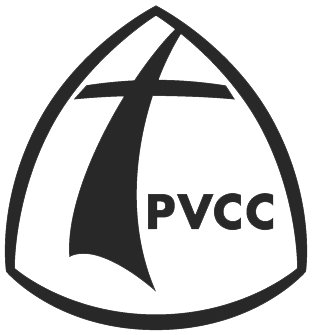
****

**NAME: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |
| --- |
| **Plenty Valley Christian College** |

|  |
| --- |
| Applied Computing: Software Development Unit 3 Area of Study 1 Software development: Programming School-assessed Coursework |

**2022 Programming on Interpreting teacher provided solutions requirements and designs, and apply a range of functions and techniques using programming language to develop and test working software modules for Outcome 1**

**Recommended writing time\*: 8 Periods**

**Total number of marks available: 84 marks**

**Module C**

**TASK BOOK**

|  |
| --- |
| **Conditions and restrictions**  • Students are permitted to bring into the room for this task: pens, pencils, highlighters, erasers, sharpeners and rulers.  • Students are NOT permitted to bring into the room for this task: blank sheets of paper and/or white out liquid/tape. No calculator is permitted in this task.  **Materials supplied**  • Question book of 14 pages.  **Instructions**  • Print your name in the space provided on the top of the front page.  • All written responses must be in English. |

|  |
| --- |
| **Students are expected to complete this task under teacher supervision in the allocated class time.** |

# VCE Applied Computing: Software Development

## Unit 3 Area of Study 1: Programming practice

On completion of this unit the student should be able to interpret teacher-provided solution requirements and designs, and apply a range of functions and techniques using a programming language to develop and test working software modules.

Students apply computational thinking skills when interpreting given solution requirements and designs, and when developing them into working modules.

**Task Structure**

The task is split into 4 parts of increasing complexity:

* Part 1 (Module A)– interpret requirements, build and test a basic solution using an approved programming language. Incorporate the use of appropriate data structures, control structures, data validation and programming features.
* Part 2 (Module B)– enhance solution to read from a CSV file
* Part 3 (Module C)– enhance solution to search a CSV file
* Part 4 (Module D)– enhance solution to search and sort a CSV file

\*Refer to marking rubrics in SEQTA for marking criteria / mark allocation

## Outcome brief

The final exam marks for a class is depicted below. As a programmer, you are tasked to write a program to assist the teacher in keying in the marks. The final product would resemble the following:

|  |  |  |  |
| --- | --- | --- | --- |
|  | English | Maths | PE |
| John | 43 | 78 | 89 |
| Damien | 23 | 89 | 90 |
| Mark | 66 | 76 | 56 |
| Joshua | 23 | 45 | 65 |
| Mary | 78 | 34 | 76 |
| Maria | 84 | 89 | 54 |

## General information

You must complete four tasks of increasing complexity by writing programs to implement the design provided. The designs in all cases are provided for you, however, keep in mind that you must:

* demonstrate the use of functions, with parameters and return values, where appropriate.
* include rich internal documentation that fully describes your logic and reasoning and implement code layout conventions (e.g. white space, indenting) for readability.
* use meaningful names for your project, data structures, functions and variables, adhering, where reasonable, to code conventions for the programming language.
* validate input data and user input.
* fully and formally test your programs, providing clear evidence of operation.

**Testing**

You should perform testing for each task. For Tasks 2, 3 and 4 this includes constructing appropriate test data based on the example.

A testing table, like the template provided in Appendix 1, should be provided as evidence.

## Task 3

**Searching a CSV File**

Using the same CSV file from Task 2, develop an efficient GUI to be used for a parent teacher interview meeting. The teacher searches for the student’s name and the data is displayed on the screen for each subject.

**This program should work as follows:**

STEP 1: Read in the CSV file *once only*. For each row in the file, store the values in a record with appropriate fields to match the columns. All the records should be kept in an array.

STEP 2: Prompt the user to enter the name of the student

STEP 3: Find the matching record in the list by searching for the name. The data for all subjects for the respective student should be displayed.

If a matching record is not found, display an appropriate response and return to STEP 2.

##### **Pseudocode**

This pseudocode describes the operation of the program (without validation added).

**Display** 'Loading file…'

**For each line in the csv file**

EnglishMarks ← first field from line

MathMarks ← second field from line

PEMarks ← second field from line

**Add** entry **to array** entries

**End For**

**Display** 'CSV file loaded into array'

**Display** "Let's find your entry."

**While we haven't found the matching entry in list entries**

nameToFind ← **Input** 'Enter student name: '

**Search** entries **for a matching entry with** nameToFind

**If we haven't found the matching entry in list** entries

**Display** "Sorry, there was no match. Let's try again."

**End if**

**End while**

**Display** 'Entry Found’

**Display EnglishMarks**

**Display MathMarks**

**Display PEMarks**

**Question / Answer (add more space as required)**

1. Explain how your code works.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Upload your code to github. Write the link to your github here. Note that your answers may not be revised after uploading:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Paste your design here:

\*Add more space as needed

|  |
| --- |
|  |

1. Paste your code here:

\*Add more space as needed

|  |
| --- |
|  |

## Appendix 1: Testing table template

Use the following testing table to check your module's functionality. The completed test table should be included with every submission.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Action** | **Expected result** | **Actual result** | **Pass/fail** | **Remedy** |
| Typed in name ’john’ into input box | John’s results presented in lblAnswer with message box saying ‘results found!’ | Maria’s results presented in lblAnswer with a message box saying ‘student not found’ | fail | Adjust what the file |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

\*Add mode lines to this table if needed

**Action**: some action, event or input to the system which should provoke a response. Typically, this uses a verb: ‘Click the select button’, ‘Select the third item from the drop down box’, ‘Focus a light on the light sensor’.

**Expected response**: this describes what the system *should* do in response to the action taken. It could start with the words ‘The system shall...’. An example might be ‘display the words “HELLO WORLD” in upper text box’. Be specific and definite.

**Actual response**: this describes what *actually* happened (or didn't happen). Try to gather as much information as possible here. ‘The text “HELLO WOR” was displayed and filled the whole text box’.

**Pass/fail**: if the actual result matches the expected result exactly, the test passes.

**Remedy**: if you can, describe what action needs to be taken to fix the problem. This is not always possible at the time of testing but can be added after the program has been fixed. e.g. ‘The size of the text box needs to be increased to accommodate longer sentences.’

## Failed test should be redone. A major change or redesign should cause all tests to be re-run as *regression* tests to ensure that no side-effects of a code change have affected other parts of the program.