**test_get_hash_algo**

1. test-01

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: 'sha256'

   Output: hashlib.sha256

   How test will be performed: automatically by running the test program containing the test set

2. test-02

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: 'md5'

   Output: hashlib.md5

   How test will be performed: automatically by running the test program containing the test set

**test_get_hash_hex**

1. test-03

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: 'test', hashlib.sha256

   Output: '9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08'

   How test will be performed: automatically by running the test program containing the test set

**test_hash_generator**

1. test-04

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: ‘sample’

   Output: ’af2bdbe1aa9b6ec1e2ade1d694f41fc71a831d0268e9891562113d8a62add1bf’

   How test will be performed: automatically by running the test program containing the test set

## 4.2 Hash To Flag

**test_pad_hashcode**

1. test-05

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: ‘575758’

   Output: ’57575857575857575857575857575758’

   How test will be performed: automatically by running the test program containing the test set

**test_grind_hash_for_colors**

1. test-06

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: ‘0396233d5b28eded8e34c1bf9dc80fae34756743594b9e5ae67f4f7d124d2e3d’

   Output: [(3, 150, 35), (61, 91, 40), (237, 237, 142), (52, 193, 191), (157, 200, 15)]

   How test will be performed: automatically by running the test program containing the test set

**test_grind_hash_for_stripe_style**

1. test-07

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: '5555556c48be1c0b87a7d575c73f6e42fnfhasdf341123abadbdbd'

   Output: 'VERTICAL'

   How test will be performed: automatically by running the test program containing the test set

**test_grind_hash_for_stripe_number**

1. test-08

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: '5555556c48be1c0b87a7d575c73f6e42fnfhasdf341123abadbdbd'

   Output: 'THREE'

   How test will be performed: automatically by running the test program containing the test set

**test_grind_hash_for_symbol_locations**

1. test-09

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: '5555556c48be1c0b87a7d575c73f6e42fnfhasdf341123abadbdbd'

   Output: 'TOP_LEFT'

   How test will be performed: automatically by running the test program containing the test set

**test_grind_hash_for_symbol_number**

1. test-10

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: '5555556c48be1c0b87a7d575c73f6e42fnfhasdf341123abadbdbd'

   Output: 'ONE'

   How test will be performed: automatically by running the test program containing the test set

**test_grind_hash_for_symbol_types**

1. test-11

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: '00001000010000010000000999999'

   Output: 'CROSS'

   How test will be performed: automatically by running the test program containing the test set

**test_hex2rgb**

1. test-12

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: '#ffffff'

   Output: (255, 255, 255)

   How test will be performed: automatically by running the test program containing the test set

## 4.3 Flag Generator

**test_generate_flag**

1. test-13

   Type: Functional, Dynamic, Manual

   Initial State:

   Input: 'Sword is pointy', 'sha256'

   Output: flag_Sword is pointy.png

   How test will be performed: manually by running the test program containing the test set, and then doing a manual visual comparison of the generated flag with its expected generated flag output

**test_generate_flag_data**

1. test-14

   Type: Functional, Dynamic, Automated

   Initial State:

   Input: 'test', 'sha256'

   Output: ([(159, 134, 208), (129, 136, 76), (125, 101, 154), (47, 234, 160), (197, 90, 208)], ('VERTICAL', 'THREE'), ('CENTER', 'ZERO', 'CROSS'))

   How test will be performed: automatically by running the test program containing the test set

## 4.4 GUI

**test_GUI**

1. test-15

   Type: Functional, Dynamic, Manual

   Initial State: GUI is uninitialized

Input: command to initialize and run the GUI

Output: GUI is initialized

How test will be performed: a command will be entered at the command line (by the user). The FlagGenerator main menu UI should open and allow the user to enter text into the input string field. The main menu should also present buttons for generating and displaying flags, which the user will then click on to test that they are functioning as prescribed.

# 5 Comparison to Existing Implementation

The flag generation program currently implements the majority of the core functionality, found in the original PAGAN program. Mainly, Flag Generator currently implements hash generation using hashlib, mapping the hash to various flag design decisions (array indices), and generating the flag image. In addition, a minimal GUI has been implmeneted to handle user input and displaying flag output. The next steps are expanding test cases and test coverage, implementing more flag design options (including more resolution choices), and enhancing the GUI, which includes the addition of a flag gallery.

# 6 Unit Testing Plan

## 6.1 Unit testing of internal functions

The pytest library will be used to carry out all unit testing for internal functions used in Flag Generator. 5 python files will be made that will contain the pytest testing functions for the modules contained within each of the 5 main components of the project. FlagGenerator, GUI, HashGenerator, HashToFlag, and JKAReader will each have a corresponding Test python file. Within each main component, most but not all internal functions are able to be unit tested. The others will require some manual testing. This is especially evident in the testing of the GUI, which will need to be done by the user to check functionality and usability requirements. For the modules that are able to be unit tested (modules mentioned in 3.1.2), assert statements will be setup to check that the modules are returning the correct values (based on externally calculated values), and that they are behaving as expected.

The "expected" cases, boundary/limit cases, and any excpetions and error handling will be tested through pytest unit tests. Using the testing coverage matrices, any unit testable funcional requirements will be covered in the unit tests. Use of external libraries (PIL, tkinter) will have less thorough testing, since they are commercially available software products, and assumed to have some level of correctness and already done testing.

## 6.2 Unit testing of output files

Some aspects of the outputted flag image will be able to be validated through unit testing, but others will be a visual/manual check. Flag generated images will have their width and height checked to ensure the user's resolution selection matches the output image resolution. Checking known hashing function outputs to make sure the corresponding flag design (colours, symbols, stripes, etc.) are all correct, will be done to verify some correctness in the generated symbols, although it will be unreasonable to find hashes for every single combination of possible output designs. Unit tests can also be done to check the naming of files, file storage, and gallery display correctness.

# 7 Appendix

## 7.1 Symbolic Parameters

N/A

## 7.2 Usability Survey Questions?

Given that the Flag Generator program heavily relies on user input and their interactions with the GUI, as well as their opinion on the outputs of the program, survey questions to gauge user experience will be used.

1. On a scale of 1-10, how easy was it for your to navigate the program using the GUI?
2. On a scale of 1-10, how appealing/aesthetically pleasing would you consider the flag images being generated?
3. On a scale of 1-10, how straightforward was it to follow the intended flow of operations? (ie. entering input –¿ accessing generated flag image)
4. On a scale of 1-10, how helpful was the "Help" button feature?
5. Would you like more (or less) ability to set certain parts of the flag?
6. Do you like having the flag gallery view? Is it worthwhile in your opinion?
7. What other designs would you like to see available for the flags being generated?
8. What parts of the GUI were not clear or straightforward to use?