# Advanced Health & Care Technical Test

**WORKING DOCUMENT: IN PROGRESS**

## My Approach - Jeremy Pridmore

Whilst listening to the verbal brief, I made notes of the key features, the inputs and outputs and some of the soft requirements such as how it is to be tested, memory, speed etc. I noted the technical platform, i.e. to be delivered as a console application and noted that the goal is to focus on writing beautifully structured code and not necessarily how good the output looks. David said that the full details of the test would be delivered via email to me. We agreed a deadline for delivery.

I brain stormed the best approach to kick start this project and the process that would best demonstrate my skills.

## Brain Storm

This is my initial Envisioning exercise. A brain storm of how I would tackle this particular mini-project.

* Treat as client. New client in myhours and track time.
* Present time spent on project and task break down.
* TFS vs GIT - TFS - choose Scrum process template vs David saying that moving over to GIT - so create a new GitHub repo.
* Need to look at Kaban process if using GIT?
* Use NUnit testing - create new test project. Which runner?
* Impact on memory key requirement - use ANTs Memory profiler? Most significant impact will be on the data structure and search algos.
* Also speed of execution is key requirement, so timing framework to compare performance of different methods?
* Need to come up with my own solution rather than looking at rehashes of other peoples solutions.
* Need to consider software design patterns - good OO design - perhaps consider outlining solution in UML behavioural/structural diagrams.
* Consider BDD - scenarios to describe behaviour of software. Although, David suggested they are more TDD.
* Need to make sure I evaluate performance and describe how I approached the development - using this document!
* Consider using Waffle.io to track project so that David has access to my tasks.
* Need to focus on just the requirements that are listed and not add any other superfluous features.
* No time for diagramming - stick to verbal descriptions of requirements and tests.
* NOTE: I am purposefully resisting the temptation to go and look up the algorithm online, instead attempting to demonstrate my analytical skills by breaking down the problem using my own knowledge of algorithm design and data structures.

## Questions

These are a set of questions that are shared with the client. This is a Q&A section that is visible to the client and maintained by both client and project team.

* Q: How does David want solution delivered/packaged?

A: Zip the executable, code and any supporting file and send to David’s personal email address.

* Q: Any performance metrics need to meet?

A: <unanswered>

* Q: Share repository now so that we can discuss if required?

A: Yes - links sent to David.

## Process

My thoughts on the process I will follow for this mini project.

* Research/Investigation.
* Adhering to agile tdd process
  1. Stakeholder participation
  2. Initial requirements envisioning and research.
  3. Writing Acceptance Tests to fulfil requirements.
  4. Just in time modelling for each test.
  5. Planning iterations and releases.
  6. Mapping requirements and tests onto a test matrix.
  7. Building the backlog.
  8. Implementing tests and code/mocks/stubs that is just barely enough.

## Participation

In order to involve all stakeholders in the project, I would ensure that they have visibility of key deliverables early on and ensure that they are actively involved in their continual evaluation. Of course, on a short term project such as this, it is more difficult to get full engagement.

* Delivered links to github and pivotaltracker.

## Requirements Envisioning

Although this is a very small set of requirements, the main requirement can still be described in terms of a high-level feature and scenario.

* Mapping out high level requirement(s)
  1. **Feature**: Search for Shortest List of Words
     + **In order to** obtain a file containing the shortest possible list of words starting at a Start Word and ending at an End Word where each word is found in a given dictionary and each subsequent word has only one letter which is different from the set of letters in the proceeding word.
     + **As a** User
     + **I want to** execute a Program with a Simple UI, type the name of a dictionary file, type Word A, type Word B, type the name of the results file, start the search and be informed that the results are ready to be viewed in the results file.
  2. **Scenario** -
     + **Given** the name of a text file containing a dictionary of words with each word listed on a separate line

**and** a Start Word

**and** an End Word

**and** assuming both words are present in the dictionary

**and** the name of a results file

* + - **When** I execute the program
    - **Then** I want to be prompted to enter the name of the Dictionary file

**and** I want to be prompted to enter the Start Word

**and** I want to be prompted to enter End Word

**and** I want to be prompted to enter the name of the Results file

**Then** I want the program to generate the results

**and** I want to be informed that the results are ready

## Feature Breakdown

The following numbered list is a break-down of the high-level feature/scenario into lower level features/scenarios.

1. **Prompting for Parameters**
   1. Feature: Prompt for name of Dictionary file.
   2. Feature: Prompt for Start Word.
   3. Feature: Prompt for End Word.
   4. Feature: Prompt for name of Results file.
2. **Generating Results**
   1. ***Preparation***
      1. Feature: Read the list of words from the dictionary file.
      2. Feature: Load words into a traversable data structure.
      3. Feature: Whilst loading, find words that only have one letter which is different.
      4. Feature: Link words together that are only one letter different to form paths.
      5. Feature: Find the Start Word.
   2. ***Execution***
      1. Feature: Maintain one or more lists of found words which represent paths.
      2. Feature: Record the length of the path.
      3. Feature: Check if End Word has been found.
      4. Feature: Record when a path has been searched.
      5. Feature: Move forward along a path to a word that has not been previously visited.
      6. Feature: Stop searching if the length of the path is greater than the shortest found path.
      7. Feature: Backtrack to search alternative paths.
   3. ***End condition***
      1. Feature: Stop searching if all paths have been searched.
3. **Returning results**
   1. Feature: Create and/or open the results file.
   2. Feature: Write to the results file a list of words that form the shortest path.

## Assumptions

The following assumptions have been made. These are situations that could occur and, given the available facts, inferences that I have made regarding particular behaviour in those situations and my subsequent assumptions regarding what the program should do in those situations. I would normally run these assumptions past the client in order to resolve them into concrete requirements.

* **Situation:** The SoR ask for a program to be written that calls a procedure that takes four parameters. It does not state the source of those parameters.

**Inference:** Should the program prompt the user to type them, should they be supplied on the command line, should they be read from a configuration file?

**Assumption:** The program will read them from the command line as this offers the best interoperability.

* **Situation:** The SoR states that it is safe to assume that both StartWord and EndWord can be found in the dictionary.

**Inference:** The dictionary may not be read completely or corrupt.

**Assumption:** The program should still check that both StartWord and EndWord exist.

* **Situation:** When searching for paths, more than one path of the shortest length may exist.

**Inference:** Should the program output both paths or the first one if encountered?

**Assumption:** Just output the first one.

* **Situation:** The dictionary file may not exist, yet the SoR does not state what to do in this case.

**Inference:** The program will need a dictionary in order to continue.

**Assumption:** The program should output a simple message saying that the dictionary could not be found.

* **Situation:** The SoR does not state the format of the contents of the dictionary file, nor what to do if the file is empty.

**Inference:** Having inspected the contents of the sample dictionary file, each word is listed on a separate line.

**Assumption:** The program will be written to read each line and treat the contents of that line as a single word and discard and empty lines.

* **Situation:** The results file may not be able to be created and the program would not be able to output results. Also, a previous result file may already exist.

**Inference:** The SoR does not say what to do if the results file already exists or if the results file cannot be created.

**Assumption:** The name of the results file shouldn’t be changed, therefore any existing file will be overwritten and if the file cannot be created or the output fails then a simple message will be output. No exception will be raised.

## Traceability Matrix

For each requirement, a test is designed that will state the input parameters and expected results. The input parameters for each test will be grouped into sets: simple, complex inputs, edge case, null cases.

The following traceability matrix lists all the above features and maps one or more tests onto those features to ensure that each feature is implemented, tested and confirms to the specification.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Feature:  Test Case: | 1.1 | 1.2 | 1.3 | 1.4 | 2.1.1 | 2.1.2 | 2.1.3 | 2.1.4 | 2.1.5 | 2.2.1 | 2.2.2 | 2.2.3 | 2.2.4 | 2.2.5 | 2.2.6 | 2.2.7 | 2.3.1 | 3.1 | 3.2 |
| 1.1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1.2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1.3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1.4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.1.1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.1.2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.1.3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.1.4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.1.5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.2.1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.2.2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.2.3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.2.4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.2.5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.2.6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.2.7 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.3.1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3.1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3.2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

## Test Cases

The following are a simple

1. **Prompting for Parameters**
   1. Test: Check that the name of Dictionary file has been set.
   2. Test: Check that the Start Word has been set.
   3. Test: Check that the End Word has been set.
   4. Test: Check that the name of Results file has been set.
2. **Generating Results**
   1. ***Preparation***
      1. Test: Check that the list of words from the dictionary file has been read.
      2. Test: Check that the list of words has been loaded into a traversable data structure.
      3. Test: Check if two words have only have one letter which is different.
      4. Test: Test that words have been linked together
      5. Test: Check that linked words are only one letter different
      6. Test: Check that the linked words form a path.
      7. Test: Check that the Start Word was found.
   2. ***Execution***
      1. Test: Check list creation
      2. Test: Check list traversal
      3. Test: GOT TO HERE ound word one or more lists of found words which represent paths.
      4. Test: Record the length of the path.
      5. Test: Check if End Word has been found.
      6. Test: Record when a path has been searched.
      7. Test: Move forward along a path to a word that has not been previously visited.
      8. Test: Stop searching if the length of the path is greater than the shortest found path.
      9. Test: Backtrack to search alternative paths.
   3. ***End condition***
      1. Test: Stop searching if all paths have been searched.
3. **Returning results**
   1. Test: Create and/or open the results file.
   2. Test: Write to the results file a list of words that form the shortest path.