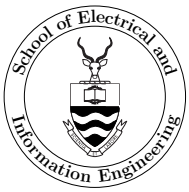


UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
School of Electrical and Information Engineering
Software Development II – Laboratory 2

OO and Unit Testing with `vector` and `string`

1 Introduction

The purpose of this laboratory is to more closely examine the functionality of the STL's `vector` and `string` classes. This is done within the context of developing an object-oriented system for text querying. Unit testing is also introduced, and forms a key aspect in specifying and testing the text query system. The laboratory outcomes are:

1. You understand and are able to use:
   (a) the `vector` and `string` classes
   (b) the `doctest` framework for writing unit tests
2. You are able to create small, simple objects which collaborate with one another to offer the required functionality.

As in the first lab, download the starter code from the course website (the Laboratories section) and extract it to a local folder. Within this folder run `git init` to initialise the Git repo. Now stage all of the untracked files and directories by typing: `git add .` Commit the files to the *master* branch with:

```
git commit -m "Initial commit"
```

Create a *solutions* branch on which to code the solutions for the exercises by typing:

```
git branch solutions
git checkout solutions
```

You are now in a position to start working on the lab exercises.

2 `vector`

The `vector` container is one of a number of containers provided by the STL. It stores elements in contiguous memory and offers an alternative for the array type provided by the C++ language.

Elements can be efficiently inserted and deleted at the back of the `vector`. However, it is expensive to insert or delete an element anywhere else. `vector` supports constant-time random access to individual elements in its sequence. This allows for efficient sorting and ordering algorithms, and `vector` is the most commonly used container.

Visit the following site for information on the `vector` class or use the supplied reference sheet: `http://www.fredosaurus.com/notes-cpp/stl-containers/vector/header-vector.html`

`vectors` are best accessed via iterators rather than by subscripts (as is done for arrays). There are a number of good reasons for this. Refer to `http://www.fredosaurus.com/notes-cpp/stl-containers/vector/iter-vector.html` for the details.

```
1  int main()
2  {
3      vector<int> vec;
4      cout << "vec: size: " << vec.size()
5           << " capacity: " << vec.capacity() << endl;
6
7      for(int i = 0; i < 24; i++) {
8          vec.push_back(i);
9          cout << "vec: size: " << vec.size()
10              << " capacity: " << vec.capacity() << endl;
11     }
12
13     return 0;
14 }
```

Listing 1: Vector Size and Capacity

Exercise 2.1

A vector is able to dynamically resize itself. Two important concepts related to this are "size" and "capacity". Examine the code in listing 1 and see if you can predict the output of lines 4 and 5 before running the program. Explain the difference between a vector's size and its capacity. Lastly, explain why the vector's capacity grows as it does. Give your answers as comments in the source code.

> Don't forget to commit after each and every exercise.

3 The `string` Class

The `string` class is actually a container class which holds a sequence of characters. `string` provides a number of member functions including `length`, `replace` and `find`. Additionally, it overloads certain operators. For example, the + operator can be used to concatenate strings. Certain STL algorithms can be used with strings, such as `getline` and `transform`.

A reference sheet for the string class can be downloaded from the course web site (taken from `http://www.fredosaurus.com/notes-cpp/strings/header-string.html`). A good alternative reference can be found at `http://www.cplusplus.com/reference/string/string/`.

4 Automated Unit Testing

Automated testing is essential when developing anything but the most trivial of software systems. In fact some software methodologies, such as XP (eXtreme Programming), insist on programmers writing tests prior to writing actual code. This is known as *test-driven development*. Two important reasons for *automating* testing are:

1. It may be extremely difficult to manually test a system that is distributed over many computers or a system that manipulates large amounts of data (for example, thousands of records in a database).

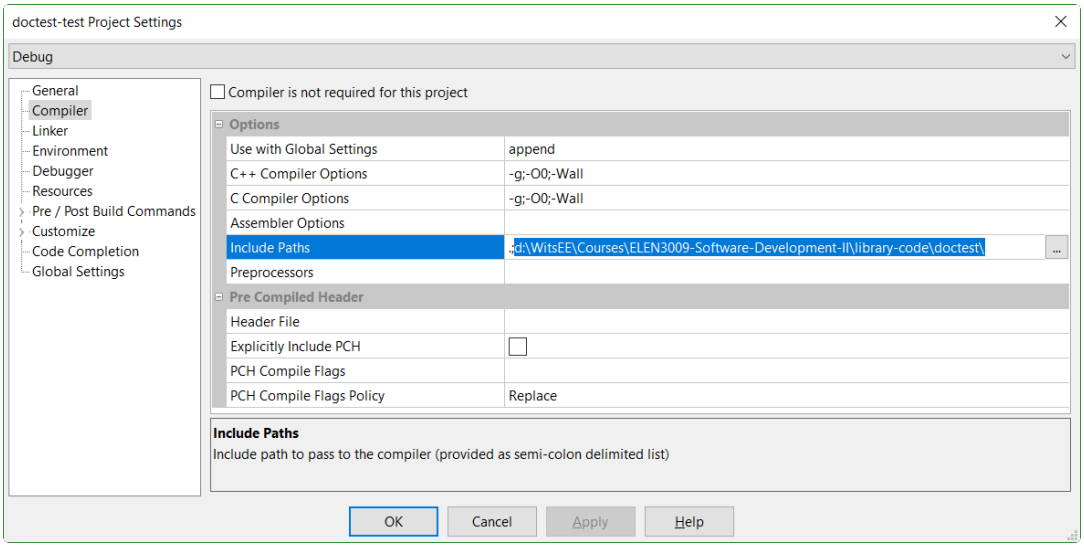


Figure 1: Compiler settings for `doctest`

2. We are often required to repeat the same tests on different versions of a software system. This is known as *regression* testing. Automated testing helps ensure that changes in the new version have not broken functionality in a previous version.

Testing at a fundamental level is known as *unit testing*. A unit in an object-oriented program is a class and unit testing attempts to uncover any defects that may exist in the class. This course utilises the `doctest` C++ testing framework. This framework is similar to other unit testing frameworks found in both Java and .Net environments. It offers fairly advanced testing functionality but in this course we will concentrate on using it to perform simple testing.

Exercise 4.1

`doctest` offers *assertions* for verifying code behaviour. These assertions are grouped into *test cases*. Read through the "simple example" section of the `doctest` tutorial in order to understand how to use the framework.

Download the `doctest` library from the Resources section of the website, and unzip it to a suitable location. Create a new *simple executable (g++)* project and add the file contained in the factorial directory of this lab to it.

Every project that uses the `doctest` library needs to include its header files. To do this, do the following: right-click on the project name and select Settings... |Compiler and supply the path to the `doctest` directory (which was initially in the zip file) . Refer to Figure 1.

Now you should be able to compile and run the tests. You will see that one fails. Correct the factorial function so that all the tests pass.

5 A Text Query System

The brief describing the system is as follows:

> Write a program that searches a text file for a user-specified search word and returns the line numbers of all instances of the word. The search must be case-insensitive and punctuation must be ignored. The system is not required to support queries for small words (less than 3 letters).

Reviewing the brief, and thinking about it from an OO perspective, it appears that there are a number of important concepts which can modelled as objects, including:

- A word, which has the responsibility of knowing what a word is (it contains no spaces and must have at least one letter) as well as being able to determine if it is equivalent to another word.
- A collection of words, or a line, which has the responsibility of knowing if it contains a particular word.
- A collection of lines, or a paragraph, which has the responsibility of knowing all of the line numbers on which a particular word occurs.
- Lastly, there is a file reader which is responsible for reading a text file into memory as a paragraph.

Figure 2 illustrates how these *objects* collaborate with each other to search for a particular word in a paragraph. There are some important things to note about this diagram. Firstly, it does not depict all the messages that will be sent by the objects in the program. It serves to highlight the interactions that take place when searching for a word. Secondly, the "alt" block indicates two alternative sequences that take place depending on whether the word is found or not. Lastly, the diagram shows that sequence diagrams are not that good at depicting for loops (although there is a better way which we will cover later) and complex flow of control. An activity diagram is a more suitable way of illustrating algorithms with complex control flow structures.

The relationship between these different *classes* can be expressed by the class diagram shown in Figure 3. We will also cover these diagrams in more detail later on in the course.

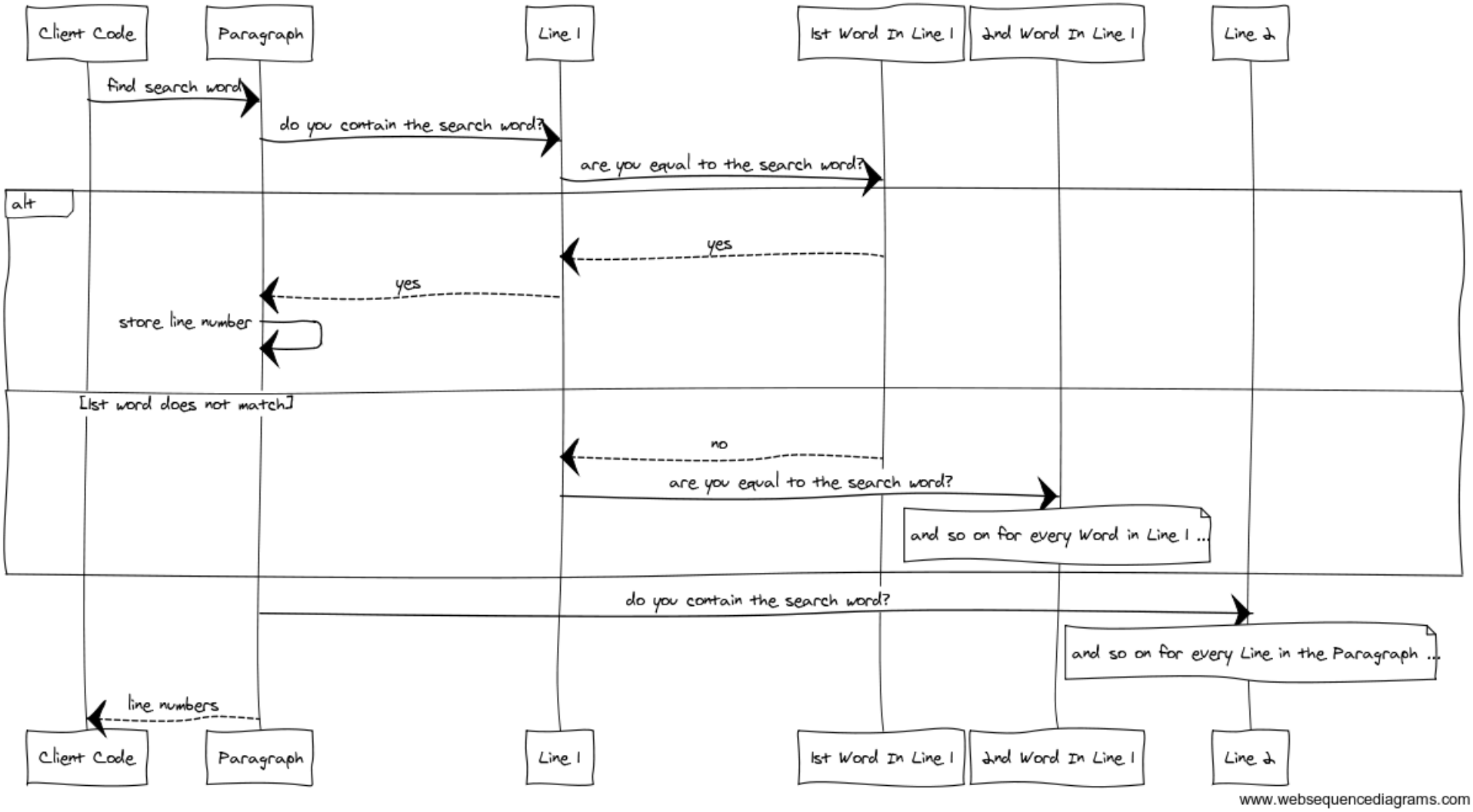


Figure 2: Sequence diagram for searching for a word

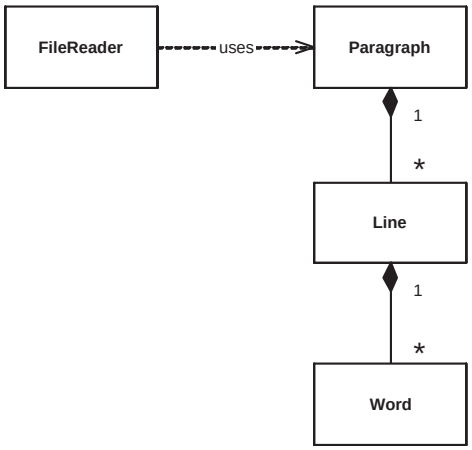


Figure 3: Classes in the Text Query Program

We will tackle these classes one at a time, starting with the lowest-level abstraction — a `Word`.

5.1 Modelling a Word

From the problem statement it seems sensible that a `Word` object must obey the following rules:

- it cannot contain any blank spaces; otherwise it will be two words
- it must contain at least one letter; otherwise it is not a word

We also need to be able to ask a `Word` object if it can be queried (words having less than 3 letters cannot be queried).

The two rules listed above can be enforced in the constructor for `Word`. If an attempt is made to construct an invalid `Word` the constructor can throw an *exception*. Throwing an exception prevents the object from being constructed by generating an error which, if not caught, will crash the program.

Exercise 5.1

Examine `Word.h` and `Word.cpp` and you will notice that certain functions have been declared but not implemented or only partially implemented. However, the *equivalence* or *is-equal-to operator* has been fully implemented. Here, we are overloading this operator to allow `Word` objects to be compared with one another, which is useful for testing, and in its own right.

Open up `TextQueryTests.cpp` which contains tests for the text query system. The first tests which are listed are those for the `Word` class. You will notice that most of the tests are commented out except for those which test the equivalence operator and a test on the constructor. Build and run the project and you should see these tests passing.

Uncomment the first test (`Case is ignored when comparing Words`) and then write all the code inside the `Word` class which is required to make it pass. Do this for each of the remaining tests for `Word`. Working in this fashion — uncommenting (or writing) one test, and then writing the code to make that test pass — breaks the task of writing a whole class up into smaller chunks and inspires confidence as you see each test passing. You will also have immediate feedback if you write code that happens to break functionality that was formerly working.

The last test for `Word` (`Word is not queryable if less than a specific size`) is for you to complete.

5.2 Modelling a Line

A `Line` can modelled as a collection of individual words. In order to support a text query we need to be able to ask a `Line` if it contains a particular word. `Line`'s public interface (see `Line.h`) captures this idea. Note, that `Line`'s constructor takes a `string` of text. This text string will be provided by `FileReader` in *production* code. In the *test* code we will simply give `Line` a text string to work with. To see how this is done review the tests for the `Line` class.

You might feel that it is more appropriate to form a Line by adding together individual `Words`, rather than passing in a string. The motivation for using a string is that we will be reading a file into memory (in `FileReader`), one line at a time, using the `getline` function (see the `string` class reference sheet). It is therefore convenient to be able to construct a `Line` directly from a `string`.

Exercise 5.2

As with the `Word` class, uncomment one test at a time and then write all the code that is required to make it pass. In doing this we are slowly building up a suite of unit tests for the text query system.

5.3 Modelling a Paragraph

A `Paragraph` can be modelled as a collection of `Lines`. We need the ability to add lines to a `Paragraph`, as well as, to query it for a particular `Word`. A `Paragraph`, will then be able to support the functionality required by the system.

Exercise 5.3

Review the declaration of the `Paragraph` class in order to fully understand the interface. The tests for `Paragraph` (in `TestQueryTests.cpp`) have been named but not implemented.

Your task is to implement *both* the tests and the code to make the tests pass (one at a time!). It is advisable to read the *Guide to Writing Unit Tests*, which is available on the course website, before doing this.

Remember, that you are writing *unit tests* which means that you need to test a `Paragraph` object in isolation from a `FileReader` object. Create fake paragraphs in your tests by simply adding lines to a `Paragraph` object.

The final test supplied in the test file is an *integration* test which you should run after you have completed the tests and code for `Paragraph`. This test tests that `Paragraph` and `FileReader` are working together correctly.

5.4 FileReader

The `FileReader` class can be used as supplied. No unit tests have been written against this class as it is difficult to observe its behaviour independently of `Paragraph`. More advanced testing techniques are needed to test `FileReader`. We may cover these later in the course.

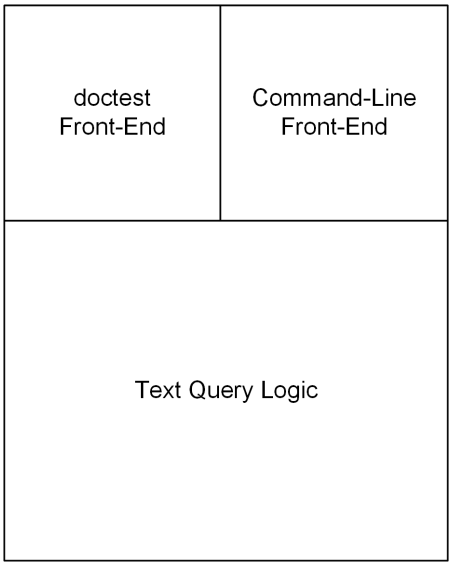


Figure 4: Different Front-Ends for the Text Query Logic

5.5 Writing the Text Query Program

After all this work we haven't actually written the text query program itself but we have thoroughly tested the objects needed to perform the program logic. Effectively, we have created a "test front-end" to the system. We now need to create the required command-line front-end so that we can actually use the system. The two front ends are illustrated in Figure 4.

Exercise 5.4

Write the program that performs the query, complete with user interface. The user interface should be similar to the following:

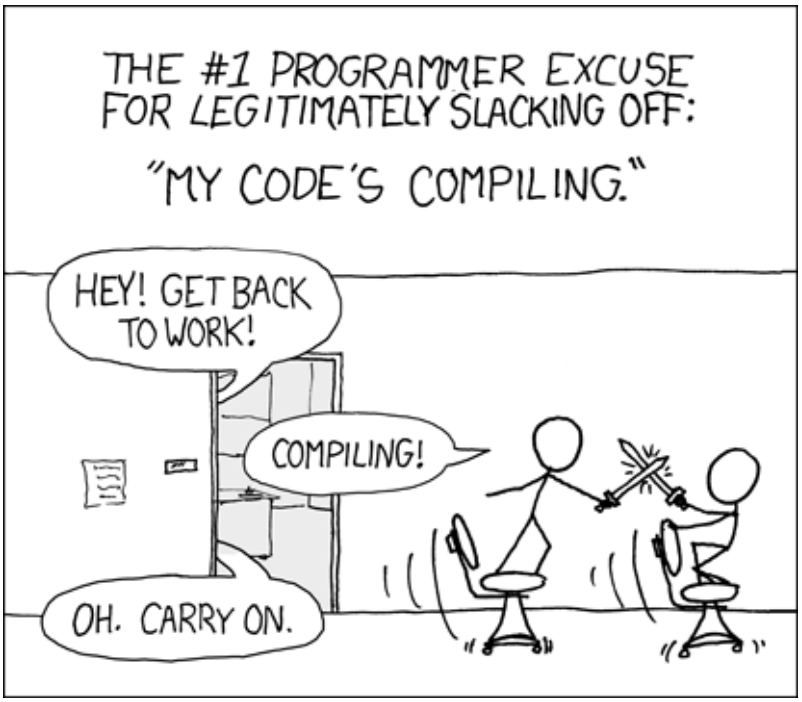
```
Please enter file name: alice.txt
Please enter a word to search for or "." to quit: Daddy

Word found: line 1
            line 4
            line 6

Please enter a word to search for or "." to quit: superstitious

Word not found
```

To do this create a console application project in your IDE and add to it all the source files that you have working on, except for the file containing the unit tests. Create a new source file/s to contain the `main` function and the UI logic.

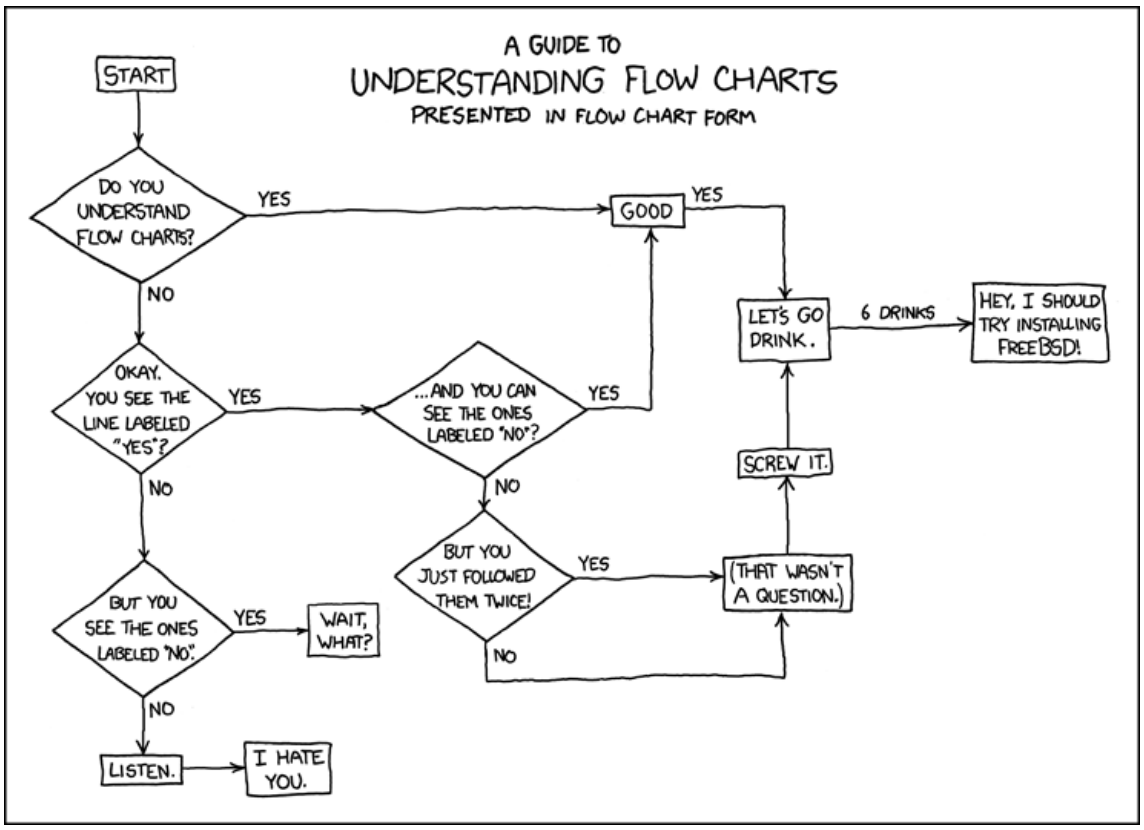
This demonstrates an important reason for separating the front-end or presentation layer from the application logic layer. Often, we have at least two different front-ends which use the same application logic — a test front-end and the actual program front-end.

Finally, how will your code handle a query for a word which appears twice on one line? Would your design have to change to support this properly? Provide your answers as comments in the source code.

Submit your lab via GitHub following the instructions in the Git/GitHub guide.



Source: `http://xkcd.com/303/`



Source: `http://xkcd.com/518/`