SE 3XA3: Test Plan

Lines Per Minute (lpm)

Team #16, Lines Per Minute (lpm)
Jay Mody - modyj - 400195508
Jessica Lim - limj31 - 400173669
Maanav Dalal - dalalm1 - 400178115

March 30, 2021

Table 1: Revision History

Date	**Developer(s)**	**Change**
February 23, 2021	Jay/Jessica/Maanav	Initial document write-up.
March 2, 2021	Jay/Jessica	Tests for Functional requirements.
March 2, 2021	Maanav	Tests for Non-Functional requirements.
March 4, 2021	Jay/Jessica/Maanav	Finish sections 1, 4, 5 and 6
March 25, 2021	Jay/Jessica/Maanav	Test Plan Revision for R1

Contents

List of Tables

1 General Information

1.1 Purpose

The purpose of this document is to the testing plan for Lines Per Minute. This document with outline the tests that will be implemented for the software program lpm. It also will provide traceability and mapping to the requirements outlined in the SRS document. This document will act as an testing outline to ensure that the software product meets all predetermined requirements.

1.2 Scope

This document outlines tests to ensure the functional and non-functional requirements for lpm are met. Automated, dynamic, static and manual tests are all outlines in this document. This document outlines the frameworks that will be used for testing. It also provides Unit Tests and an outline of testing for the Proof-of-concept.

1.3 Acronyms, Abbreviations, and Symbols

- The terms "**terminal**" and "**command line**" are used interchangeably.
- **Command line interface** (abbreviated as **CLI**) is an interface for an application provided through the command line.
- **Command line application**: An application that is delivered using a CLI
- **Typing interface**: The interface that users practice their typing.
- **FR**: Functional requirement.
- **NFR**: Non-functional requirement.
- The terms "**system**" and "**application**", and "**product**" all refer to the lpm application this document specifies.
- **pip**: A package manager for python (**?**).
- **PyPI**: The python package index, which serves python packages through the pip tool (**?**).
- **Pylint**: Pylint is a Python static code analysis tool which looks for programming errors, helps enforcing a coding standard, sniffs for code smells and offers simple refactoring suggestions.
- **pytest**: A Python testing library. It will be used to create and execute unit tests.
- **black**: A Python code formatter. It will be used to ensure our code is consistent throughout our codebase.

1.4 Overview of Document

This document will outline the Testing Plan for lpm, which is a software program adapted from wpm. The test plan will describe how the functional and non-functional requirements will be satisfied using the PyTest framework and manual tests.

2 Plan

2.1 Software Description

The software product lpm is a command-line tool that will allow users to measure their typing speed when typing code passages. The software product is built with inspiration from the Pypi package wpm. The product is built using Python and is tested via the Pytest package.

2.2 Test Team

The test team will consist of Maanav Dalal, Jay Mody and Jessica Lim.

2.3 Automated Testing Approach

The team will be using automated testing by employing GitLab's CI/CD Pipelines. This will allow us to run multiple different tests upon every commit, ensuring that code which is eventually merged meets the code quality we have set for our project. This will involve three main procedures:

1. Linting using Pylint for static analysis
2. Code formatting using the black code formatter.
3. Dynamic testing of our code using pytest.

Through this automated procedure, it is ensured that any new commits made will conform to our code requirements and the developers are immediately informed about any breaking changes. We have decided that this is the optimal approach to testing as Pylint, black, and pytest are popular libraries for linting, code formatting, and testing respectively. As such, there is a lot of support for these libraries and them being frequently used speaks to their clear benefits in development.

2.4 Testing Tools

Pytest will be be main testing tool for this project. Pytest test modules will be used for unit testing. Pytest reports will also be consulted to ensure complete test coverage.

2.5 Testing Schedule

See Gantt Chart with the testing plan at the following link:

3 System Test Description

3.1 Tests for Functional Requirements

3.1.1 Command Line Interface

Start Typing Interface

1. **test-CLI1**: Start Typing Interface
   Type: Functional, Manual
   Initial State: Newly started terminal session.
   Input: lpm
   Output: The lpm typing interface, all languages should be part of the list of available code snippets.
   Description: The test member runs the "lpm" command from various directories to ensure the program is accessible and runs from any directory.

2. **test-CLI2:** Start the Typing Interface with Specific Languages
   Type: Functional, Manual
   Initial State: Newly started terminal session.
   Input: lpm python, lpm python java javascript
   Output: The lpm typing interface with only python code snippets, the lpm typing interface with python, ~~and~~ java and javascript code snippets
   Description: The test member runs the inputs in terminal and verifies that only the correct code snippet languages are displayed. The first test ensures a single language can be loaded, the second ensures multiple languages can be loaded.

Show Help Menu

1. **test-CLI3**: Show Help Menu
   Type: Functional, Manual
   Initial State: Terminal session with lpm installed~~Newly started terminal session.~~
   Input: lpm –help
   Output: The text-based help menu
   Description: The test member runs the input command and ensures the help menu is displayed is displayed in the output.

Show Typing Statistics

1. **test-CLI4**: Show Statistics
   Type: Functional, Manual
   Initial State: Terminal session with lpm installed ~~Newly started terminal session.~~
   Input: A. lpm –stats (with existing history) B. lpm –stats (with no history)
   Output: A. The user's lifetime typing statistics B. Display that there is no history
   Description: The test member runs the input command and ensures that the typing statistics are displayed in the output.

Change Settings

1. **test-CLI5**: Change Settings
   Type: Functional, Manual
   Initial State: Terminal session with lpm installed ~~Newly started terminal session.~~
   Input: lpm –settings
   Output: Open a vim session with lpmconfig.json file in terminal ~~The users settings menu.~~
   Description: The test member runs the input command, changes a setting, and ensures it is reflected when they run the program again.

3.1.2 Typing Editor

Code Snippet Navigation

1. **test-TE1**: Display Randomized Order Snippet
   Type: Functional, Manual
   Initial State: Terminal session with lpm installed ~~Newly started terminal session.~~
   Input: lpm, lpm
   Output: Typing interface with random code snippet, Typing interface with random code snippet
   Description: The test member runs the input command lpm, and moves to the previous and next code snippets, taking note of what snippets they have encountered. They then quit run the lpm command again, and ensure that a different order of snippets is shown.

2. **test-TE2**: Next/Prev
   Type: Functional, Manual
   Initial State: Terminal session with lpm installed ~~Newly started terminal session.~~
   Input: lpm, then the following combination of arrow keys is pressed: →, ←, ←, →
   Output: The initial code snippet, the next code snippet, the initial code snippet, the previous code snippet, the initial code snippet
   Description: The test member runs the input command lpm, and verifies that the user is able to move to the next and previous code snippets using the arrow keys.

3. **test-TE3**: Start ~~/Stop~~
   Type: Functional, Manual
   Initial State: Terminal session with lpm installed ~~Newly started terminal session.~~
   Input: lpm, Type character that is not escape or the arrowkeys
   Output: Typing interface with started timer
   Description: The test member runs the input command lpm, and verifies that the user is able to start ~~and stop~~ the timer by inputting a key ~~and using the escape button respectively.~~

4. **test-TE4**: Exit
   Type: Functional, Manual

Initial State: lpm session running but no snippet in session ~~Newly started terminal session.~~
Input: lpm
Output: Terminal screen exited from lpm ~~Typing interface~~
Description: The test member verifies that the user is able to quit the program using the escape key (assuming the timer has not been started or if it has, the user has stopped it).

5. **test-TE8**: Stop
Type: Functional, Manual
Initial State: Terminal session with lpm installed and lpm snippet session started with timer activated
Input: Escape key
Output: Typing interface with timer ended and cursor at the beginning
Description: The test member runs the input command lpm and begins the timer. Test user verifies they stop the timer by using the escape button

Editor Information

1. **test-TE5**: Typing Interface Link
Type: Functional, Manual
Initial State: Terminal session with lpm installed ~~Newly started terminal session.~~
Input: lpm, then the top link is clicked.
Output: Typing interface, as well as a link to the hyperlink opened up in the user's browser
Description: The test member runs the input command lpm and verifies the given hyperlink references the source code for the snippet, by clicking on the hyperlink ~~verifies that for the given code snippet the correct author, title, and time is displayed.~~

2. **test-TE6**: Keystroke Information
Type: Functional, Manual
Initial State: lpm code snippet in session ~~Newly started terminal session.~~
Input: A. Key matching next character on screen, B. Key not matching next character on screen
Output: A. Character in CORRECT_COLOR B. Character in INCORRECT_COLOR, All other characters in TEXT_COLOR ~~Typing interface~~
Description: The test member runs the input command lpm, and types in a mix of correct and incorrect characters. The test member should verify that incorrect inputs should be converted to INCORRECT_COLOR, and correct inputs should be converted to CORRECT_COLOR. Characters that have not been passed yet should be TEXT_COLOR.

3. **test-TE7**: Typing Interface Statistics
Type: Functional, Manual

Initial State: Terminal session with lpm installed ~~Newly started terminal session.~~
Input: lpm, then the user presses a character
Output: Typing interface, with stats information showing
Description: The test member runs the input command lpm and verifies that the code snippet accuracy, lpm, cpm and wpm are present on the screen.

4. **test-TE9**: Keystroke Information Correction
   Type: Functional, Manual
   Initial State: lpm session with previously typed characters
   Input: Backspace key
   Output: Character returned to TEXT_COLOR and cursor moved back one spot
   Description: The test member runs the input command lpm, and types in a mix of correct and incorrect characters. The test member should verify that upon pushing backspace, the text color of the last character is returned to TEXT_COLOR and cursor is moved back.

3.1.3 Code Snippets

Code snippet lengths of valid character and line length

1. **test-CS1**: Snippet Line Length Test
   Type: Functional, Automated
   Initial State: Snippets object is fully loaded with all lpm code snippets from pickle
   Input: Snippets object loaded with every single Snippet.
   Output: Pass or assertion error if any snippet length is larger than MAX_LENGTH
   Description: This test will go through every Snippet ~~code snippet~~ in the Snippets object ~~data base~~ and will output if any snippet is larger than MAX_LENGTH.

2. **test-CS2:** Snippet Character Length Test
   Type: Functional, Automated
   Initial State: Snippets object is fully loaded with all lpm code snippets from pickle
   Input: Snippets object loaded with every single Snippet.
   Output: Pass or assertion error if any snippet length is larger than MAX_COLS
   Description: This test will go through every Snippet ~~code snippet~~ in the Snippets object ~~data base~~ and will output if any snippet is larger than MAX_COLS.

Sufficient Code snippets available in Python, Java and Javascript

1. **test-CS3**: Snippets Per Language
   Type: Functional, Automated
   Initial State: Snippets object is fully loaded with all lpm code snippets from pickle
   Input: Snippets object loaded with every single Snippet.
   Output: Pass or assertion error if any language has less than MIN_SNIPPETS snippets

Description: This test will go through every Snippet ~~code snippet~~ in the Snippets object ~~data base~~ and will output if Python, Java or Javascript have less than MIN_SNIPPETS snippets.

3.1.4 Statistics

Track statistics per individual code-snippet

1. **test-S1**: Test lines per minute speed
   Type: Functional, Dynamic, Automated
   Initial State: lpm environment properly setup, Empty stats History
   Input: Type one line in 10 seconsd, Type 2 lines in 10 seconds, Type zero lines in 10 seconds ~~No typing~~
   Output: 6 lines/min, 12 lines/min, 0 lines/min
   Description: Will test if the statistics by providing an input for typing an individual code, and checking if the lines/min statistics match the the expected duration.

2. **test-S2**: Test characters per minute speed
   Type: Functional, Dynamic, Automated
   Initial State: lpm environment properly setup, Empty stats History
   Input: Type 20 characters 5 seconds, Type zero characters in 10 seconds ~~No typing~~
   Output: 240 char/min, 0 char/min
   Description: Will test if the statistics by providing an input for typing an individual code, and checking if the char/min statistics match the the expected duration.

3. **test-S3**: Test words per minute speed
   Type: Functional, Dynamic, Automated
   Initial State: lpm environment properly setup, Empty stats History
   Input: Type 10 words in 10 seconds, Type zero lines in 10 seconds ~~No typing~~
   Output: 60 words/min, 0 words/min
   Description: Will test if the statistics by providing an input for typing an individual code, and checking if the words/min statistics match the the expected duration.

4. **test-S4**: Test errors rate
   Type: Functional, Dynamic, Automated
   Initial State: lpm environment properly setup, Empty stats History
   Input: Match code statement perfectly, Make one char mistake when typing a 100 char line, Make a mistake on every character
   Output: 100%, 99%, 0%
   Description: Will test if the statistics by providing imperfect input, and checking the output accuracy

~~**Track statistics per typing session**~~

Track statistics per program lifetime

1. **test-S5: Test lines per minute lifetime speed**
   Type: Functional, Dynamic, Automated
   Initial State: lpm environment properly setup, clean slate with newly installed program
   Input: Type 3 lines in 10 seconds, close session, type 4 lines in 10 seconds
   Output: 21 lines per minute
   Description: Will test if the statistics by providing an input for typing multiple code snippets throughout multiple sessions, and checking if the lines/min statistics match the the expected duration.

3.2 Tests for Nonfunctional Requirements

3.2.1 Look and Feel

The typing interface shall respond to a user input within 10ms.

1. **test-LF1: User input latency $\leq$ 10ms testing**
   Type: Manual
   Initial State: lpm is opened and a code snippet is loaded.
   Input: User will type out the code snippet
   Output: The game should perform all functions properly with a max of 10ms latency.
   Description: This test will be performed through a third party (likely peers) who will type through the code snippet and comment on the responsiveness of the cli. Given that this is a nonfunctional requirement, the 10ms is arbitrary by design, as what really matters is how the user feels using the cli to type.

The user interface should be visible in both light and dark terminal backgrounds.

1. **test-LF2: Light theme test**
   Type: Manual
   Initial State: Command line is open
   Input: The user launches lpm in light mode
   Output: The lpm package opens in light mode
   Description: The test is the user testing to ensure light mode works as expected.

2. **test-LF3: Dark theme test**
   Type: Manual
   Initial State: Command line is open
   Input: The user launches lpm in dark mode
   Output: The lpm package opens in dark mode
   Description: The test is the user testing to ensure dark mode works as expected.

The provided theme shall be easy on the eyes and follow the WCAG AA or AAA specification in terms of colour choice.

1. **test-LF4: WCAG AA testing**
   Type: Manual
   Initial State: lpm is opened and a few characters are typed, such that all potential colours are displayed (i.e. some correct characters as well as some incorrect characters.)
   Input: The hex value of all the colours for each theme of the lpm package
   Output: A WCAG number above 4.5
   Description: The colours of lpm will be extracted using a windows tool to extract hex values, and input into a WCAG AA contrast checker to see if it complies or not. The WCAG number gives information about the contrast ratio of the colours. Above 4.5 indicates WCAG AA compliant.

The code snippets chosen should be diverse, and representative of the languages' syntax

1. **test-LF5: Code snippet diversity testing**
   Type: Manual
   Initial State: lpm is loaded
   Input: The user will use skipping functions to view multiple code snippets.
   Output: The user will have seen various code snippets and be able to comment on their diversity / uniqueness.
   Description: The purpose of this test is to ensure the tool is useful for our users, as repeating the same type of text snippets will serve no real purpose, as programming is not typing the same thing over and over again every time. This qualitative analysis will help us determine that we have enough diversity in our code snippet library, or recognize that we must expand our library to provide the user with relevant benefits.

The entirety of the user interface should fit within a terminal window sized MAX_COLS (width) x MAX_LINES (length) or greater ~~640x480 pixels or larger, and scale up according on the current terminal window size.~~

1. **test-LF6: lpm resoultion testing**
   Type: Manual
   Initial State: lpm is loaded in a terminal of size MAX_COLS (width) x MAX_LINES ~~640x480~~
   Input: The user will type a code snippet
   Output: The user will have seen the entirety of the lpm screen, including their code snippet and statistics. Through this process, they will be able to inform us if the package was usable in a window with a resolution of MAX_COLS (width) x MAX_LINES ~~640x480~~
   Description: The purpose of this test is to ensure the tool is useful at small resolutions, and also set a cap for the minimum resolution we officially support. If the user is able to recieve a full-featured experience at this level, we have satisfied the

NFR.

3.2.2 Usability and Humanity

The system's typing interface, as well as its cursor indicator, should be intuitive.
The system shall be easy to use for anyone with basic knowledge of the console.
The instructions will be easily comprehensible by anyone with basic understanding of English.

1. **test-UH1: General use testing**
   Type: Manual
   Initial State: Command line is open
   Input: User will open lpm and interact with the software
   Output: The user's overall impressions on the usability, typing interface, and comprehensibility of the lpm package
   Description: The three associated NFRs can all be tackled in one large manual user test with (potentially) multiple users all testing the package. In the process of testing, the users are able to give relevant qualitative feedback on their experience and the usability of the package.

The user will only require the keys on a typical 60% keyboard to correctly type all given code.

1. **test-UH2: 60% keyboard testing**
   Type: Manual
   Initial State: lpm is opened and a code snippet is loaded. The user is using a 60% keyboard
   Input: The user types through two code snippets.
   Output: The user is either successful or unsuccessful in typing the required code snippets
   Description: The test ensures that despite the limited keys of a 60% keyboard the lpm package is completely usable.

3.2.3 Performance

When the user loads the package, excluding the first time loading, the time it takes for the package to be ready to accept user input shall not exceed 1 second.

1. **test-PF1**
   Type: Manual
   Initial State: Command line is open, lpm package previous loaded
   Input: lpm command is entered
   Output: the lpm package is interactive

Description: This is run by a user running the lpm command and testing how long until the package is ready to output. While this could be a dynamic test through use of time libraries, we believe it is easier to implement by manual testing, and slightly more practical too - if a system has changes but they have not propagated to the user's view, it does not matter if technically the program can accept input.

Application should be available 99.999% (5 nines) of the time. This translates to 5.26 minutes of downtime in a given year, afforded by user updates of python or lpm, as well as unforeseen circumstances in the CI/CD Pipeline.

1. **test-PF2**
   Type: Manual
   Initial State: PyPI page opened
   Input: python package name, 'lpm'
   Output: Uptime based on pypi statistics and versioning
   Description: This manual test compares any failed pushed versions with the amount of time they are up. Due to CI/CD testing, uptime should actually be 100%, given that no corrupted or broken versions of lpm should ever be uploaded. The buffer time is left in case of some unforseen circumstances, or if PyPI goes down throughout the year, meaning our package is unable to be downloaded.

3.2.4 Operational and Environmental

The system shall work on Python 3.6 and up ~~Python 2 and Python 3~~.

1. **test-OE1**
   Type: Manual
   Initial State: Python 3.6 or above, lpm repository installed
   Input: run lpm
   Output: Package is successfully run on multiple versions of Python.
   Description: A test suite meant for Python 3.6 + ~~both Python 2 and 3~~ will be created and run. Any errors in output or exceptions caught.

The system shall work on Linux, macOS, and Windows operating systems.

1. **test-OE2**
   Type: Manual
   Initial State: lpm repository installed
   Input: run lpm, run CI/CD test suite
   Output: lpm can be installed and run on all 3 platforms.
   Description: This test suite ensures that our package is platform-agnostic, assuming the platform can have a valid version of Python ~~2 or~~ 3 and Pip installed on their platform.

In terms of computer specs, the package shall run on any computer that is able to run ~~Python 2 and~~ Python 3 based on their respective minimum requirements (i.e. if running on Python 3, the user's computer shall at least have Python 3's minimum requirements to be supported officially by the lpm package).

1. **test-OE3**
   Type: Manual
   Initial State: A third party who uses a laptop that is 10 years old does not currently have lpm installed opens their terminal.
   Input: The third party types pip install lpm, runs lpm, then types out one code snippet
   Output: Qualitative information on the performance of lpm, and information about whether it matched their standards.
   Description: Given our goal to be compatible with many different platforms, this test is essential to ensure lpm is working as expected, even on platforms over a decade old.

Application should be installable via the pip package manager.

1. **test-OE4**
   Type: Manual
   Initial State: Environment without lpm package installed
   Input: pip install –upgrade lpm
   Output: lpm package successfully updated.
   Description: This is a simple automated test that will be run in our testing pipeline, ensuring that our package is easily updated through use of pip.

3.2.5 Maintainability and Support

Application should be easy to update (via pip)

1. **test-MS1**
   Type: Manual
   Initial State: Environment with non-current version of lpm installed
   Input: Changes are committed, and a pull request is made for an issue.
   Output: Pull request is accepted, and changes are merged into the master branch. The lpm package now has the changes
   Description: This test ensures that the package is easy to update for users.

It should be easy to add additional code snippets.

1. **test-MS2**
   Type: Manual
   Initial State: lpm installed
   Input: New code snippet can be added with a permalink ~~A new code snippet is added to the snippets file and the file is saved~~. A pull request is made requesting

addition of the snippet.
Output: The code snippet is added to the lpm repository and shipped to users in the next release of lpm. Users will get new snippets when they reset snippets
Description: This is an essential test to have, because part of the success of the project we are building ours off of, wpm, is that other people can contribute texts to the package. We want to ensure this process is as easy as possible for developers, as it ensures that we can have people feel more empowered to add to the package.

3.2.6 Security

External systems shall not have access to the system.

1. **test-SC1**
   Type: Manual
   Initial State: lpm is installed on the system
   Input: A third party tries to access the system without access to it.
   Output: The party is unable to access it since there are 0 states in which someone external can access
   Description: This test is pretty straightforward, there is not much to say - the system by design is inaccessible by external systems due to PyPI's way of managing packages as well as the fact that our system does not rely on the internet after it is installed.

3.2.7 Cultural

Code snippets that include harmful, vulgar, controversial, political, or offensive content shall not be displayed in lpm

1. **test-CT1**
   Type: Manual
   Initial State: lpm is installed and running.
   Input: A third party will page through all of the quotes in the lpm tool.
   Output: The third party will inform us if they find any of the snippets harmful, vulgar, controversial, political, or offensive, and if so, corrective measures will be taken in the form of removing said quotes from the document.
   Description: This is an important test to ensure that our code snippets are not offending our users or negatively impacting them in any way. This is significant since we want to create a tool that is welcoming to all.

3.2.8 Legal

Code snippets shall have a valid open source license or the developers have given explicit permission to be displayed in lpm

1. **test-LG1**
   Type: Manual
   Initial State: lpm repository

Input: A code snippet is added to the repository with with link to code snippet ~~reference to the license it uses in a comment.~~
Output: The code snippet will provide a link to a source repository. If the link is a part of a repository with a valid open source licence, or lpm has explicit permissions from the developers, it will be accepted to the code base.
Description: This is an important addition to ensure that we are not legally responsible for the display and /or modification of any code snippets developers have uploaded.

3.3 Traceability Between Test Cases and Requirements

Traceability Matrix	
Functional Requirement #	Test ID
FR1	test-CLI1
FR2	test-CLI1
FR3	test-CLI2
FR4	test-CLI3
FR5	test-CLI4
FR6	test-CLI5
FR7	test-TE1, test-CLI2
FR8	test-TE1
FR9	test-TE5
~~FR10~~	~~test-TE5~~
FR11	test-TE7
FR12	test-TE2
FR13	test-TE2
FR14	test-TE3
FR15	test-TE8
FR16	test-TE6
FR17	test-TE6
FR18	test-TE6
FR19	test-TE6, test-TE9
FR20	test-TE4
FR21	test-CS1
FR22	test-CS2
FR23	test-CS3
FR24	test-CS3
FR25	test-S1, test-S5 ~~test-S6~~
FR26	test-S3, test-S5 ~~test-S6~~
FR27	test-S2, test-S5 ~~test-S6~~
FR28	test-S4, test-S5 ~~test-S6~~

Table 2: Functional Requirements Traceability Matrix

Non-Functional Requirements Traceability Matrix	
Non-Functional Requirement #	Test ID
NFR1	test-LF1
NFR2	test-LF2, test-LF3
NFR3	test-LF4
NFR4	test-LF5
NFR5	test-LF6
NFR6	test-UH1
NFR7	test-UH1
NFR8	test-UH1
NFR9	test-UH2
NFR10	test-PF1
NFR11	test-PF2
NFR12	test-OE1
NFR13	test-OE2
NFR14	test-OE3
NFR15	test-OE4
NFR16	test-MS1
NFR17	test-MS2
NFR18	test-SC1
NFR19	test-CT1
NFR20	test-LG1

Table 3: Non-Functional Requirements Traceability Matrix

4 Tests for Proof of Concept

The tests for the lpm proof of concept will be similar to the overall test plan, but will be largely focused on functional requirements, rather than non-functional requirements.

4.1 Command Line Interface

For the proof of concept, the tests for the Command Line Interface are equivalent to the tests listed in section 3.1.1.

4.2 Typing Editor

For the proof of concept, the tests for the Typing Editor are equivalent to the tests listed in section 3.1.2.

4.3 Code snippets

To test the code snippets, rather than running the tests on the entire test database, a test will be run on every snippet added.

1. **test-poc1**: POC Snippet Line Length Test
   Type: Functional, Static
   Initial State: Code snippets stored in working lpm program

Input: a. Code snippet of valid length, b. Code snipped of length over MAX_LINES ~~MAX_LINE_LENGTH~~
Output: a. Pass, b. ExceededLineLengthException
Description: This test will take in a code snippet and will output if the snippet throws an ExceededLineLengthException.

2. **test-poc2**: POC Snippet Character Length Test
Type: Functional, Static
Initial State: Code snippets stored in working lpm program
Input: Code snippet of valid length, Code snipped of length over MAX_COL ~~MAX_CHARACTER_LENGTH~~
Output: a. Pass, b. ExceededCharLengthException
Description: This test will take in a code snippet and will output if the snippet throws an ExceededCharLengthException.

4.4 Statistics

For the proof of concept, the tests for the Command Line Interface are equivalent to the tests listen in section 3.1.4.

4.5 Non-functional Requirements

Non-functional requirements will not be explicitly tested for in this proof of concept, as they are for the most part overall system performance requirements.

5 Comparison to Existing Implementation

Since we have access to the completely working wpm package, we can use that to better test and validate that our package, lpm, is working as expected. Specifically, we can test specific functionalities in both lpm and wpm to see if the system functions similarly.

Given that we provide metrics for lines per minute, and wpm provides words and characters per minute, we can test if a user types similarly on both systems by using the below formula to convert our lpm metric to the cpm (characters per minute) metric given by wpm:

$cpm = 88 \times (lpm/2) \times 1.3$

This equation represents:

- 88 characters per line of code maximum, multiplied by:
- the lines per minute outputted by our program, divided by:
- 2 as many lines in code are half-lines or just brackets

- 1.3 is multiplied across the entire equation as a rough estimate of how much slower someone types when typing code, due to often unfamiliar brackets and other symbols used while programming.

Combined, this equation gives a rough conversion between lpm and cpm from the wpm package, to ensure that our packages have similar performance. These tests will be manually performed by the same third party to ensure consistency in results.

Furthermore, we can use wpm as a qualitative test to ensure that all the features it has are somehow implemented in lpm. For example, wpm has options to choose the theme of the package, and lpm will be tested against those options to see if they exist as well.

6 Unit Testing Plan

Unit testing will be conducted using pytest and code coverage will be tracked using coverage.

6.1 Unit testing of internal functions

Any tests that were specified as type "automated" must be included in the unit test suite. For example, the FRs in the statistics subsection of the SRS will all be unit tested, as specified in section 4 of this document. will all be unit tested. Each unit test will contain a sufficient and diverse range of input/output examples that should cover both common inputs, edge case inputs, and invalid inputs that cause exceptions.

6.2 Code Coverage

As many of our tests are manual, it is important that we are able to provide reports and document the results as best as possible. As part of this process, we will be running code coverage reports for our manual test cases. This allows us to get a better idea of where we might want to focus our testing. If we find that a particular section of the code is not being run during our manual tests, this indicates that we may need to revise our test plan to include these sections.

In addition, we will also be generating code coverage reports for our unit tests.

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.

Constant	Value	Definition
MAX_LINES ~~MAX_LINE_LENGTH~~	20	Longest allowed length of code snippet, with respect to number of lines
MAX_COL ~~MAX_CHARACTER_LENGTH~~	80	Longest character length of any line of a code snippet
MIN_NUM_SNIPPETS	10	Minimum of code snippets required for every required language
TEXT_COLOR	Terminal Color [252, 235] ~~#FFFFFF~~	Main text colour for UI interface
CORRECT_COLOR	Terminal Color [243, 235] ~~#00BFFFF~~	Default text colour for correctly typed characters in UI interface
INCORRECT_COLOR	Terminal Color [9, 88]~~#F9BF3B~~	Default text colour for incorrectly typed characters in UI interface

Table 4: Symbolic parameters for lpm program

7.2 Usability Survey Questions

A usability survey may be conducted to ensure that the software product achieves the desired non-functional requirements such as usability and look & feel requirements.

1. Is it easy to install and start-up lpm?
2. Does the lpm user interface integrate well with your current terminal aesthetic?
3. Are the code snippets representative of your regular programming line habits?
4. On a scale from 1-10 how would you rate lpm in comparison to other typing-speed tests?
5. Are there any ambiguous visuals or instructions when using the software product?