Automated assessment tools for COMP61511: Software Engineering Concepts in Practice

A dissertation submitted to The University of Manchester for the degree of Master of Science in the Faculty of Engineering and Physical Sciences

2018

by

Marta Pancaldi

School of Computer Science

Contents

Word count: 23461

List of Figures

List of Tables

List of Code Snippets

List of Abbreviations

ASCII	American Standard Code for Information Interchange
CI	Continuous Integration
CPU	Central Processing Unit
CSS	Cascading Style Sheet
CSV	Comma-Separated Values
DOC, DOCX	Microsoft Word document
GNU	recursive acronym for GNU’s Not Unix
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LOC	Lines of code
PDF	Portable Document Format
stderr	standard error
stdin	standard input
stdout	standard output
TA	Teaching assistant
TXT	plain text file
Unix	originally UNICS (Uniplexed Information and Computing Service)
UoM	University of Manchester
UTF	Unicode Transformation Format
WC	word count

Abstract

Grading assignments is a lengthy and repetitive task for teachers. When it comes to programming exercises, the process becomes also troublesome and error-prone, as it is difficult for humans to assess a piece of code impartially and accurately: for example, if an instructor wanted to verify that a student programme behaves as expected, they would need to run dozens of tests, which is time-consuming and not a guarantor of the objectivity and correctness of the process.

In this work, we investigate the automated evaluation of programming assignments applied to *COMP61511*, the Software Engineering MSc course at the University of Manchester. In previous runs of the course, feedback for students came quite late due to the lack of working assessment tools: receiving prompt feedback about one's skills and understanding of specifications is essential in learning programming.

The tools we developed are used to run automated tests on a clone of WC – the Unix word counter –, which students are asked to implement as a project for *COMP61511*. Our auto-graders can execute the student's code and compare the results to the model programme, and perform other tests such as computing the code coverage. The instructor can customise the test cases and the number of marks each grading parameter should receive, then the assessment tools will calculate a grade and generate a feedback, which students will receive in a few hours after the deadline.

This project involves both pedagogical and technical aspects: first, we defined possible models of feedback and studied expected learning outcomes, then we implemented a set of tools based on our analysis that will perform an accurate evaluation of the student programming exercise, and should represent significant savings in time and effort for the instructor.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I wish to express my deepest gratitude to my supervisor, Dr Bijan Parsia, whom I initially learnt to know as an instructor during the COMP61511 course, for believing in my abilities and stimulating me with his enthusiasm. Being given the task to improve the very same course I had attended has been a valuable opportunity and a genuine honour. I hope my effort will facilitate the challenging tasks every teacher has to deal with in the years to come.

Thanks to the other professors and teaching assistants I had the chance to meet at the University of Manchester, in particular: Dr Daniel Dresner, Mr Robert Haines, Dr Caroline Jay, Dr Sandra Sampaio, Dr Liping Zhao.

Thanks to my fellow Computer Science students, for they always understand what you are going through, and my beloved flatmates, for enduring my endless hours spent typing.

Thanks to mum and dad, for supporting and loving me even though I am never at home lately.

Finally, thanks to the people I forgot, because a list of acknowledgements is just not enough.

Chapter 1

Introduction

1.1 Introduction

COMP61511: Software Engineering Concepts in Practice is the first module of the Software Engineering pathway, or theme, of the MSc in Advanced Computer Science taught by Dr Bijan Parsia at the University of Manchester. The course is structured in five weeks of lectures (20 hours) and laboratory classes (15 hours), and is worth 15 credits, out of a total of 180 credits for the entire degree. Previously named *Software Engineering Overview*, the course has been taught with the present structure since the academic year 2015/2016 and is currently in its third year.

The syllabus [1] includes the following topics, which also represent the main stages of software development:

- **Software engineering as a discipline**: the theory behind the practices, and why it is important in the activity of a software engineer.
- **Software development project management**: development planning, resource allocation, time and effort estimation.
- **Software design and architecture**: understanding a stakeholder's needs and requirements for a software product, defining the specifications and modelling the system.
- **Software construction and development methodologies**: implementation of the requirements, choice of programming languages and frameworks.
- **Quality assurance**: software testing and debugging, validation and verification, code review techniques, quality standards.

More specific topics that are addressed include software complexity, reverse and re-engineering, product internal and external qualities and professionalism in software engineering.

The study plan includes a practical section where students are guided towards the development of a complete programme in Python over the course of five weeks, following the techniques and notions acquired during the lectures.

The goal is to build a clone of the WC (short for word count) command line tool [2] found in Unix-like operating systems, starting from the most basic functionality up to a "one-to-one" instance, which ideally produces the same output of WC for every given input value. The principal specifications of the programme are listed week by week, but students are required to develop the tool through a process of reverse engineering [3] and black-box testing [4], that is to understand WC's behaviour and reproduce it by relying solely on input and output combinations.

This project focuses on various aspect of the COMP61511 coursework, in particular improving the assessment process of the programming assignments, as well as trying to fix some problems that emerged during the course.

1.2 Project objectives

The author of this project attended COMP61511 as student during the 2017/2018 academic year as part of her MSc study programme, and now the objective is to tackle a series of issues that were highlighted during her experience in the classroom and discussing with the course instructor afterwards. In particular, the major criticism that has been raised towards the course concerns the timing of the feedbacks that students received for their assignments, which were released several weeks after the course had been finished.

Of course, teachers have a long time at their disposal to assess a class' submissions and exams (at the University of Manchester, the process takes four weeks on average). However, especially in case of programming assignments, receiving a response about one's work in a relatively short time would represent a valuable help in understanding any mistakes made and improving one's performance for the following assignment. Also, grading assignments, particularly software projects and inside a large class, is often a long, time-consuming and repetitive task. Having the chance to simplify the process, and possibly reduce the amount of manual effort an instructor needs to make, would most likely represent a relief and a substantial decrease in the work.

The solution we present in this project draws on the existing literature about automated assessment of programming assignments, and related software systems. There are, in fact, several aspects of the WC clone that can be graded automatically: for example, the output correctness can be checked by comparing the counts computed by the student's programme with the one from the original tool, and a suitable algorithm can do this easily even for hundreds of output strings.

Of course the instructor needs to define a set of test cases that the programme can use and establish the weight all marking criteria have, so the contribution of a human grader is ineliminable. However, a machine can take the responsibility to execute the given tests, and could even be more efficient, precise and less error-prone by doing so.

The goal of this project is then to create a body of software tools that automate several aspects of code assessment, in order to reduce the workload associated with the marking process and provide students with prompt and meaningful feedback at the same time. This is why we have worked on a series of tools that will perform such tests automatically, and this report will describe how we accomplished this.

1.3 Project deliverables

The practical part of the project consists, as mentioned above, in a set of scripts that together will serve to arrange all student submissions, perform tests and assign grades to them. All scripts are also developed in Python. To give an overview of their functionality, they will:

- check the submissions were committed before the chosen deadline, or treat the late ones according to the school's policy;
- organise the submissions in case multiple attempts are allowed, so that only the latest is graded;
- perform structural (i.e. the submission follows the specifications, no prohibited external libraries were used,...) and correctness (i.e. the script behaves like WC does) tests on the student programmes. All test cases and marking parameters are defined in a configuration file that the instructor can customise as he deems appropriate;
- compute the marks for each parameter and store the results of all students;
- arrange the results and feedbacks for the students so they can be uploaded to the university grading system.

We also provided a set of tools for student use, like:

- submission samples containing all needed files named in the desired way, which their students can use as base to build their code;

- submission preparation scripts that perform a basic inspection of the student submission, such as checking for compilation errors or verifying that the files have been at least modified (without testing any aspect of the programme correctness).

As an additional content, we analysed the current architecture of the coursework and introduced some changes that could improve both the structural consistency of the coursework itself and relevance and coherence of the exercises with respect to the lecture topics.

1.4 Dissertation outline

This report is divided into the following sections:

Chapter 2 – Background

We review a series of existing works and software products that deal with different issues concerning testing and grading automation.

Chapter 3 – Research Methods

The functionality of WC is explained in details, together with the different steps that students are expected to follow to build a clone of the tool. Then we present the motivations behind the choice of such programme as coursework, and finally discuss the specific features of the tools we created and what goals each wants to achieve.

Chapter 4 – System Design

We describe the implementation of the automated graders and the impact they have on the coursework. This chapter contains the detailed functionality of all software tools developed for this project.

Chapter 5 – Testing and Evaluation

Here we discuss how the tools were tested, how the verification and validation processes were carried out and any limitations that we encountered.

Chapter 6 – Conclusion

Finally, this chapter summarises all the work done and describes what the future steps of the development of automated graders may be.

Chapter 2

Background

2.1 Chapter overview

This chapter focuses on the characteristics an automated grading tool should have, the benefits it provides compared to a manual evaluation of programming assignments and the disadvantages that it may bring, as a tool that performs automated tasks. An overview of the existing software graders is also presented, illustrating how they are employed in a classroom context.

2.2 Automated evaluation of software engineering assignments

The possibility of using computers to ease or even automate the process of grading software assignments has been largely investigated within the field of Computer Science Education [5]–[7].

In his works starting from 1991, Jackson [8][9] has introduced the possibility to grade student programming assignments using automated software, and described in details the features that an automated grader should have, including its advantages and limitations [10].

First, it is known that the activity of marking student software takes a great amount of time and effort from an instructor, and it is not guaranteed that no errors have been made in the process, since any human activity is potentially error-prone. However, there are recurring and repetitive tasks that may be assigned to a machine, and a computer may even be more accurate and efficient that a human being in performing these tasks. For example, a human instructor might take hours to visually compare the resulting output of dozens of student programmes, whereas a computer would need seconds to complete the same task if correctly instructed to do so. Although the human element can and should not be eliminated from the process, these are the main motivations behind conducting research in the field of automated evaluation, especially in an era where software engineering classes are becoming larger and larger.

The assessable aspects of a submission include the correctness (whether a programme executes as it is meant to do), the efficiency (e.g. the CPU time spent running the code), the complexity (e.g. McCabe's cyclomatic complexity metric [11]), the style (mainly whether the code follows the standards for its language) and the test coverage (how many statements were executed at least once by the tests).

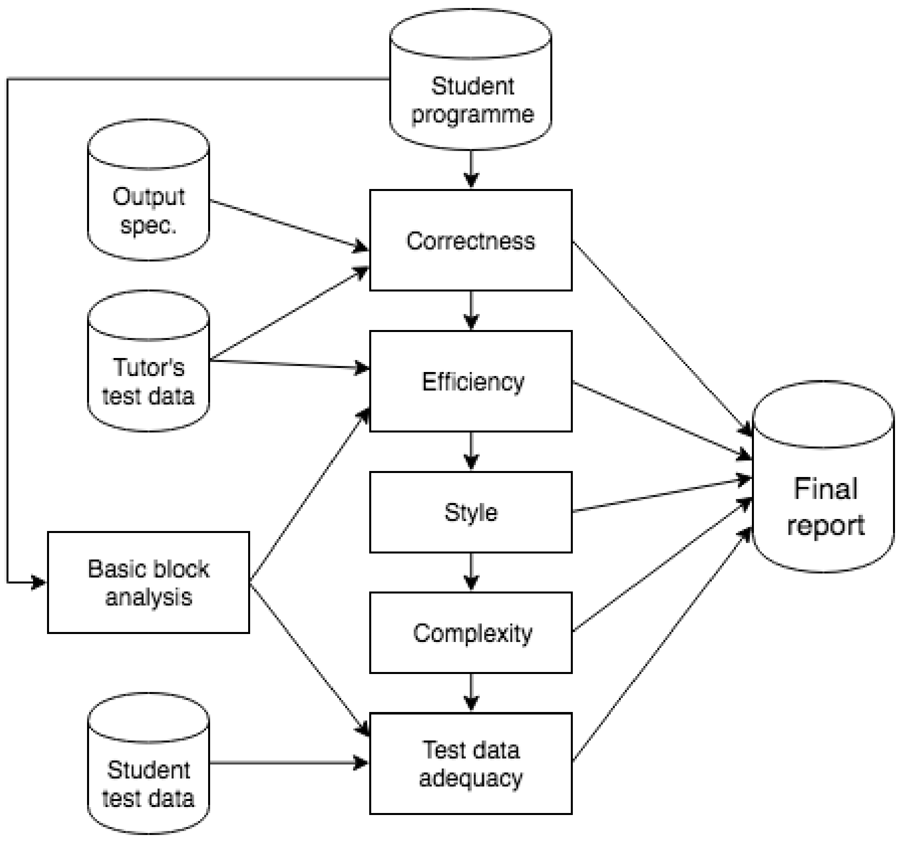


Figure 2.1: Diagram of the assessment process by Jackson.

A clear limitation to automating assessment of correctness, which is perhaps the most relevant indicator of whether a student performed adequately or not, is the need for a well-defined result or set of output values that can be compared with a model solution. A simple evaluation like *right / wrong* would not be explicative enough for most assessment metrics, so a scaling approach should be adopted instead.

For instance, consider a function that returns a list of ten attributes, of which only one appears wrong in the student's output: in cases like this, assigning a percentage of the maximum mark would be a more reasonable approach than giving no marks for a single incorrect portion of the result. The instructor has then to clarify what are the exact parameters to be assessed and what level of tolerance should be implemented, if any. At this point, the two general approaches are either the "absolute" and "relative" marking:

- the instructor can establish an “ideal” value of comparison to assess some parameter (e.g. to receive a mark, the line coverage must be at least 80%), or

- a programme model previously built by the instructor may be used as a comparison metric, and marks are then assigned according to how well the student’s programme performed against the model.

There should be a limit, however, to the “interference” that the assessment tool has on the student experience in building the programming assignment. Jackson specifically writes about a trade-off [10] between having an automated grader available and expecting student programmes that have no room for vagueness: in fact, to increase the level of autonomy of the grader and decrease the chances it assesses as incorrect some output that only slightly deviates from the expected, “ideal” one, it would be necessary to assign totally unambiguous specifications; however, in this case the risk is that students, knowing that an automated tool is being used, may focus too much on following literally, even “blindly”, the directives. They would not put their effort into analysing the programme requirements, understand any vague point and make a judgement according to their reasoning.

Since specifications are hardly ever unambiguous in a real developer’s routine, learning how to handle ambiguities should also be a critical skill to be taught in a software engineering course [12]. Therefore, we accept that any automated grader should be able to handle vagueness as well, although it would necessarily make the programme more complex.

About how the automated grader is perceived by students, experiences with the programme in the classroom [13] showed that students appreciate the celerity of feedback guaranteed by the grading tool and the completeness of the analysis, despite knowing that their exercise had been mostly evaluated by a computer. The fact that a human instructor was still present behind the tool, who could adjust the marking scheme in case it exhibits weighting problems or other issues that software cannot detect, however, helped reassure students of the effectiveness and fairness of the grading process.

In conclusion, although they involve a few intrinsic drawbacks, the benefits that automated graders provide are worth exploring: if built carefully, taking into account the students’ observed difficulties and allowing human customisation, students can receive a fast and thorough response on their work with only a small effort from the instructor.

In Chapter 3 we will explain why the choice of WC as a programming exercise for COMP61511 also brings many advantages to the way an automated grader should assess it.

2.3 More grading tools and related work

In this section, we describe a few tools for the automatic assessment of code that have been developed and used in previous works, as well as outline common characteristics that we used in our own tools.

The features described by Jackson in his works were grouped into the programme ASSYST [13]. ASSYST offers a user interface that allows a visual management of the student submissions and contains the source code, the tests and the evaluation report as it is generated. The marks assigned depend on a scale defined by the instructor: the score computed by the system for the desired metrics obtains full marks or a fraction of the full marks depending on where it falls in the scale. In order to pass correctness tests, the student programme must compile correctly. Then, the output is compared to the expected one using a pattern matching technique based on Lex and Yacc [1], which guarantees flexibility when assessing the output.

For example, consider a programme that reads two numbers, computes the sum and must print the output as “The sum of x and y is z”. With 15 and 27 as input values, acceptable outputs should be “The sum of 15 and 27 is 42” or “The sum of 27 and 15 is 42”. In this case, the grammar and lexicon defined by the tool allow both versions to be recognised as correct, whereas parsing and comparing the output literally may lead either solution not to be accepted.

Other aspects that ASSYST checks are performance and complexity, using the CPU time and the McCabe’s metric; the code style, measuring the length, the amount of comments and the indentation; and the test adequacy, computing the amount of executed statements over the total statements.

AUTOGRADE [14] assesses the student’s performance by comparing his/her submission to a model programme developed by the instructor. It automatically runs the student’s test suite on the programming assignments developed by other students in the same class: this requires quadratic time in the number of students involved, so the fact that the process can be automated is a great advantage with respect to the feedback speed. Each student receives more points according to the number of defects his/her test suite was able to find in other students’ submissions, as well as how the student’s code performed against other students’ test suites.

The advantages of this approach include a level of competition, since students are motivated to put their best effort into building code that is not easily breakable, as well as designing exhaustive test suites. Also, having multiple students working on a programme with the same requirements should help everybody identify as many defects as possible. Again, the assignment specifications

[1] http://dinosaur.compilertools.net

must leave very little room for vagueness and test suites must follow a standardised format, otherwise a fully automated grader will not be able to recognise correct output properly, and evaluating the correctness will require the participation of a human instructor.

Finally, AUTOGRADE introduces the ability of an automated grading tool to continue the marking process despite any unforeseeable errors that may occur. This is accomplished by producing log information that possibly identify the errors observed and allow the grader to continue with further submissions. Of course, the fact that a submission result depends on how it performs against other students' submissions, like in this case, this ability adds complexity to the grader's logic: for example, after an error interrupts the grading process of one submission, all previous submissions in the class will have to be re-tested.

MARMOSET [15] aims to give students a feedback on their submission as early as possible in the development process, while limiting the amount of information released at the same time.

This tool introduces a system of early releases of the assignment code and a "pay-for-hint" feedback model, where the student can submit his/her solution and receive an immediate feedback on the amount of tests passed. The instructor's test cases, which are usually kept secret until the submission deadline, are instead shared gradually: for example, when students release their programme, the tool shows the passed tests and reveals more details, for instance, of the first two failed tests only. A multiple, yet limited, number of releases is available: virtual tokens, typically two or three per project, are spent whenever students attempt a release, and can regenerate after a few hours.

With this approach, students are motivated to design their own test cases, since the assessment tests are limited, and to start programming as early as possible, in order to be granted more chances to release and receive feedback. Additionally, restricted possibilities to release their code discourage them from programming by "trial-and-error", that is experimenting without a precise method until a solution proves successful.

Also, MARMOSET provides an error tracking system that an instructor can use to evaluate the results at the class level: for example, the tool shows what exceptions occurred most often in the student programmes, which would give the chance to discuss the common problems encountered and improve the class' understanding of the programming exercise.

OTO [16] [17] is a customisable tool for assessing programming projects based both on unit testing and output comparison. It specifically addresses the needs for quick, up-to-date feedback, which would not otherwise help students understand the programming assignment while still working on it, and for a specific and thorough analysis of the assignments to highlight the problems observed (e.g. a feature of the specifications that was misunderstood), so that student get to know the reasons behind any low mark in the final evaluation.

An additional characteristic is the option to integrate the tool with custom assessment metrics and extension modules, in order to address various aspects of marking (not only the tests, but also the style, readability, performance, etc.), as well as programming assignments written in several languages. This is accomplished through a system of individual marking scripts, which define the tasks to be executed to grade an assignment: such scripts not only define the actions to be performed (e.g. compile the code, run the test suite), but also the order and any preconditions that must be observed (e.g. if compilation fails, tests should not be run). OTO also allows the integration of external tools into the system through *extension modules*: such modules can be, for example, the compiler and unit test support for different languages, auxiliary tools like a plagiarism detector or reporting tools like a statistics generator and a system to email students with the report.

PROGTEST [18] [19] is a web-based automatic assessment system specifically designed as support for the teaching of software testing that focuses on the need for prompt and concrete feedback as well. The instructor must provide a reference code that ProgTest will use as instance to test the students' programmes. Also, the set of tests provided by the instructor will be compared to the student's test cases, and a mark will be given based on the rate of tests passed.

An interesting feature of PROGTEST is the possibility for the teacher to allocate different weights for all desired evaluation criteria. Eventually, the tool generates grades that are calculated from the number of tests passed (for example, the ratio between the student's and the instructor's defined test cases), and according to the weight defined for each relevant criterion. Eventually, students receive a feedback of their work and, depending on the instructor's approach, the results can be an indicator of the difficulties encountered more often by students.

The developers are also investigating how a custom testing tool like PROGTEST could be integrated in learning management systems (like BlackBoard, the management system used at the University of Manchester) in a single environment: if the two systems were able to communicate efficiently, this would bring considerable advantages in the way student submissions are managed and assessed.

Finally, WEB-CAT (Web-based Center for Automated Testing) [20] [21] is another tool that is designed to assess the students' testing skills, rather than checking whether the code produces the correct output exclusively, and to provide concrete feedback to help improve their correctness and completeness. Students are required to write their own tests for the programming assignment. However, as long as the assignment is online, the tool allows them to receive unlimited feedback, which is returned immediately whenever the code is released. The suggested mark consists of an evaluation of the coding style, performed by a static analysis, and the correctness of the code, by testing it against the test suite that the student built. The instructor can decide the weight to be assigned to each parameter, and may manually assess other parameters, such as design and readability.

In his works, Edwards [22] used WEB-CAT to rethink the Computer Science curriculum to introduce the concept of test-first (principally using test-driven development as a strategy for development and testing), and showed the benefits of practising such strategy in a classroom context [23]: in particular, testing is often perceived as a difficult, annoying and even unnecessary activity by students, who prefer to focus on producing the correct output and normally do not feel rewarded enough for building effective test suites. However, the quality of feedback that tools like WEB-CAT give should represent a valid help for students to improve their understanding of the programme specifications and above all their testing skills.

Chapter 3

Research and System Design

3.1 Chapter overview

Section 3.2 explains the functionalities and features of the WC tool, which students are required to reproduce for the COMP61511 main programming assignment. The specifications of the current coursework are then described and the problems met during the latest run of the course are analysed. Finally, we propose solutions and improvements.

3.2 WC (GNU coreutils)

WC (short for word count) [2] is a command written in C found in Linux and other Unix-like operating systems, including descendant or work-alike systems such as MacOS. It is part of the base commands of GNU coreutils (Core Utilities), and as we write it has reached version 8.30 (July 2018).

Main flags

WC is a programme that counts the number of lines, words, bytes, characters and the length of the longest line of a given file or list of files, and prints those to the console output.

The indication to compute counts for which parameters is expressed through the use of flags. Flags can be used individually or together in any combination; however, the order by which the counts are printed is always the same (lines – words – characters – bytes – max-line-length) regardless of the order of the flags. Flags may also appear more than once without affecting the print order. Every flag has both a long and a short form, as follows:

`-l`	`--lines`	: print the newline count.
`-w`	`--words`	: print the word count.
`-m`	`--chars`	: print the character count.
`-c`	`--bytes`	: print the byte count.
`-L`	`--max-line-length`	: print the length of the longest line.

WC can also be called without flags. In this case, the programme counts the number of lines, words and bytes, just as the three flags `-l -w -c` were specified, and prints them followed by the chosen file name.

The basic output looks like:

```
$ wc file.txt
11      23      156     file.txt
```

Listing 3.1: Invocation of the WC command. When invoked without flags, WC prints the number of lines, words and characters contained in the file.

Finally, WC can be used to count words, lines, etc. of all kinds of files in any format: this includes not only standard text files (e.g. `txt`, `log`, `html`, `java`,...), but also formats involving different encodings (such as `pdf`, `docx`, `jpg`, `zip`, `mp3`). Ideally, the WC clone developed by the students should recognise all possible formats and compute the correct counts for each of those.

There are a few other features of the WC command worth mentioning.

Newline count. As both the list above and the manual [24] mention, `-l` is the count of newline (`\n`) characters, rather than the line count. The difference is that WC counts the amount of `\n` in the whole text, while an intuitive way to count would be splitting the text and then counting the actual lines. The latter normally reproduces WC's behaviour, unless the file has no `\n` at the end, which would result in an off-by-one error. Students need to keep in mind the difference when designing the WC clone algorithm.

Max display width. `--max-line-length` is not actually the length of the longest line, but the maximum display width. This is evident if the following command is run:

```
$ printf '\t' | wc -cmL
1       1       8
```

Listing 3.2: Invocation of the WC command with a tab character as input.

A tab `\t` is a single, one-byte character, as the counts display, which however has a width of 8 spaces. This shows that the `-L` flag counter takes into account the display width of the characters in each line, rather than their actual length. Since tabs are set at every 8th column, the display width of a tab figures like 8 characters.

Also, non-printable characters are given 0 width. For example, the display length of the `null` character (`\0`) is 0, whereas the character and byte counts consider it nonetheless:

```
$ printf '\0' | wc -cmL
1       1       0
```

Listing 3.3: Invocation of the `wc` command with a non-printable character as input.

–files0-from=file

`--files0-from=` is another possible flag that can be used with `wc`. It disables the processing of files on the command line (i.e. it cannot be used together with a list of files) and computes the counts of those listed in the target *file*. The files named in *file* must be separated by the NUL character `\0`. This can be achieved by using the GNU command `find` followed by a list of files and the `-print0` argument: in Listing 3.4 the list consists of the `txt` files found in the current directory. If no target file is specified, the argument reads a list of files from the *stdin*, as shown in Listing 3.5. Again, the desired counts can be computed by specifying the flags described in the previous section.

```
$ find ./*.txt -print0 > file_for_files0
$ wc --files0-from=file_for_files0
0       1       33      ./test2.txt
2       6       56      ./test.txt
2       7       89      total
```

Listing 3.4: Use of the flag *–files0-from=* with a NUL-separated file.

```
$ find ./*.txt -print0 | wc --files0-from=- -mwclL
0       1       33      33      33      ./test2.txt
2       6       56      56      33      ./test.txt
2       7       89      89      33      total
```

Listing 3.5: Use of the flag *–files0-from=* with *stdin*.

stdin

Like most GNU command line tools, `wc` accepts input data as *stdin* (abbreviation for standard input [25]). The *stdin* is invoked if the argument `-` (dash) is called or if no files are specified in the command. At this point, the user can enter any text, as Listing 3.6 shows. The input mode terminates when keys `CTRL+D` are pressed, then the programme computes the desired counts:

```
$ wc -
Is this the real life?
Is this just fantasy?
Caught in a landslide,
no escape from reality.
4       17      92      -
```

Listing 3.6: Invocation of wc through standard input.

Alternatively, the standard input can be piped to WC. Listing 3.7 shows a few examples how this can be accomplished.

```
$ cat test.txt | wc
       1       2       15
$ echo "hello world\n" | wc -
       1       2       14      -
$ printf "hello world\n" | wc -lcwL
       1       2       12      11
```

Listing 3.7: Invocation of wc through standard input.

Help and Version

`--help` and `--version` are flags that simply print WC's user manual or information about the current release of the tool (see A.1 and A.2).

Errors

WC features error handling messages that the students need to replicate in their Python clone. Errors that WC catches include:

- "no such file or directory": if one of the files or directories specified does not exist (A.3);
- "is a directory": if one of the arguments is a directory (then all counts are printed as 0) (A.4);
- "invalid option": if an unknown short flag is used (e.g. `-k`) (A.5);
- "unrecognised option": if some unknown long flag is used (e.g. `--wordz`) (A.6);
- "extra operand": if `--files0-from=` and other files (or the `stdin` mode) are used together in a command (A.7).

While the first two errors are recoverable, that is WC prints the error next to the bad file or directory and continues with any other files specified, the others do not allow WC to run correctly, so the corresponding error is printed and the programme exits. If more than one of these errors is present, the first that appears in the command takes precedence (A.8).

Spacing

When printing out the counts, WC follows an interesting indentation pattern, which is shown in Listing 3.8: typically, it aligns the counts on the right, then the file names follows after one space character.

```
$ wc *
     12      53     276 file1
    520    4308  182523 file2
     48     226    2752 file3
      7      39    1302 file4
    493    3444  135838 file5
   1200    8137  297945 file6
...
   2280   16207  620636 total
```

Listing 3.8: Indentation pattern in WC.

For the WC clone of COMP61511 assignments, students are not required to reproduce the exact spacing of the programme run on a Unix machine. When the functionality of the programmes is assessed, only the numeric values needs to be correct in order to pass the tests.

Encodings and file formats

Intuitively, a programme that counts lines, words and characters has some kind of text file as input. However, WC's functionality is not limited to simple text formats, such as the popular `txt` and `log` or many programming languages like `py`, `java` and `css`.

Consider for example more complex types of text files, such as `pdf` or `docx`: any programme that visualises such formats may give the option to measure the human-readable content, but the count is likely to be much different from the one computed by WC. Moreover, WC accepts non-text files such as images, audio, video, archives, etc.

To receive full points, the WC clone should be able to accept any type of of file, so the challenge for the students is to understand how WC deals with various encodings: for example, unicode spaces [26] count as word separator as well, and non-printable characters [1] are usually ignored by the counter.

[1]Printable and non-printable characters: https://web.itu.edu.tr/sgunduz/courses/mikroisl/ascii.html

3.3 Current coursework specifications

As already mentioned, students are required to develop a Python-based version of WC over the course of five weeks. The following scheme summarises the specifications of the four assignments, as described in the current coursework.

CW1: “miniwc”

First implementation of WC that accepts only one file name and prints the line, word and character counts followed by the file name, each separated by a tab character and on the same line.

No flags should be implemented. The file path should not be “-”, as the `stdin` mode is not supposed to be implemented yet, so the argument “-” is treated as a missing file.

Error handling must reproduce WC’s behaviour whenever possible, and the programme should also handle “miniwc” errors, that is passing too many or wrong arguments.

No tests are required at this stage of the coursework. The only Python module allowed is `sys`, which is needed to parse the command line arguments. Any built-in functions from the Python Standard Library [27] (i.e. functions that do not involve an `import` statement) may be used.

```
$ python3 miniwc.py file.txt
       12      23      98      file.txt
```

Listing 3.9: *miniwc.py* invocation and output.

CW2: “from miniwc to wc”

The programme must support the three main flags (`-l`, `-w`, `-c`, for enforcing the count of the lines, words or bytes respectively) in any combination and print the appropriate output. A list of file paths can now be provided, still excluding `stdin` as “-” or as an empty list.

Error handling should again cover input errors such as passing wrong flags or other arguments, and follow WC’s behaviour in all possible cases.

Tests can be done using the DOCTEST [28] library, which allows to invoke the `wc.py` script as SUBPROCESS [29] and compare the command line output directly.

```
$ python3 wc.py file1.txt file2.txt -w -c -l
     12     23      98      file1.txt
     25     49     263      file2.txt
     37     72     361      total
```

Listing 3.10: *wc.py* (CW2) invocation and output

CW3: "refactoring"

Python's ARGPARSE [30] and UNITTEST [31] libraries must be used to refactor the code. No additional WC functionality should be implemented at this stage.

Students are then required to write an essay where they compare their experience with DOCTEST versus UNITTEST and ARGPARSE versus manual argument parsing from `sys.argv`.

CW4: "last bits of functionality"

The programme must accept all remaining flags (`-m`, `-L`, `--files0-from`, `--help` and `--version`) in any combination and form (short and long options). Support for the `stdin` [32] mode must be provided. The programme should recognise all types of input files, including non-text formats. All outputs should be identical up to space, including error handling.

The TIME utility [33] should be used to compare the performances of WC and the student's `wc.py` with input files of various size. Students should then discuss the results in a short essay.

3.4 Problems observed

The current assignments did not show any evident structural problems. However, it is possible to think of features that could be improved, and even new tasks and technical challenges that may add practical value to some concepts taught in the course – concepts that represent valuable skills in Software Engineering.

Late feedback

The aspect of the coursework that was most criticised, and on which we focused most within this project, is the lack of prompt feedback: the final submission, that is CW4, was due on 4th November 2017; the feedback and grade were released on 14th March 2018, almost two months after the course exam.

The feedback came very late due to the lack of working automated tools to assess the programming assignments, which required the instructor to do most of the grading process by hand. Being able to rely on a software product that does the job correctly and efficiently would benefit both the students, who would receive feedback more quickly, and the instructor, who could save hours of work and potential errors in assessing dozens of submissions.

Feedback thoroughness and information given

Quote 1 represents an example of a feedback that students received for CW1.

> There seems to be a miniwc.py (1). Only "sys" was used (1). You passed 23% of the basic tests for 1 of 5 points. Unicode spaces confounded you, costing 1 point. Binary files confounded you, costing 1 point. Your formatting was correct for 1 point. Total: 4/10.

Quote 1: Example of feedback for CW1 given to students.

This kind of feedback clearly explains how the points were attributed. This was useful to understand one's errors and missing features, however this information was shared after the coursework had been concluded and submitted for several weeks. This made the feedback not as much relevant and helpful as it would have been if shared little time later, possibly before the end of term.

We believe that the deepness of the feedback could also be improved: especially in the early stages, receiving a mere percentage of passed tests (e.g. 23%) does not represent much useful information about what errors were made. Some weaker students might even feel demotivated in case they receive a low mark in this respect without knowing the reasons behind the failed tests. However, this matter will be discussed more thoroughly, since sharing tests cases indiscriminately (as in "these are the tests you failed") would not give students a chance to review their code and think on solutions on their own.

Also, in Chapter 4 further assessment criteria that were adopted last year will be discussed – for instance, the weight that failing tests for files containing binary characters or symbols in different encoding formats (e.g. `pdf`, `docx`, `png`,...) should have.

3.5 Motivation and objectives

This section analyses the reasons behind the choice of building a `wc` clone as a programming exercise for COMP61511.

3.5.1 Complexity and Understanding a specification

Programming complexity was among the topics of the first COMP61511 lecture. The lecture analysed what are the characteristics that make a software system complex, examining in particular the so-called "Rainfall problem". The original description [34] of this assignment states:

> Write a program that repeatedly reads in positive integers, until it reads the integer 99999. After seeing 99999, it should print out the correct average.

The performance of the students while developing this programme has been studied extensively over the years [35]: an apparently trivial problem contains challenges that many novice programmers fail to consider, leading to logic errors such as not ignoring negative input values or forgetting to add a "divide-by-zero" error when computing the average [36]. Also, it has been observed since 1983, that students tend to make the same kinds of mistakes over the years [37], thus several studies have examined the characteristics of the Rainfall problem and attempted to explain why even an easy programming task can hide difficulties.

The analysis made by Seppälä *et al.* [38] indicates that students often may not pay enough attention to the problem statement, or assume that some specifications did not have to be included in their implementation, since not explicitly stated in the description [39]. Additionally, poor results in the functional correctness of the Rainfall programme and lack of consideration for corner cases that trigger errors in the execution suggest that programmers should focus more on testing, which in many cases helps identify bugs in their code.

Concerning the choice for COMP61511 coursework, WC is undoubtedly a more complex programme than the Rainfall task, which is appropriate for a four-week-long assignment, yet it is small and focused enough to allow the development of a complete work-alike. However clear and well-defined the specifications are, students still need to understand the problem they are solving and learn the functionality to reproduce even if not explicitly asked to (e.g. one may assume that their clone only needs to work for standard text files, while WC accepts *all* kinds of files).

As a result, from the point of view of complexity, WC gives student the chance to learn how to extract and fully understand a specification, elaborate a design and finally test its functionality to discover corner cases.

3.5.2 Creating useful programmes

The time available during term is hardly enough for a student to design and build a medium-sized software product, even when working as a team.

There is not much that can be done to increase the size of school programming projects, because of the limited time. On the other hand, suggesting larger tasks would help to create relevant experience for a future job as software engineer – since professional software products often contain several thousands of lines of code [40]. However, school projects also tend to be developed "for their own sake", that is students build them to get a grade, and they are typically not very utile from a general user perspective.

wc is a tool that already exists, is installed on millions of systems and is used in many software packages, so the students are not creating anything new. Nevertheless, it is a real programme: at the end of the four weeks, students should be able to understand thoroughly how it works and their clones should reproduce its behaviour as closely as possible. We believe these – working on a real programme and being able to duplicate a behaviour – also represent valuable skills to be acquired. For example, the wc clone could be a substitute for any machine that does not have wc by default (e.g. MacOS' wc version has limited functionality).

3.5.3 Reverse engineering and Black-box testing

Specifications rarely cover all possible behaviours a programme can have. Therefore, while building their clone students have the original wc as reference: as a general rule, the same input data should produce the same output between `wc.py` and wc. This is the concept behind reverse and re-engineering [3] – the analysis and the process of reconstructing a software end product – and black-box testing [41], that is basing the tests on the output value only, without knowing about the implementation [42] (as opposed to white-box testing, which is the normal testing programmers do with their own code).

wc is invoked from the command line and produces some output, which is the sole information the programmer can base the implementation on. The task is then to build a model for wc determined by tests and external observations. Figure 3.1 shows the feedback loop that is typical of this process of re-engineering [43].

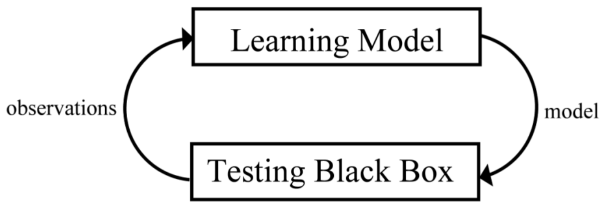


Figure 3.1: Learning model of black-box testing [43].

Therefore, a clone of wc is an opportunity to re-engineer an existing software tool, as a teaching approach to abstract a system's features and recreate it based on their understanding of the input-output behaviour [44].

Building a version of wc is also a standard example of program slicing [45], that is reconstructing a programme by studying its data and control flow and capturing its execution path. In this case slicing a programme literally means decomposing it to its minimal form that still produces the original set out output values: this is evident in the process of reproducing `wc`'s behaviour, starting from `miniwc`, that is the minimal functionality, up to a complete working version.

3.5.4 Assessment automation

Due to its nature of command-line based tool, it is relatively easy to test and assess the students' `wc` clones. Additionally, the process can be automated in many of its parts: once a set of test cases, consisting of a input command and an expected output result, are defined, a grade can execute them on `wc.py`, store the result and compare it to the expected output.

These are advantages in the context of a moderately large software engineering class, as automating the evaluation and grading process would provide quicker feedback to students, which could help improve their performance. This opportunity will be discussed further in section 3.7.

3.6 Solutions and improvements

Faster and more useful feedback

Feedback represents essential information in software development and testing to guide the students towards the right path. It needs to be frequent, quick, if not immediate, and useful in assessing strengths and weaknesses of code [46]. However, it would be extremely laborious and time-consuming for a teacher to provide such feedback manually.

Also, within this project we investigated how feedback should be given to students so that it guarantees the maximum possible gain in terms of learning and improving, yet without preventing students from identifying independently any mistakes they made and fixing them. We want the evaluation to be educative, so that students can think of their own solution once given an appropriate hint, possibly supporting those who are facing major difficulties, while avoiding to give them a ready-made answer that will not help their learning progress.

Overcoming BlackBoard's limitations

The School of Computer Science at the University of Manchester uses BlackBoard Learn [2] [3], a virtual learning environment, to manage most courses: among many functions and tools, the online app allows instructors to share the content of their lectures, assign homework exercises to students and grade those after they have been submitted.

[2]http://www.blackboard.com/
[3]http://online.manchester.ac.uk/

Unfortunately, BlackBoard does not grant much customisation when grading assignments: for example, it is not programmed to select and download only the most recent submissions (assuming that multiple attempts are allowed) and those sent before the deadline. It does, however, allow instructors to compute a set of grades "offline", for example on the instructor's machine (rather than, for example in case of a coursework essay, reading the student's attempt online and assigning a grade directly), and then upload the computed marks. For this reason, we are going to use automated tools to grade the submissions and exploit them to overcome this and other limitations of the BlackBoard environment as well, such as filtering out the late and old submissions.

Proposal of changes

As previously stated, we did not introduce consistent changes in the coursework structure, since we believe that the existing steps in building a clone of WC are logical and well-defined.

In Chapter 4.2 we will, however, propose a new set of specifications. These are not very different from the old ones, except for a few small changes in the order of the tasks, since the aim of the changes is to make them as clear and understandable as possible. Also, the objective is to improve the continuity with the coursework content.

New exercises with larger code bases

Students in the COMP61511 class wrote 263 lines of code on average for the final version of WC. This result makes it a reasonably complex project for a university course, but is significantly smaller than any software project the student will probably have to deal with in a work environment as software engineer.

Complex software systems can have thousands or even millions of lines of code [40]. Also, the course syllabus follows the IEEE & ACM 2009 Software Engineering Curriculum Recommendations [47] concerning the expected outcomes for the MSc course, which state:

> A student who has mastered the [Core Body of Knowledge] will be able to develop a modest-sized software system of a few thousand lines of code from scratch, be able to modify a pre-existing large-scale software system exceeding 1,000,000 lines of code, and be able to integrate third-party components that are themselves thousands of lines of code. Development and modification include analysis, design, and verification, and should yield high-quality artefacts, including the final software product.

For this reason, an effective way to engage students with a large project is suggesting an existing software system to study and use to develop more complex set of programmes. In this matter, Python offers plenty of open-source frameworks and libraries that can be used for this purpose.

A further possibility would then be giving students the guidance and the tools to integrate one or more of these libraries with their own contribution, such as building a plug-in, which would represent a positive indication that they have really mastered and deeply understood the functionality of the target programme.

In this project we will not discuss in details what these new larger assignments should be due to the limited time available. However, we will try to provide the instructor with all the tools necessary to design new exercises independently and use the grading tools described here to assess them, although the grading tools have been designed to evaluate a precise set of exercises.

3.7 Features of the assessment tools

In the background chapter we presented several examples of existing tools that address the problems related to automated grading with different approaches and techniques. This section analyses the structure and features of the tools developed for this project, presents similar aspects and discusses differences.

Human contribution in automated assessment

First of all, the purpose of developing automated grading tools is mainly relieving a human instructor from doing most of the assessing activities manually. As Jackson pointed out in his works [8], there are many tasks in grading assignments that are remarkably repetitive for a human, while a computer that has been instructed to perform the same tasks might complete them more quickly and effectively. Also, considering how difficult is for an instructor or teaching assistant to treat all submissions in a class equally, especially after many hours of repetitious assessment, there is little doubt that a machine would perform better in this situation too, in terms of accuracy and impartiality.

The evaluation of a programme like a clone of WC, moreover, is particularly suitable for automated assessment: in most cases we can count on an unmistakeable source that tells whether the WC clone's output behaved correctly or not, that is the original WC itself. Except for few custom error messages we need to introduce for not-yet-implemented features, the only task we need to do is running WC to obtain the correct output for some given input, and this can be automated as well, once we have defined a list of relevant test cases to execute on the student's programme.

After having a list of input and expected output values, the assessment is simply about comparing automatically WC's and the clone's outputs, which is performed equating the counts obtained:

again, the process may take days of work from a human, especially with a large class, and is potentially error-prone.

Vagueness in the specifications – Model solution for testing

Another issue that Jackson [13] highlights concerning automated grading is the need for students to follow a precise set of specifications that leaves very little room for ambiguity: otherwise, that is leaving students to freely interpret the requirements (in the limits of the requirements themselves) and for example print out their solution without strict regulations, the risk is that no automated tool will be able to differentiate between a correct, a partially correct or a wrong solution.

Fortunately, once again WC is a suitable programme in this sense: once the requirements for each phase of the coursework have been defined, the chances for students to misunderstand them and develop a partially correct functionality that only a human could detect are strongly limited. This is due to the fact that the original WC represents a "ultimate model" and a turning point in establishing the output correctness: after ignoring the spacing, a programme would only need to compare the counts computed and verify the formatting, and if either does not correspond, then the test case fails.

This makes the WC clone a relatively easy programme to build an automated grader for, in comparison with many assignments that have a wider set of acceptable output values and behaviours, and therefore would require more tolerance from a tool that aims to assess them automatically.

Grade scale

When grading a programming assignment, there are metrics that can be assessed just as right or wrong, with no need for the "partially correct" spectrum. In the case of WC submissions, for instance, either a script exists with the expected name or not, so it can be treated as a binary parameter and will be given 1 or 0 points accordingly (this is also similar to the approach that PROGTEST [18] uses to allocate different weights for the various metrics).

The same could be done with the various tests – either a test case passes or fails –, but to verify where the functionality under test (e.g. the byte counter) was implemented correctly multiple tests are usually necessary. For this reasons, marks will be assigned following a scale established by the instructor: this could be either a proportional scale (e.g. if one metric had a 40% passing rate it will receive 2/5 points, or 4/5 with around 80% passing rate) or an arbitrary scale (such as, test suites with < 25% coverage are given 0 points, between 25% and 85% are given 1 point, over 85% 2 points).

Error tracking

The grading tools were developed and tested to cover most cases and recognise most errors that the students could do, however it may happen that some unforeseen error is not caught properly and causes the grader to stop working. Following the approach realised by AUTOGRADE [14], the grading tools are guaranteed to continue running at least for the next submission in the list. Additionally, a log file is created for each submission and is intended to save the results computed. It will also be used to save the messages related to such errors, so that the instructor can identify and fix the problem in the code more easily.

Similarly to MARMOSET [15], the reason behind the error tracking system (both student errors and exceptions caused during the grading process) is also to have an overview of the most common problems met by the class. If many students failed a precise set of tests or have similar exception stack traces, it may be an indicator of collusion, or it could just mean that the class perceived some tests as too difficult. The objective for the instructor is therefore to gather useful information, both at the single student and at the class levels, to improve the coursework and the grading process itself.

Feedback

Many existing tools we analysed focus on how to give feedback to students. First, of course, the main purpose of our automated graders is to provide quick feedback – that is, available within a few hours or days at most.

Various criteria are used to assess the students' submissions, so each feedback will consist of a list of marks and a brief explanation that states whether the corresponding requirement has been or not. The advantage of having this kind of feedback as early as possible is of course that the student gets to know in what features of the submissions he/she did as expected and, vice versa, if there are aspects of the coursework that need to be improved.

Some tools examined, such as WEB-CAT [20], introduced the option of students submitting multiple versions of their code, which results in multiple feedbacks too. This matter will be discussed further in Chapter 4.7, where we will describe possible feedback models, different in frequency and "price" for the students, to be adopted for the COMP61511 grading tools.

Cross-testing

Although it could provide useful information about a student's skills in testing, cross-testing – that is running a student's test suite against all other student programmes in the class to see how well it performs – is a feature that was not implemented. As the case of AUTOGRADE points out [14], if an unexpected error causes the grader to be interrupted while assessing a submission, this will lead to the re-evaluation of any previous submission, as each one's score partly depends on all other's in the class. At this stage our tools might not be robust enough to guarantee a smooth process, so we preferred not to introduce this complexity.

Extending the tools

The grading tools were conceived to assess the four phases of the COMP61511 coursework. However, although very focused on the task of testing the behaviour of a WC clone, we thought it would be useful to make them extensible to other kinds of programming tasks. For example, most functions that are shared between the four marking scripts are general enough to guarantee this (e.g. the function that runs a command, saves the output and compares it to a given expected value up-to-space): if not immediately reusable, these can be adapted to serve similar purposes.

Also, following an approach similar to OTO's, the grading scripts are mostly composed of small semi-independent modules that can be simply called with a minimal effort to modify the code.

3.8 Risk considerations

Changes introduced by a former student

This project for the improvement of the COMP61511 course is being carried out by a student who has lately attended the course itself as part of her MSc degree. While this fact represents an advantage with regard to the knowledge of the programme and the recent experience with the topics and tasks, this might represent a risk factor for objectivity and impartiality.

For example, the author has only gone through the course as a student, therefore understanding how some mechanics work from the teacher's point of view might not be immediate. Similarly, what students think concerning the grades and the feedbacks, or in general about the grading process of their coding exercises, might not correspond to the lecturer's view, due to both the lack of experience in teaching and probably the inability to judge their own work objectively. Also, in the context of re-designing the coursework structure, deciding whether the difficulty of some exercises is adequate to the level of the class could be a challenging task. Finally, some ideas that a student has to improve the teaching might result in little pedagogic impact, again due to the lack of expertise.

In order to overcome these difficulties, the author committed to work closely with the course's instructor, who could help and guide the author with suggestions to choose what he believes to be the best approaches to teaching.

Effectiveness of changes

Another risk concerns the actual changes that will be made to the course structure and content: as described in Chapter 5, their real effectiveness will only be tested when the course is taught in a classroom context during the following years. However, all changes were motivated by scientific evidence and/or the experience and judgment of the teacher, and the ad-hoc assessing tools were extensively tested, as the following sections will show, so that the risk of introducing ineffective changes is limited.

3.9 Ethics and professional considerations

Privacy

Some of the deliverables that the project will create are automated tools that will test and evaluate pieces of code written by students. The tools were tested on sample versions of the desired programme and on the code exercises produced by students over the past years, in order to achieve the maximum possible code coverage. The use of actual coding exercises created by past students for testing the ad-hoc evaluation tools was relevant because they represent actual pieces of code, which can be exploited as training material for a work of software that is proposed as a "substitute" for the teacher.

However, as provided by the university's regulations concerning the use of material produced by students, the code exercises will be anonymised – that is, tracing the author of the source code will not be possible by examining the code itself, for example by the first and last name, university ID number or email address.

Students' usernames

The only means of identification in the assignment were the student usernames, which are generally composed of a eight-character string of letters and numbers. Usernames were assigned to students at the time of enrolment at the University and it should not possible to trace back the name or other information of the student through the username; however, for security reasons the past student submissions with real usernames were used, exclusively for testing purposes, on the instructor's machine only, where they were gathered and stored originally. Also, in the local result database, usernames were replaced with random strings and the aliases were stored in a separate table, so that it can be easily destroyed once the need to go back to the real username no longer exists.

In this work, the sole purpose of preserve the username was to keep track of and differentiate the assignments. The only real username used is the author's (*"mbaxtmp2"*), and all other sample assignments composed to test the grading scripts were fabricated randomly, preserving the base structure (8 letters and numbers) only.

Code samples

All code samples were used for testing purposes only. The code chunks that were inserted in the dissertation as practical examples, for instance to illustrate some common development errors made by students, are either anonimysed or derived from the author's original coursework.

Future students

This work created a set of tools to grade student assignments and specifies evaluation parameters that are likely to be used for future runs of COMP61511. Therefore, it is essential that all information described in this dissertation is kept private and secure from prospective students. The material that should not be shared with future students include: the coursework specifications (until they are asked to implement the coding assignments during the term), the source code of the grading scripts, the test cases used to assess the exercise.

Evidently, sharing this material (e.g. a student who gets to know in advance what the tests cases are) would at the very least falsify the students' results and performance, which would additionally defeat the purpose of the whole Software Engineering course.

Code quality

Since the assessing tools shall be presumably used by future professors, teaching assistants and students, the author committed to maintain the highest possible quality standards in writing the code, especially in the aspect of maintainability, readability and fault tolerance. In fact, it is likely that other programmers will have to intervene on the code, as the author will no longer have access to the computer systems of the university after graduation, therefore these three qualities are especially important in this case.

Chapter 4

System Implementation

4.1 Chapter overview

In Chapter 4 we list and discuss what the coursework specifications should contain in a future run of course COMP61511, including any changes suggested for the coursework itself, then we describe in details the features of the grading tools that have been developed for this project.

4.2 Suggested specifications for new assignments

The structure of the coursework was not changed substantially, since building a clone of WC as a programming exercise in subsequent steps does not leave much room for reversing or combining the various phases: it already makes sense to begin with the most basic functionalities and proceeding by gradually adding more complex tasks. However, a few adjustments and improvements were introduced.

4.2.1 Week 0 (before the lectures start)

Preliminary questionnaire: picture of the class level, feedback on the student's current knowledge

During the week preceding the first lecture, the instructor sent an email to the class with information about the course and material on Python and Linux, recommending that students install Python on their machine and learn at least the basics of the language, since having the working environment functional would help them during the first lab. Later, it was noticeable that some students did not follow the tutorial despite having little or no knowledge of Python, Linux and the command line, so they had trouble when setting up their machine for Lab 1 and the first coding exercises. If they had been familiar enough with the language and the environment, they could have focused on the problems for the whole lab, which would have then simplified their homework and probably spared them some time.

Clearly, students should not be forced to start with the basics of Python, especially if they already have experience with the language. However, during Lab 1 students were asked to complete a short online quiz with simple questions about their current knowledge of Python and Linux, so that the instructor could get a picture of the level of the class. Our suggestion is that a similar quiz should be given earlier and students should receive a feedback: for example, based on the results and the mistakes made, it could suggest online tutorials for novices or more experienced programmers, or vice versa reply that the student's knowledge is adequate. Perhaps a bonus point might be given to students for completing the questionnaire.

4.2.2 Week 1 – miniwc

Git: version control and backup

The practice of version control should be encouraged from the beginning.

Software development is typically managed through subsequent versions of code: developers use it to keep track of the changes they make in code, to easily switch the working version (for example in case of problems with the latest one), to create branches to work on different features of the code in parallel, and also to have an updated backup of the source code. The use of version control is critical when there are many programmers involved in the development, but even if there is only one programmer working on a project (this is the case of COMP61511 coursework, as students are required to work individually on their programmes) [48] it brings considerable advantages.

A widely used version control system is Git [1]. At the moment there are no official data available about the experience with Git during the 2017/18 COMP61511 course, but the author realised, while working on the WC clone and comparing her result with her colleagues', that there was a significant number of students that had never worked with Git before. Also, during the following course of the Software Engineering module – COMP62521: Agile and Test-Driven Development –, which was largely based on Gitlab, three out five members in the author's team still showed little confidence with the tools.

Again, we expect that students at a Master's level become aware of their own weaknesses and recover autonomously from any prerequisite they lack, but we understand that they have often obtained their previous degree in several different countries, therefore they are likely to have variegate

[1]http://git-scm.com

backgrounds and past experiences [49]. Particularly if this case applies, we want students to learn the practical use of Git, at it will be an essential skill in a typical working environment as software engineers [50]. The tool that students should use is the School's Gitlab system [2], however any other Git service such as GitHub [3] or BitBucket [4] will be fine, as long as students use a private account to commit their code, since having their repository public might result in collusion or plagiarism.

Miniwc specifications: more focus on objectives

No changes to the requirements for `miniwc` were introduced: the programme still consists of the basic functionality of WC, i.e. calling the programme from the command line with a single input file will produce the line, word and byte counts. The differences from last year's version are:

- The name of all files needed so that the submission can be considered complete – that is `miniwc.py`, `doctest_wc.py` and a directory named `test_files/` containing a set of input file for the doctest – are stated in the requirements. This is mainly to facilitate the automated tool's assessment job, as its capacity to identify the correct files would be limited if the files were given names of the student's choice (e.g. if a student developed two versions of the programme, called `miniwc_1.py` and `miniwc_2.py`, the tool would not be able to understand which one must be assessed). However, we want students to follow the requirements thoroughly, including the name of the files, as following a specification carefully is a relevant skill for software engineers [51].

- There is a definite set of points for handling files with binary, unicode and other special characters. For COMP61511 coursework 2017/18, the instructor did set goals for unicode spaces and binary character as well (see section 3.4), and the specifications stated that the WC clone had to work will any kind of file. However, since the feedback came after a few weeks, many students realised too late that the challenge of unicode and binary characters was in fact an important characteristic of the programme. Our suggestion is therefore to specify more clearly in the requirements that the clone must accept binary files, possibly in the form of a hint (e.g. "We saw that WC counts a new word when there is a space. Is the character " " the only space a computer understands?"), which would initially give the chance to investigate the issue autonomously. Also, quicker feedback will allow students to know if their solution is on the right path, or to receive more assistance if they are having difficulties.

[2]http://gitlab.cs.man.ac.uk/
[3]http://github.com/
[4]http://bitbucket.org/

- The practice to build a clone for `wc` makes use of black-box testing and follows the rule "equal input must produce equal output". However, for `miniwc` we limited the functionality of `wc`: receiving two files as input will work for `wc`, but following the `miniwc` specifications this would be an error to catch. For this purpose, since the assessment tool will check if the requirements have been satisfied, a simple solution is to provide sample error messages that students could insert in their programmes – for example, "Error: only one file allowed" or "Error: stdin not implemented yet".

Testing: how students verify that the code works as expected

Students are asked to design a preliminary test suite for their script using DOCTEST, which will not be assessed yet (e.g. with line and input coverage). The purpose is to let students get familiar with the library for when their test suite will be evaluated.

Evaluation and feedback: assessing students' programmes

Again, the core structure of the feedback is very similar. The main changes introduced affect the points given to the amount of tests passed, which separately consider plain text, binary and unicode files. Also, separate points are given if the single files needed for the submission exist.

For inner statistical purposes, percentages of tests passed and other data, such as any known errors that the student made, are automatically computed and saved to a local database.

The feedback is intended to be available to students within a few days since the submission deadline. Students will receive a grading schema that explains which marks they obtained, thanks to successful tests or requirements satisfied, and which they missed, with a brief explanation.

4.2.3 Week 2 – From miniwc to wc

Discussion of CW1 results

Thanks to the grading scripts speeding up the assessment task, feedbacks for CW1 should be available to students by the end of the second lecture and lab. This would give the instructor the chance to discuss the results of the class, possibly highlighting the most common errors and failed tests observed in the students' works. During this lab, hints and directions for the difficulties encountered most frequently may be given to students. Moreover, having the results of CW1 already available, should give the teachers themselves an overview of the class' level: for example, if a considerable number of students faced serious problems while setting up the Python environment or figuring out what to do to complete the coursework, the instructor might decide to allocate more points for some of the easier tasks, in order to level the marks.

Specifications and Functionality testing

- In the previous CW2, only the flags `-l`, `-w`, `-c` had to be implemented. Our suggestion is to add the remaining two, `-m` and `-L`, at this stage already: after understanding the logic behind flags, implementing two more should be fairly straightforward. Moreover, if the two flags are already working, later there will be more time to focus on more complex functionality, such as the `stdin` mode or the binary files.

- At this point, the `wc` clone should accept the said flags and multiple files, so additional functionalities have been introduced since `miniwc`. The specifications will describe the remaining error messages that students will have to insert in their code, concerning the flags `--help`, `--version` and `--files0-from` and the `stdin` mode. In particular, the flags that have not been implemented can be treated as standard invalid or unrecognised options (see 3.2), while any command that triggers the `stdin` mode should return a warning reading that the `stdin` is not available yet.

- The doctest suite should be fully functional at this stage, since it will be assessed and give a defined amount of marks for this coursework.

  N = percentage of tests passed. Max points = 5.

$N = 0$	$\Rightarrow$ 0 points
$0\% < N < 25\%$	$\Rightarrow$ 1 point
$25\% \leq N < 50\%$	$\Rightarrow$ 2 points
$50\% \leq N < 75\%$	$\Rightarrow$ 3 points
$75\% \leq N < 100\%$	$\Rightarrow$ 4 points
$N = 100\%$	$\Rightarrow$ 5 points

  Quote 2: Example of grading ranges for tests passed.

Evaluation and feedback

- Intuitively, CW2 and the following exercises introduce new behaviours of the programme, but are not supposed to break the existing functionality. For this reason, the tests performed to assess the functionality of `miniwc` will be used again for `wc` of CW2. Tests for the new functionality will be given more marks (e.g. 6 points for `wc` tests; 3 for `miniwc` tests), and the amount of marks given for the previous coursework could decrease after each week (e.g. 6 total marks for `miniwc` in week 1, 4 in week 2 and 2 in weeks 3 and 4), as new functions are introduced.

- Unless it is only pass/fail (for example, if the `wc.py` script is present the submission receives 1 mark; if it is not, 0 marks), the amount of points for each grading parameter is given proportionally to the percentage of tests passed. For instance, a programme that passed 50% of the tests should be given 2 to 3 points out of 5, depending on the marking scheme that is decided by the instructor. In order to get full points, 100% successful tests should be reached. See Quote 2 for an example. The amount of points given for the actual number of test cases has been reduced since the previous version, as we introduced a more effective criterion to assess input coverage.

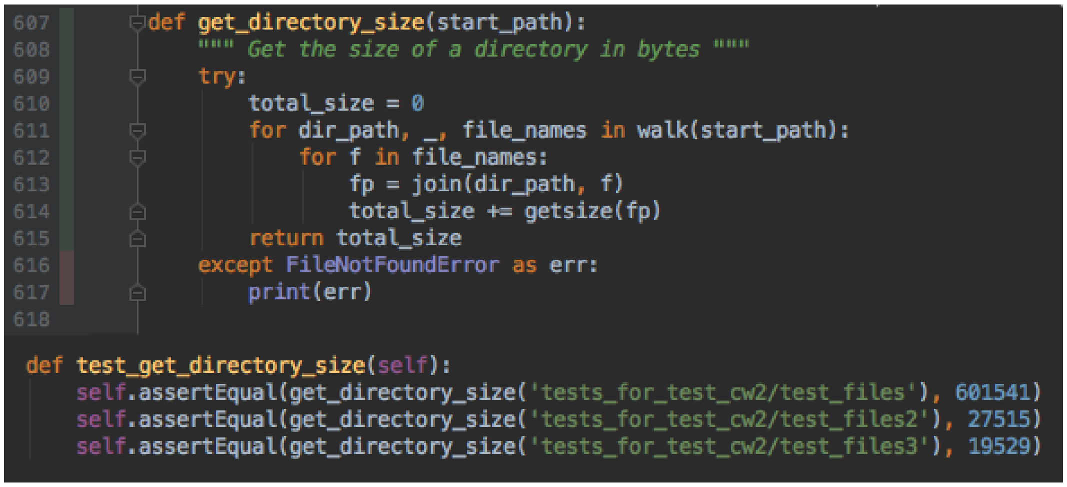
- The tool adopts various criteria to assess the student's test suite, mainly concerning the input coverage: the students need to insert a minimum number of test cases to obtain marks (for example, 100 cases – again, this is customisable by the teacher), have a variety of input files in terms of size and formats, and above all the test suite should cover as many lines of code as possible. Line coverage is a criterion that has been introduced to better assess the students' tests, which were only evaluated with static analysis in the previous runs of the COMP61511 course. It is important to have a good code coverage to ensure that all code branches have been tested.

  In the example in Figure 4.1, the green bar on the left shows the lines that have been covered by the test (below). There is no test involving a directory that does not exist in the path specified, in order to show that the last two lines of the `get_directory_size` function are not covered.

4.2.4 Week 3 – Refactoring

Discussion of CW2 results: peer review

An interesting activity that was proposed during Lab 3 was code review in pairs. A possible problem that was observed during peer code review in 2017/2018 concerned pair combinations: students were paired in a similarly random way (they were asked to choose a partner they were not familiar with, in order to avoid that close friends could falsify the evaluation) and the task of each member of the pair was to run their own test suite on the colleague's `wc`. The objective was to identify errors or test cases not covered by the other's programme.



```
def get_directory_size(start_path):
    """ Get the size of a directory in bytes """
    try:
        total_size = 0
        for dir_path, _, file_names in walk(start_path):
            for f in file_names:
                fp = join(dir_path, f)
                total_size += getsize(fp)
        return total_size
    except FileNotFoundError as err:
        print(err)

def test_get_directory_size(self):
    self.assertEqual(get_directory_size('tests_for_test_cw2/test_files'), 601541)
    self.assertEqual(get_directory_size('tests_for_test_cw2/test_files2'), 27515)
    self.assertEqual(get_directory_size('tests_for_test_cw2/test_files3'), 19529)
```

Figure 4.1: Line coverage after running tests.

Studies have investigated the experience of peer review in the classroom, in terms of accuracy and effectiveness of the review on learning outcomes: reviews allow students to compare their work with each other, learn new techniques and put their effort into thinking critically about their approach [52]. It may be worth considering to conduct aggregate peer reviews, that is to have one work assessed by multiple students, which helps improve review accuracy[53].

Naturally, when pairs are formed in a random fashion, it can happen that students that received a high mark in the first assignment are paired with somebody who received a lower grade. While this may represents a valuable opportunity for the "poor" student to improve his/her code, the contribution that the latter gave to the "smart" student's code quality was not as relevant. A solution to this might be, once the initial feedbacks are available, pairing students who obtained similar grades.

Specifications and Functionality testing

- No new features have been introduced for this stage of the coursework.
- During this week students need to understand how to use the packages ARGPARSE [30] and UNITTEST [31] and use them to refactor their code. Refactoring is a process that does not change the behaviour, but only the *non-functional requirements* of the code. Typical refactoring aims at improving readability and maintainability and reducing complexity, or simply re-designing some functionality. Therefore, purpose of CW3 is to change wc.py so that it uses ARGPARSE to receive the command line arguments in place of `sys.argv`, and to re-structure `wc` in *units* (i.e. separated functions, as opposed to a single block for the whole

`wc` functionality, like some students may have implemented the programme) so that it can be tested using UNITTEST. Another task for the coursework, to be submitted at the end of term, is to write a short essay that compares the student's experience while refactoring and any advantages and disadvantages observed between the two versions, pre- and post-refactoring.

- Students now need to use specific names for the argparse and unittest files (e.g `wc_unit.py` and `wc_argparse.py`), for the reasons described in 4.2.2.

Evaluation and Feedback

Although no further functionalities need to be introduced at this stage, students have the chance to improve their existing code, apart from refactoring. For this reason, the assessment for CW3 only involves a few more complex test cases for the evaluation. At this phase, the student's `wc` clone should:

- pass all previous `miniwc` tests;
- accept any input consisting of flags (short and long form) and files in any number and order (including combinations and repetitions, which we remind do not affect the printing order);
- recognise all sorts of "unknown" flags and display an error message like WC does, and treat any missing files accordingly;
- still return a custom error if not implemented flags or the `stdin` mode are called.

4.2.5 Week 4 – Last bits of functionality

Discussion of CW3 results: errors observed

Again, thanks to the grading tools the results for the whole class should already be available: no extra functionality of WC was added since last time Also, one of the topics of this lecture is debugging, so the instructor could decide to take actual bugs from the students' programmes as examples, possibly directly asking the students to explain how they realised a bug was present in the code and describe the procedure they adopted to solve it. This should represent a chance for weaker students, who might still struggle with errors in their code, to receive suggestions on what steps to follow in order to tackle a problem in their code and what specific tools could be used to ease the process (e.g. the debugger offered by their favourite IDE and Gitlab's issue tracker). The lecture could also present concrete examples of debugging strategies, possibly describing benefits and drawbacks of each of those (e.g. using a debugger versus manually printing/logging).

Specifications and Functionality testing

- As the final stage of the coursework, students should implement all remaining functionalities of WC, that is the flags `--help` and `--version` and the `stdin` mode. The two flags `-m` and `-L`, respectively calling the character count and the maximum line length, should already have been implemented: now the students should have time to focus on finishing the programme, so that it is as similar to the original WC as possible. At the end of CW4, the clone should accept all kinds of files too.

- The objective of refactoring in the previous phase was also to request students to think critically about possible programming techniques and approaches that can be adopted when developing some functionality: for example, students should now have a definite idea of what the advantages of unit testing over DOCTEST are, or vice versa, or again if both can be used at the same time to accomplish different purposes. For this reason, at this stage students are free to use either testing framework. They need, however, to include both test suites (e.g. `doctest_wc.py` and `unittest_wc.py`) in the submission, as both version should be available since CW3 and will be assessed for line coverage and other parameters. Similarly, either approach between `argparse` and `sys.argv` is a choice the students can make independently, according to the observations done while programming and later on in the essay.

Evaluation and Feedback

- Separate marks are now given for passing different categories of tests, namely the usual `miniwc` cases (2 marks), the `wc` (simple and complex cases in CW3 are now a single category, for a total of 7 marks), `--help` and `--version` (2 marks), `--files0-from` (2 marks) and `stdin` cases (3 marks). Again, all points specified here can be customised.

- In the previous run of the course, the output of `--help` and `--version` flags were tested using the exact WC output (see A.1 and A.2). While this makes sense for `--help`, since we want it to describe the same commands as WC, some students thought it would be more reasonable to insert personalised information in the output of `--version`, such as the student's own name or the University of Manchester as the copyright owner. As a results, no points were given in this case, even though the general functionality was correct (e.g. the flag `--version` "wins" over the counter flags: when calling, for example, `python -w directory/* --version`, the output is the version page, as in A.2). Additionally, the content of the version page may not be the same over time, such as the actual version of WC, which is 8.30 at the moment but is likely to change in the future. To solve this problem, a function that detects similarity between text contents has been implemented (using the *SequenceMatcher* class of the Python's DIFFLIB package [5]). If the similarity index is 1, two strings are identical; if it is 0, they have no letters in common in the same position (e.g. "apple" and "peach"). An output with a similarity index of 0.8 is enough to pass the tests for `--help` and `--version`.

- Tests for the `stdin` mode were not present in the previous run of COMP61511 either. These tests are done through the SUBPROCESS library [6], which allows a Python script to execute a process through the command line and capture its output. The library also allows the tests to inject input data to a sub-process, in those cases where data would normally be typed by the user (see examples in 3.2). This input injection is used to simulate the behaviour of the `stdin` mode using the student's `wc.py`, and therefore to test its behaviour automatically.

4.2.6 Week 5

Discussion of CW4 results: performance

This is the final week of the course. CW4, which included the last bits of functionality of WC, is due before this last lecture, so at this point students are expected to have a version of the programme that is as close as possible to the original tool. This gives the opportunity to analyse and discuss how a Python clone performs compared to WC.

For this purpose, students could use the Unix utility TIME [33], that invoked before a command measures the time needed for that command to execute. How well the `wc.py` programme performed is not among the marking metrics, since speed of execution does not represent a relevant indicator of the programme's correctness. Also, a programme of the size and complexity of WC is intended as a tool that takes a few seconds at most to produce an output, so performance is not a critical attribute in this matter. However, since the final result should be a clone of an existing programme, it is interesting to investigate the differences in their behaviours, and try to understand the reasons why if there are any.

CW4 required students to run the TIME utility on their programme using combinations of input parameters and types of files of various sizes, compare those with the results for WC (e.g. see Listing 4.1) and then draw conclusions about their observations in a short essay. For example, `wc.py` usually runs slower than WC, and students should notice that one of the reasons behind this, apart from their programme's complexity, is the fact that interpreted programming languages like

[5]https://docs.python.org/3.6/library/difflib.html

[6]https://docs.python.org/3/library/subprocess.html

Python are generally slower than compiled languages like C, the language WC is written in.

In the previous runs of the coursework, students were only asked to produce an essay explaining their observations: it would be interesting to discuss the results in class as well, so that everybody can share their reasoning and possibly draw further conclusions from multiple experiences.

```
$ time python wc.py test-file.txt
    362    1689    15713   test-file.txt
real   0m0.545s
user   0m0.234s
sys    0m0.091s
$ time wc test-file.txt
    362    1689    15713   test-file.txt
real   0m0.019s
user   0m0.002s
sys    0m0.006s
```

Listing 4.1: Use of the TIME utility to measure performance of *wc.py* against WC.

4.3 Implementation of the assessment tools

This section includes an extended technical description of the tools that were developed to assess the student submissions, according to the criteria that we introduced in the "Evaluation and Feedback" parts of the previous section.

The assessment is performed in multiple phases, which we describe below: first, submissions need to be downloaded from BlackBoard, the submission manager, and stored in a local directory, divided by assignment number; then, a script pre-processes the archives to exclude late submissions and extracts the archives. Finally, the actual assessment tools process the submissions one by one, run tests and compute marks, which can eventually be uploaded to BlackBoard.

4.3.1 Select last submissions before deadline

As previously explained (see 3.6), in case multiple submissions are allowed for some coursework, BlackBoard does not allow the instructor to download only those submissions that are the latest before the given deadline. The whole set of submissions can be downloaded, but then the instructor must select the valid ones manually. The purpose of the first script is to accomplish this task automatically, since each submission's attempt date can be derived by its file name, then the latest valid submissions are straightforwardly selected by comparing the date and the deadline.

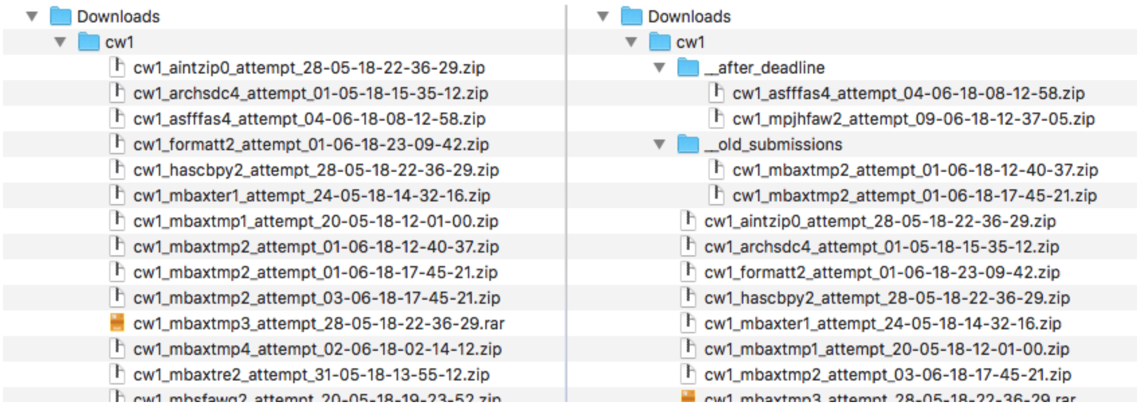
The script takes as arguments a directory containing a list of submission archives and a deadline date and time. The submissions are then processed one by one and their date (parsed from the archive name, which is created by BlackBoard when the submissions are downloaded) is compared with the chosen deadline: any late submissions are moved to a separate directory named `_after_deadline/`. Those will be then graded according to the University's guidance on late submission [7].

Subsequently, the tool searches for multiple submissions by the same username (e.g. *mbaxtmp2*). If some are found, the two submission dates are compared: the most recent stays in the main directory; any older one is moved to `_old_submissions/`. Mainly for testing purposes, but also because the exact deadline for future submissions is not known at the moment, or it may change, the script resets the target directory at the beginning of each run. Also, in case there are multiple submissions from the same student and only one of those is late, the latest submission before the deadline is preserved and then evaluated as "punctual".

Listing 4.2 and Figure 4.2 show a run of the script for CW1, with 3rd June 2018 at 23:59 as deadline, and the content of the target directory before and after.

```
$ python select_last_submissions_before_deadline.py Downloads/cw1/ 3-6-18 23:59
cw1_asfffas4_attempt_04-06-18-08-12-58.zip : Submission after deadline (03 Jun 18 23:59).
cw1_mpjhfaw2_attempt_09-06-18-12-37-05.zip : Submission after deadline (03 Jun 18 23:59).
Multiple submissions found for mbaxtmp2.
```

Listing 4.2: Invocation of *select_last_submissions_before_deadline.py*



```
▼ Downloads
  ▼ cw1
      cw1_aintzip0_attempt_28-05-18-22-36-29.zip
      cw1_archsdc4_attempt_01-05-18-15-35-12.zip
      cw1_asfffas4_attempt_04-06-18-08-12-58.zip
      cw1_formatt2_attempt_01-06-18-23-09-42.zip
      cw1_hascbpy2_attempt_28-05-18-22-36-29.zip
      cw1_mbaxter1_attempt_24-05-18-14-32-16.zip
      cw1_mbaxtmp1_attempt_20-05-18-12-01-00.zip
      cw1_mbaxtmp2_attempt_01-06-18-12-40-37.zip
      cw1_mbaxtmp2_attempt_01-06-18-17-45-21.zip
      cw1_mbaxtmp2_attempt_03-06-18-17-45-21.zip
      cw1_mbaxtmp3_attempt_28-05-18-22-36-29.rar
      cw1_mbaxtmp4_attempt_02-06-18-02-14-12.zip
      cw1_mbaxtre2_attempt_31-05-18-13-55-12.zip
      cw1_mbsfawg2_attempt_20-05-18-19-23-52.zip
```

```
▼ Downloads
  ▼ cw1
    ▼ _after_deadline
        cw1_asfffas4_attempt_04-06-18-08-12-58.zip
        cw1_mpjhfaw2_attempt_09-06-18-12-37-05.zip
    ▼ _old_submissions
        cw1_mbaxtmp2_attempt_01-06-18-12-40-37.zip
        cw1_mbaxtmp2_attempt_01-06-18-17-45-21.zip
      cw1_aintzip0_attempt_28-05-18-22-36-29.zip
      cw1_archsdc4_attempt_01-05-18-15-35-12.zip
      cw1_formatt2_attempt_01-06-18-23-09-42.zip
      cw1_hascbpy2_attempt_28-05-18-22-36-29.zip
      cw1_mbaxter1_attempt_24-05-18-14-32-16.zip
      cw1_mbaxtmp1_attempt_20-05-18-12-01-00.zip
      cw1_mbaxtmp2_attempt_03-06-18-17-45-21.zip
      cw1_mbaxtmp3_attempt_28-05-18-22-36-29.rar
```

Figure 4.2: Submission directory before and after running the script.

[7] http://documents.manchester.ac.uk/display.aspx?DocID=29825

4.3.2 Unzip submissions

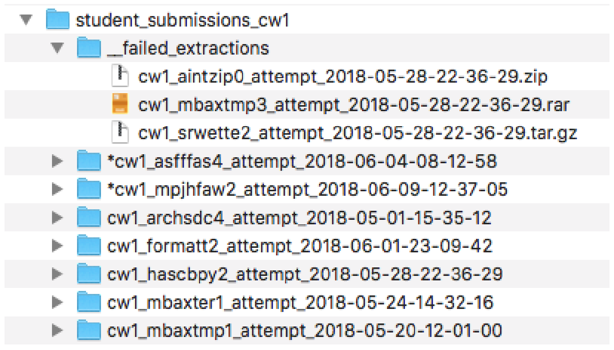
The script called `unzip_submissions.py` takes as arguments a submission archive (or a list of archives) and an optional target directory, in which the archives will be extracted (the default directory is `student_submissions_cwX/` where the number X is between 1 and 4, according to the stage of the coursework). The archives were downloaded from BlackBoard and are typically named with the following classification: the coursework id, the student's username and the attempt date and time (in the format `yyyy-mm-dd-HH-MM-SS`), plus the name that the students gave to their submissions (again the coursework id and their username, as requested by the instructor). For example, `cw1_mbaxtmp2_attempt_2017-10-24-16-44-06_mbaxtmp2_cw1.zip`

The tool performs an additional check of the archive format: if the submission could not be extracted for some reason (the student used some compression algorithm different from `zip`, such as `rar` or `tar.gz`, or the `zip` is corrupted), the archive is moved to the directory `_failed_extractions/` to allow a manual inspection.

The `unzip` script is meant to be run after `select_last_submissions_before_deadline.py`, which excludes from extraction the old submissions and prepares the late ones to be evaluated accordingly. Listing and Figure 4.3 show how the script is invoked and the resulting directory.

```
$ python unzip_submissions.py Downloads/cw1/*
Files in source directory 'Downloads/cw1/': 14
===========================================================================
*** cw1_mbaxtmp3_attempt_28-05-18-22-36-29.rar is not a zip file.
* Unzipping cw1_hascbpy2_attempt_28-05-18-22-36-29.zip
*** Downloads/cw1/cw1_aintzip0_attempt_28-05-18-22-36-29.zip is not a zip file
* Unzipping cw1_mbaxtmp1_attempt_20-05-18-12-01-00.zip
* Unzipping cw1_archsdc4_attempt_01-05-18-15-35-12.zip
*** cw1_srwette2_attempt_28-05-18-22-36-29.tar.gz is not a zip file.
  ...
* Unzipping cw1_mpjhfaw2_attempt_02-06-18-09-14-48.zip
===========================================================================
Archives successfully extracted in target directory 'student_submissions_cw1': 11
Bad zip files: ['cw1_mbaxtmp3_attempt_28-05-18-22-36-29.rar', 'cw1_aintzip0_attempt_28-05-...
These were moved to failed_extractions/ and will require manual inspection.
```

Listing 4.3: Invocation of *unzip_submissions.py*



```
▼ student_submissions_cw1
    ▼ __failed_extractions
        cw1_aintzip0_attempt_2018-05-28-22-36-29.zip
        cw1_mbaxtmp3_attempt_2018-05-28-22-36-29.rar
        cw1_srwette2_attempt_2018-05-28-22-36-29.tar.gz
    ▶ *cw1_asfffas4_attempt_2018-06-04-08-12-58
    ▶ *cw1_mpjhfaw2_attempt_2018-06-09-12-37-05
    ▶ cw1_archsdc4_attempt_2018-05-01-15-35-12
    ▶ cw1_formatt2_attempt_2018-06-01-23-09-42
    ▶ cw1_hascbpy2_attempt_2018-05-28-22-36-29
    ▶ cw1_mbaxter1_attempt_2018-05-24-14-32-16
    ▶ cw1_mbaxtmp1_attempt_2018-05-20-12-01-00
```

Figure 4.3: Resulting *student_submissions_cw1/* directory after running the *unzip* tool. The directories with a * at the beginning are the late submissions.

4.3.3 Test and grade submissions

Following the coursework structure described in Chapter 4.2, four assessment scripts were developed for the corresponding assignments, from CW1 to CW4.

The structure of the four programmes is very similar. There is a main Python script for each part of the coursework, whose purpose is to process a set of submissions and compute marks for all grading criteria. These criteria are defined inside a configuration file composed in the JSON format.

Configuration via JSON file

The primary means of configuration and customisation of the grading scripts are a set of four JSON [8] files. Each file specifies data for the corresponding assignment, for example the required files, the Python packages that may be used, the points to be given for each parameter and pairs of input and expected output to be used as test cases.

The tools read the values from the JSON and use them to compute the corresponding grades (see Listing A.9). For instance, the tool for grading CW1 checks that the required files – the script `miniwc.py`, the test script `doctest_wc.py` and a folder with input test files – are included in the submissions and that `miniwc.py` only used the module `sys`, and gives points as specified in the section *marks_structure*. Subsequently, it runs the script with the tests listed in the *output* section (e.g. `python miniwc.py test_files/100B_file.txt`) and verifies `miniwc.py` computed the counts correctly. The points given for each category (text, binary or unicode files) are proportional to the amount of tests passed.

[8] JavaScript Object Notation or JSON is an language-independent file format for storing and transmitting data objects, which consist of pairs with an attribute and a value: http://www.json.org. See Listing A.9 as an example.

```
{
'required_files': {
    'script'      : "miniwc.py",
    'test_script': "doctest_miniwc.py",
    'tests_dir'   : "test_files"
  },
  'allowed_modules': ["sys"],
  'marks_structure': {
    'script_present': 0.5,
    'tests_present': 0.5,
    'required_libraries': 1
  },
  'marks_correctness': {
    'correct_formatting': 1,
    'text': 5,
    'binary': 1,
    'unicode': 1
  },
  'output': [
    {
      'lines': "0",
      'words': "1",
      'bytes': "100",
      'file': "test_files/100B_file.txt"
    }, ...
}
```

Listing 4.4: Configuration JSON file for CW1.

Refer to A.10 for another example of a JSON configuration files, written for CW3.

Grading scripts

There are four scripts called *test_and_grade*, one for each assignment. A grading script takes a set of submission directories as command line argument. The directories are created by the script `unzip_submissions.py` and are named `student_submissions_cw[1-4]/` by default. A submission directory follows the nomenclature described in section 4.3.2, so non-submission folders are easily identifiable.

Each grading script typically checks that the submissions specified are valid directories (e.g. it skips the generated directories like `_failed_extractions/`), it parses the username and the attempt date from the directory name, then it calls the functions that will compute all the marks specified in the JSON file, keeping separated the "**structure**" from the "**correctness**" marks. Eventually, it gathers all computed marks, prints and stores them into a local database, together with the corresponding username, attempt date and sum of all marks. Listing 4.5 represents the results printed to the console after running the analysis described.

```
$ python test_and_grade_cw1.py student_submissions_cw1/*
===========================================================================
*** Checking content of submission
STUDENT_SUBMISSIONS_CW1/CW1_MBAXTMP2_ATTEMPT_2018-06-01-17-45-21 on 2018-08-02 19:35:31
The main script 'miniwc.py' seems to exist.
The test script 'doctest_miniwc.py' and input test files seem to exist.
The submission contains all the required files for cw1.
Only ['sys'] has been used.
The output format seems to be correct for a test file. Proceeding with the tests...
Apparently the script does NOT handle binary files.
---------------------------------------------------------------------------
*** Statistics for the script (miniwc tests):
Correct line counts:     8 \ 20,        40.0%
Correct word counts:    16 \ 20,        80.0%
Correct byte counts:    17 \ 20,        85.0%
Correct file names:     17 \ 20,        85.0%
Total correct tests:     7 \ 20,        35.0%
Errors observed in files:
* off-by-one-line: ['CW1/cw1_test_files/100B_file.txt', 'CW1/cw1_test_files/10KB_file.txt', ...]
---------------------------------------------------------------------------
*** Statistics by file type (standard text, binary, unicode spaces):
Total text files:     14       Correct:        7,      50.0%
Total binary files:    3       Correct:        0,       0.0%
Total unicode files:   3       Correct:        0,       0.0%
---------------------------------------------------------------------------
Marks computed: {'script_present': 0.5, 'tests_present': 0.5, 'required_libraries': 1,
'coverage': 0, 'correct_formatting': 1, 'text': 3, 'binary': 0, 'unicode': 0}
___________________________________
Total | Penalties | Suggested grade
  6.0 |         0 |      6.0 /  10.0
---------------------------------------------------------------------------
Record added to database.
===========================================================================
*** Checking content of submission
STUDENT_SUBMISSIONS_CW1/CW1_MPJHFAW2_ATTEMPT_2018-06-02-09-14-48 on 2018-08-02 19:35:45
The submission contains all the required files for cw1.
...
===========================================================================
Done.
```

Listing 4.5: Invocation of *test_and_grade_cw1.py* and output produced for the submission by the user *mbaxtmp2*.

As already mentioned, the general grading script computes marks concerning the "structure" and the "correctness" of the submissions, which are also defined in the JSON configuration. This distinction is to separate the two different parameters adopted to assess the submissions, mainly to verify whether the student understood and implemented the requested architecture for his/her submission, and of course if the code he/she developed produces the desired output. Thinking about any future changes in the coursework, it is useful to distinguish the two aspects, also to prevent the whole script from becoming excessively long and complex.

Submission structure

The criteria that `test_structure.py` assesses are:

- the Python packages and libraries imported in the script (some existing libraries and code sources would oversimplify the student's work, whereas we want them to elaborate their own solutions) – only `sys` for CW1, other like `doctest` and `argparse` for the following assignments;
- the files and directories required for the submissions are present – the `wc` or `miniwc` script, the test script (DOCTEST or UNITTEST), a set of input files for the said test suites;
- the line coverage of the tests is adequate – the ranges can be personalised, but a good test suite should cover at least 80-90% of the lines of code;
- other parameters that vary with each assignment, such as the amount and the size of input files for testing and the number of DOCTEST cases – useful to give a *prima facie* indication of the effectiveness of the student's tests.

Submission correctness

`test_correctness.py` marks the submissions with the following criteria:

- The formatting of the output is correct up to space.
  In Chapter 2 we described various approaches on how to evaluate the output correctness of some programme. For example, ASSYST [13] used a pattern matching system based on a grammar and a lexicon, which allows to parse the output and compare it to the expected one while maintaining a level of flexibility. This approach proved not necessary for an assignment like WC: the programme has a fixed output order (e.g. it prints out the count of the lines, then the words, etc. regardless of the order of the flags), therefore in this case it is much easier to just split the parsed output whenever a space character (including tabs) is found and then compare the single numeric values. If a student printed the counts in any different format, for example on separate lines, or forgot the file path at the end of the line, the tool considers it as incorrect, since it means the student misunderstood the specifications or did not use WC as a reference for reverse-engineering.
- The number of tests passed.
  Each assignments has different specifications (as described in section 3.3) and each JSON file includes a list of test cases, consisting of input flags and files and expected output, that

check if the requirements have been met. For each test case, the tool executes the student programme and stores the output, then it compares the counts.
The correctness criteria for CW1 only refer to the `miniwc` script and have separate marks for plain text, binary and unicode input files. Starting from CW2, the tests cover possible inputs for wc (combination of flags and files), up to CW3 and 4 that feature separate marks for the `miniwc` and `wc` functionalities, for the flags `--files0-from`, `--help` and `--version` and for the `stdin` mode.

4.3.4 Task automation

PyDoit [54] is a Python library whose purpose is to define automated tasks that will run with a simple console command.

Tasks are defined in a script named `dodo.py`. The example in listing 4.6 shows the definition for a task that will extract CW1 submissions (located in `Downloads/cw1`), through the usual command defined in the "actions".

If a task depends on another task (e.g. the grading script must be executed after the submissions have been extracted by `unzip_submissions.py`, which comes after the late/old submissions have been filtered out), the priority of the tasks can be specified in the `dodo.py` through the annotation `@create_after`.

```
@create_after(executed='last_submissions_cw1')
def task_unzip_cw1():
    for zip, task_name, extract_dir in [(f, f.split('_')[1] + "__" + f.split('_')[3],
                                         f.split('.')[0]) for f in listdir('Downloads/cw1')]:
        yield {
            'name': task_name,
            'actions': ['python unzip_submissions.py Downloads/cw1/%s' % zip],
            'targets': [Path("student_submissions_cw1") / extracted_dir]
        }
```

Listing 4.6: Sample task definition for extracting CW1 submissions.

A relevant advantage of using PyDoit versus invoking the single grading scripts is that the library allows the developer to define specific "file dependencies" for each task: if none of these files changed from the latest run, the tasks will not be executed again. In this case, we wanted the grading script to test the submission again only if the script or the penalty files were modified since last assessment (e.g. the submission was marked as "bad" by the grading tool because it was written for Python 2, which caused a compile error. The instructor fixed the `wc.py` script to be run with Python 3 and decided to add a 1-point penalty), so setting `wc.py` and `penalties.json` as dependencies for the grading tasks accomplishes this result.

Similarly, specifying files or directories as "targets" makes the process skip a task if the said targets already exist. For example, we do not want the unzip script to run again if the target archive has already been extracted: if a submission archive is called `Downloads/cw1_mbaxtmp2.zip`, we can prevent the command `unzip_submissions.py cw1_mbaxtmp2.zip` from extracting it again by adding `‘‘student_submissions_cw1/cw1_mbaxtmp2/’’` (i.e. the path to the extracted directory) as "target" of the unzip task.

Listings 4.7 and 4.8 show chunks of the output produced by the `doit` command (used to execute the tasks defined in `dodo`) twice: the second time, the unzip and grade tasks are not run again, as the `--` symbol instead of a `.` indicates. When running `doit`, the execution order follows the priorities defined in the configuration script.

```
$ doit
.  last_submissions_cw1
.  unzip_cw1:mbaxtmp1__2018-05-20-12-01-00.zip
.  unzip_cw1:mbaxtmp2__2018-06-01-17-45-21.zip
.  unzip_cw1:mbaxtre2__2018-05-31-13-55-12.zip
...
.  grade_cw1:mbaxtmp1__2018-05-20-12-01-00
.  grade_cw1:mbaxtmp2__2018-06-01-17-45-21
.  grade_cw1:mbaxtre2__2018-05-31-13-55-12
...
.  last_submissions_cw2
.  unzip_cw2:mbaxtmp2__2018-06-19-12-52-15.zip
...
.  grade_cw2:mbaxtmp2__2018-06-19-12-52-15
...
```

Listing 4.7: Invocation of *doit*: tasks marked with a `.` are executed.

```
$ doit
.  last_submissions_cw1
-- unzip_cw1:mbaxtmp1__2018-05-20-12-01-00.zip
-- unzip_cw1:mbaxtmp2__2018-06-01-17-45-21.zip
...
-- grade_cw1:mbaxtmp1__2018-05-20-12-01-00
-- grade_cw1:mbaxtmp2__2018-06-01-17-45-21
...
```

Listing 4.8: Second invocation of *doit*. `--` indicates that the tasks was not re-run because the file dependencies did not change, or some target already existed.

4.4 Code stubs

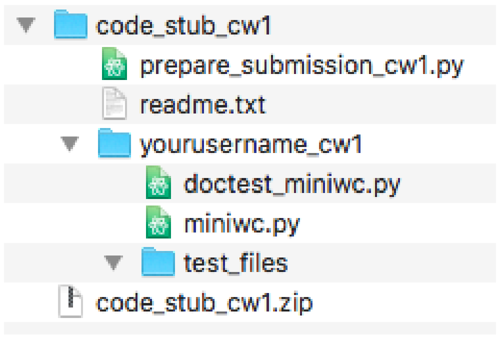
In software engineering, a stub is a piece of code that exists as a temporary substitute for some functionality that has not been developed yet. In our case, stubs consist of a set of files that will give students a guidance on how to structure their submissions.

For example, a submission stub for CW1 is composed of a directory that the student will rename with his/her username, which includes the `miniwc.py` script, the test suite as `doctest_miniwc.py` and a directory that will contain input test files.

Code stubs will be available for students to download from BlackBoard when the requirements of a new coursework are available. It is not mandatory to use such stubs as a starting point, as following carefully the specifications would lead to the same result in terms of structural correctness. However, we recommend students to use it, especially with the first stage of the coursework: results from past runs of COMP61511 show that errors like a misnamed `wc` script or a missing test file were very frequent at the beginning, presumably when students were still getting familiar with a new model for explaining requirements. Although understanding specifications is an important skill that any future software engineer should master, we believe this represents a valuable aid to reach this objective if not already acquired.

Also, for example, code stubs for `miniwc.py` would already include the specific messages for the errors "More than one file" or "Stdin mode not yet available".

Figure 4.4 shows the stub archive as it would look like once downloaded from BlackBoard and extracted.



```
▼ code_stub_cw1
      prepare_submission_cw1.py
      readme.txt
   ▼ yourusername_cw1
         doctest_miniwc.py
         miniwc.py
      ▼ test_files
   code_stub_cw1.zip
```

Figure 4.4: Example of files contained in a code stub.

The archive also includes a file called `prepare_submission.py`, whose functionality is described in the following section.

4.5 Preparation scripts

Preparation scripts represent another aid to students, for them to reach the highest structural correctness possible. Such scripts consist of a function callable from the command line that performs various checks on the student's submission.

The students can call the function from their own machine with their username and the submission directory as arguments. The tool will then check that:

- the directory contains all required scripts, including test files;
- some changes were made to the script files, i.e. the scripts do not look like the model files in the submission stub;
- the WC script compiles.

If all conditions are satisfied, the preparation script will then compress the submission directory into a new archive, with the student's username and the coursework id in the file name.

As mentioned previously, the purpose of the preparation scripts is only to guarantee the students have a submission that is structurally reasonable, and at the moment they are not meant to test that the programme behaves as expected. In section 4.7 we discuss the opportunity to provide a preliminary feedback when running the preparation script already.

Listing 4.9 shows the execution of a preparation script for CW1 and the console output that it produces if the submission is reasonable from the structural point of view.

```
$ python prepare_submission_cw1.py mbaxtmp2 yourusername_cw1/
* The submission contains all required files: ['miniwc.py', 'doctest_miniwc.py', 'test_files'].
* test_files/ appears to contain some files.
Creating zipped file called mbaxtmp2_cw1.zip for submission.

Your submission seems reasonable. Note that we did not check the correctness of any file,
just that it existed and/or was different from the stubs.
```

Listing 4.9: Invocation of *prepare_submission_cw1.py*

4.6 Results and Grade analysis

After the submissions have been assessed and a grade has been assigned to them, the next phase involves uploading the computed grades back to BlackBoard, so that students can view them. The tools also offer the option to save results to a local database, which can be used to analyse students' progress or the class performance.

4.6.1 Report and grade upload

While running, each evaluation script produces a report log for every submission: it contains all information about successful tests and marks computed (the same information printed to the console while running the tools, as in Listing 4.5).

Listing 4.10 shows an example of a result log for a CW4 submission.

```
===========================================================================
*** Checking content of submission STUDENT_SUBMISSIONS_CW4/CW4_EJASCBU7_ATTEMPT_2018-07-03-13-49-16/
   on 2018-08-09 17:59:13
The submission contains all the required files for CW4.
* The main script 'wc.py' seems to exist.
* Test script(s) seem to exist.
Only ['sys'] has been used.
Apparently the script handles binary files.
---------------------------------------------------------------------------
*** Statistics for the script (miniwc tests):
Correct line counts:   20 \ 20,        100.0%
Correct word counts:   17 \ 20,        85.0%
Correct byte counts:   20 \ 20,        100.0%
Correct file names:    20 \ 20,        100.0%
Total correct tests:   17 \ 20,        85.0%
---------------------------------------------------------------------------
*** Statistics by file type (standard text, binary, unicode spaces):
Total text files:      14      Correct:        13,     92.9%
Total binary files:    3       Correct:        1,      33.3%
Total unicode files:   3       Correct:        3,      100.0%
---------------------------------------------------------------------------
The doctest tests reached 50.0% line coverage for wc.py: 1/2 points.
The script passed 85.0% of miniwc tests: 1/2 points
The script passed 95.2% of wc tests: 6/7 points
The script passed 100.0% of files0-from tests: 2/2 points
The script passed 100.0% of help/version tests: 2/2 points
The script passed 86.7% of stdin tests: 2/3 points
---------------------------------------------------------------------------
Marks computed:
{'script_present': 0.5, 'doctest_present': 0.5, 'unittest_present': 0.5,  'required_libraries': 0.5,
'coverage': 1, 'miniwc_tests': 1, 'wc_tests': 6, 'files0-from': 2,  'help-version': 2, 'stdin': 2}
____________________________________
Total | Penalties | Suggested grade
 16.0 |         0 |   16.0 /  20.0
============================================================================
```

Listing 4.10: Result log produced for a sample submission for CW4.

We also want students to know the criteria that were used to grade their assignments, so the feedback that students will receive in their BlackBoard account will look similar to this report. BlackBoard accepts data in the `CSV` (comma-separated values) spreadsheet format, therefore the tools will eventually produce one `CSV` file per coursework, containing a list of usernames with the corresponding grade and result log, which the instructor will use to upload grades and feedbacks to the University system.

4.6.2 Database: trends and statistics

We already mentioned that the assessment tools ultimately creates a database record that consists of the submission attempt and grading dates, the student's username, the numeric values of the parameters assessed (e.g. the marks computed, or the percentage of successful tests), the errors observed and other data such as the LOC (lines of code) count in the `wc` script. Specific functions were then created to extract the records from the database and perform analysis on the data: for example, a function selects all records with distinct usernames that have the latest assessment date, which of course represents the most recent grades for each student. It is also possible to extract only the numeric values or the meta-data (LOC counts, number of doctests, the errors,...).

Therefore, the database is meant to be not only a storage for the marks computed, but also an opportunity for the instructor to have an overview of the assignment results. Useful information that can be obtained include:

- Possible problems with the current assignments: for example, if a large number of students passed or failed *all* tests of some category, this could be an index of excessive ease, or difficulty, of the exercises.
- Possible cases of plagiarism or collusion between students: two or more programmes, for instance, failing the exact same tests or containing similar errors might indicate that their authors worked together on the code, or cooperated beyond the allowed limits.
- Tests that many people failed, which may indicate that the class misunderstood some requirements or faced common problems. This could give the instructor and TAs the chance to address specific aspects of the coursework during the lab and give students indications on how to proceed.
- Errors related to test cases failed, or similar patterns of common errors: again, discovering such evidence would help the instructor address common difficulties in the class and perhaps focus more on solving these issues in the following lectures and labs.

Ultimately, the database also stores the text of the feedback that the students will receive, which can be uploaded to BlackBoard in a suitable CSV file. The following sections describe what these feedbacks will consist of and what modalities may be used to allocate them.

4.7 Feedback models

During the phase of discussion between the author and the course instructor regarding the type and the timing of feedback to be returned to students, some models emerged that are described below. The choice of what model to implement is obviously up to the teacher. The advantages and disadvantages of each model will be listed here, so that maybe both can be used alternatively in order to test their effectiveness in a real classroom context.

4.7.1 Single attempt model

This model was used in the previous runs of COMP61511 and, intuitively, it offers students a single attempt to submit their coding exercise within the given deadline. If a student fails to upload a submission, or uploads it after the deadline, he/she will be awarded 0 points (or his/her mark will be reduced according to the late submission policy in effect at the time). Except for an early feedback on the submission structure that the student can obtain by running the `prepare_submission` tool, only one round of feedback will be given together with the final grade.

Listing 4.11 shows what a sample feedback for a CW4 submission would look like.

```
There seems to be a wc.py: 0.5/0.5 points.
There seems to be a doctest_wc.py: 0.5/0.5 points.
There seems to be a unittest_wc.py: 0.5/0.5 points.
Only allowed libraries have been used: 0.5/0.5 points.
The doctest tests reached 50.0% line coverage for wc.py: 1/2 points.
The script passed 85.0% of miniwc tests: 1/2 points.
The script passed 95.2% of wc tests: 6/7 points.
The script passed 100.0% of files0-from tests: 2/2 points.
The script passed 100.0% of help-version tests: 2/2 points.
The script passed 86.7% of stdin tests: 2/3 points.
Penalties: none.
Total marks: 16.0/20.0
```

Listing 4.11: Feedback that a student would receive for his/her CW4 submission.

4.7.2 Multiple attempts model

In this model, students can upload a coding submission multiple times within the deadline.

If multiple submissions are allowed, the tool we have developed that filters out submission archives that are late or that have valid more recent attempts comes in handy, since BlackBoard does not offer an option to select the latest valid submissions at download time. This would be the case where a single round of feedback is given for everybody at the end. Alternatively, the students would receive feedback every time they submit their code.

Of course, the second option would made sense if the feedback was given immediately, or after a short time after the submission: at the current stage of the grading tools, this is not feasible, as the assessment process needs to be started manually and this would require the instructor and TAs to check continuously for new submissions. A working Continuous Integration system, as described in Section 6.4.1, would of course do the job efficiently. However, since it is not available yet, below we suggest an alternative model for multiple submissions, which limits the amount of attempts the students can make and schedules accurately the feedback rounds.

4.7.3 Pay-for-hint model

It is intuitive, and studies have shown, that the students who received higher grades for a programming task were generally the ones who started and finished it as early as possible [55].

The model suggested here is a variation of the multiple attempts one, except it does not give feedback "for free": students may submit their exercise a few days before the deadline, receive an early feedback and expected mark, then may decide to make changes to their code and submit a second time, but this time they will receive half of the points they would be given otherwise.

This scheme simplifies how a sample course week would take place, where Day 0 represent the lecture and lab day:

- **Day 0: lecture + lab**. Students can download a stub with the needed files and directories. A `prepare_submission` script is included, which gives an early feedback on the structure (some files are missing, `miniwc.py` was not modified, the programme does not run or raises a compile error...).

- **Day 0 to Day 5**: students work on the assignment autonomously and early submissions must be done by 5 pm to receive feedback the following morning. Students can submit early in the week, but they can receive such feedback from Day 6 only. Students may also decide not to exploit the early submission and submit directly at the end of the week.

- **Day 6**: feedback as "You passed X% of the tests" are released in the morning (e.g. by 10 am) for submissions done by 5 pm on Day 5. Students can either keep their mark (with no need to re-submit), or have the chance to submit again by Day 7 at 9 am and get ½ for any extra point they would receive. For example, if a student obtained 6/10 points for Attempt-1 and would get 3 more points for Attempt-2, the total points will be instead 7.5/10 (6 + ³⁄₂).

- **Day 7 (Day 0 of the following week)**: final submission due by 9 am. The second round of feedbacks should be available at the beginning of the afternoon lab. The feedback should contain the marking criteria and the "categories" of the tests passed and failed (e.g. off-by-one error, the byte counts fails, the script doesn't read binary files correctly...). The students can then discuss their result with a TA if they wish so. During the lecture and lab the new assignment is presented.

Some problems may result from this model. Students must choose between working completely on their own and receiving feedback for half extra points, which means spending potential marks for a "hint". For example, objections that students might rise are "I submitted early because you told me so: I could have figured that out, so I should receive full points" or "The intermediate deadline put me in a hurry and I believe I missed some marks because of that. It is not fair that I received half points for something I could have done smoothly in my own time".

Such arguments about the fairness of the grading criteria should of course be avoided by clearly explaining the model beforehand. In any case, hints are meant to be general indications and represent a chance to get a few more points that the student would have probably missed, so there should not be much room for complaints, after making sure that the students have understood the criteria.

Also, releasing indications about the correctness of the coding assignments, even if only in form of percentage of passed tests, may introduce the chance of collusion, for instance if a student that scored poorly on his first attempt asked a colleague to fix his/her broken code.

On the other hand, as already mentioned, this model includes some effective strategies to motivate students to start coding as early as possible, and also to help the weaker students to understand their errors or misunderstandings and offer them a chance to recover, so the instructor may consider it despite some drawbacks it involves.

Chapter 5

Testing and evaluation

5.1 Chapter overview

The grading tools developed for this project went through a process of verification and validation that is outlined in this chapter.

5.2 Validation

Validating a software product means ensuring that it meets the needs of a customer or other stakeholder [56]. The aim of the software validation process is to answer the question "are we building the right product?" [57].

For this project, frequent meetings were organised with the course instructor, who is going to be the primary user of the system: all requirements for the grading tools developed in this work were established according to either the existing model of COMP61511 or the instructor's needs and preferences for new assessment features to be integrated. The project was structured from the beginning in a way that would allow the grading tools to be extended with new small components, time permitting (also assuming the instructor will be able to integrate new features into the system). However, all the main functionalities, that is tools for testing and computing grades for the students' programming assignments according to the WC specifications (Section 4.2), have been developed, and the main requirements satisfied.

5.3 Verification

Verifying a product means ensuring that it complies with a set of defined requirements and conditions [56]. The process of software verification aims to answer the question "are we building the product right?" [57] – in other words, does the product behave as expected?

More details about the phases and processes performed to verify the grading tools are provided in the following sections. In particular, the measures to test the software product in the three stages of creation – development, release and user testing – will be described.

5.4 Development testing

Development testing concerns all tests that are performed as the code is being developed, in order to identify bugs and defects [57]. Tests were performed as the various grading functions were being built, in order to reach the highest coverage possible.

This section addresses specific techniques that were adopted for testing – unit, integration and system testing.

5.4.1 Single marking methods - Unit testing

Unit testing means "Testing of individual hardware or software units or groups of related units" [58]. To better organise the various marking functions, which are often repeated among the assignments, and also to allow further functions to be added easily, these were divided into units, or small components, of software, and were aggregated in a single script named `utilities.py`. The individual components can then be tested using various input parameters [57].

About fifty of such functions are contained in `utilities` at the moment: these methods execute elementary tasks such as getting the content of a directory as a list, computing the percentage between two numbers and parsing and converting a date string from a directory name. More complex sets of tasks that are specific to our grading tools are also present. Examples include:

- saving the console output of a command;
- checking the content of a submission and comparing it to the required files of some coursework, to verify if any of those is missing;
- checking that the `miniwc` or `wc` output is in the correct format;
- parsing the modules and libraries imported in a script and checking if any of those is not allowed for the current coursework;
- computing the amount of points to be given for a percentage of tests passed, according to the grading scale defined by the instructor;

- printing the statistics of passed and failed test cases in a readable format and saving them to a log file.

Despite their more or less manifest complexity, these functions are all focused on a single goal or at most a set of close goals, so that the testing activity can also be targeted to a small set of tasks. All tests for the said functions are included in a script named `unittest_utilities` and for each function extensive tests have been run, by calling it with different input parameters.

Consider, for example, the function that checks if some WC output was printed in the correct format: the function takes a string as input and returns True if the string corresponds to a valid `wc` or `miniwc` output, or False otherwise. The format an valid `wc` output is like "`number number number [...] string`" and the values can be separated by single-character spaces, tabs or other kinds of space. Note that this function merely checks the format and does not test if the counts are correct or the file exists.

A set of test cases for the function `is_format_correct()` should cover different types of output to show that it performs a correct check. Listing 5.1 displays a set of input values used to test the function and presents different formats of valid inputs.

```
def test_is_output_format_correct(self):
    (1) self.assertTrue(is_format_correct("\t33\t106\t907\ttests_for_test_cw1/miniwc.py\n"))
    (2) self.assertTrue(is_format_correct("33 106 907 tests_for_test_cw1/miniwc.py"))
    (3) self.assertTrue(is_format_correct("\t0\t0\t0\tsome-file\n"))
    (4) self.assertTrue(is_format_correct("\t12\tsome-file\n"))
    (5) self.assertTrue(is_format_correct("\t12\46\t1266\t1266\t235\tsome-file\n"))
    (6) self.assertTrue(is_format_correct("\t0\t0\t0\tsome-file\n\n"))
    (7) self.assertFalse(is_format_correct("33\n12\n234\nfilename\n"))
    (8) self.assertFalse(is_format_correct("33\t21\t234\t23\n"))
    (9) self.assertFalse(is_format_correct("x\t12\t234\t23\n"))
    (10) self.assertFalse(is_format_correct("Output: 33 21 423 filename"))
    (11) self.assertFalse(is_format_correct("Error: file not supported"))
    (12) self.assertFalse(is_format_correct(""))
    (13) self.assertFalse(is_format_correct_miniwc("33 12 234 87 filename"))
    (14) self.assertFalse(is_format_correct_miniwc("33\t24\tfilename\n"))
```

Listing 5.1: Unit test cases for the function `is_output_format_correct(outputstring)`

Tests 1-6 show valid `wc` output formats, while the remaining tests cover many formats that do not follow the expected format and are described below.

- (1) tab-separated counts;
- (2) space-separated counts;
- (3) all counts equal to zero;
- (4) output for a run with one flag;

- (5) output for a run with all flags;
- (6) output with a double newline character at the end;
- (7) newline-separated counts, which results in an output printed over multiple lines;
- (8) no file name at the end;
- (9) one of the counts appears as letter instead of a number;
- (10) the output should just replicate `wc`'s and not contain any extra words;
- (11) error messages are treated as wrong output (there is a separate function to check format of error messages);
- (12) empty string;
- (13) there is an extra count for a `miniwc` output;
- (14) there is a missing count for a `miniwc` output;

The code above showed an example of how tests were performed for the unit functions. Ideally, testing should cover all possible input combinations to avert the possibility of errors or unexpected output values, but of course this is not possible in most circumstances. The purpose of unit testing, in this case, is to cover the most common input parameters and explore as many corner cases as possible, especially the ones that might cause errors [57].

All tests for unit functions present in `unittest_utilities` were designed to reach 100% line coverage (only those functions that are used to print out data to the console or a log file did not reach total coverage, due to their nature).

5.4.2 Integration testing

Testing single functions that are limited in size and scope can be done with a fairly straightforward process, which was described in the previous section. However, when a software product is the result of many small functions combined in a single programme, assessing the programme as a whole is also critical to verify if its general behaviour is correct. This process is also known as integration testing [58].

In the case of the grading tools for COMP61511, in fact, the entire grading system is composed of several individual functions (four grading scripts, one for each part of the coursework, one script for filtering out all late submissions and one to extract them, plus a few more auxiliary scripts for extracting data from the database in a more readable format), as outlined in Chapter 4. Details about the integration testing was done for each script, or set of scripts, are provided in the sections below.

Code stubs and prepare submission scripts

A code stub for a `wc` coursework is just a directory that the students can download from BlackBoard and contained the files that are necessary for a complete submission. The scripts contained in the stubs (e.g. `miniwc.py`, `wc.py`, `doctest_wc.py`, ...), once extracted, either are empty files or contain configuration code for the submissions (for example, the block that allows the script to be run directly from the command line [1]) or empty functions for the student to develop. The goal of the prepare submission scripts is then to process the directory and return error/warning messages in case one or more scripts are missing or misnamed, if they appear not to be different from the stub's original ones (meaning the student did not intervene to develop the `wc` function), or if the code failed to compile.

The prepare submission scripts were then verified by running them on various sample submissions to cover all possible logic paths, that is:

- submissions with compile errors in `wc.py`;
- with no main script, test script (doctest or unittest depending on the coursework) and other required files;
- with no test input files inside a `test_files/` directory;
- with misnamed files (e.g. `wc_CW1.py` or `wc_username.py`, or just `doctest.py`);
- with a blank `wc` or `doctest_wc.py`.

Also, since each `prepare_submission` takes a username and the submission directory as parameters, appropriate error messages are displayed if those are not provided.

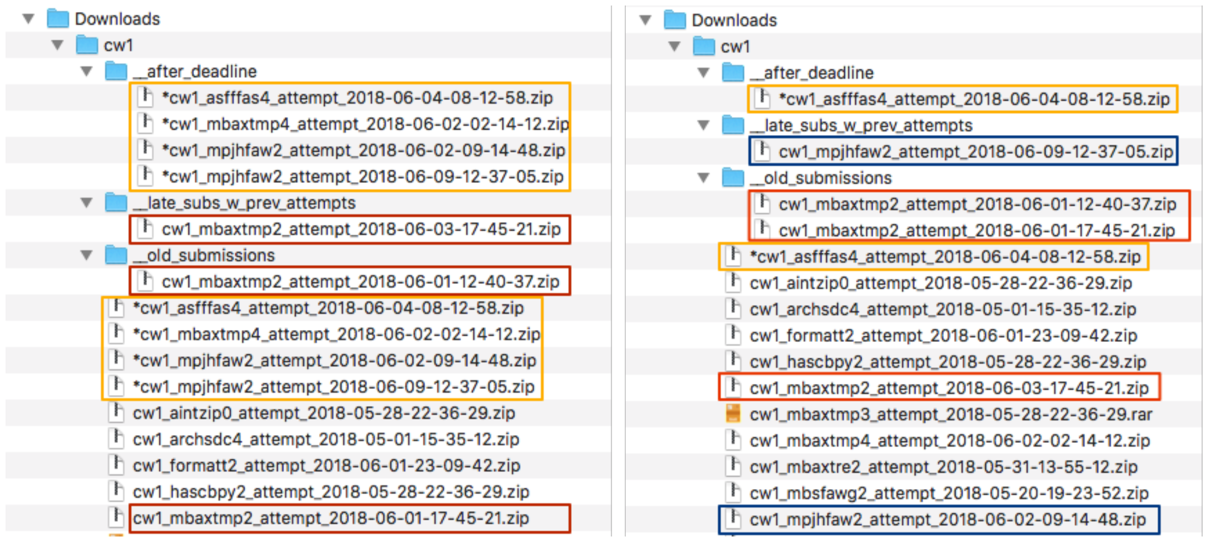
Filtering and unzipping submissions

Whenever the script is executed, `select_last_submission` restores the source directory by putting all archives to the upper level (i.e. for example, moving all content of the temporary folder `Downloads/cw1/__after_deadline/` back to `Downloads/cw1`.

[1] `if __name__ == '__main__': ...`

Since the function accepts a deadline date and time as input parameters, the tests covered various cases to verify that the submissions are classified correctly. For example, a successful execution of the script on a set of submissions (e.g. `Downloads/cw1/`), of which some were sent after the chosen deadline, should eventually produce the following output, which will be then processable by `unzip_submissions`:

- a sub-directory called `Downloads/cw1/__after_deadline/` containing all late submissions (without any valid previous attempts);
- a sub-directory called `.../__late_subs_w_prev_attempts/` containing any late submissions that have a previous attempt (and the main directory `Downloads/cw1/` will contain a submission with the same username for all "late submissions with previous attempts");
- a sub-directory called `.../__old_submissions/`, containing any previous attempts of a submission, which also be present in `Downloads/cw1/` with the same username;
- all content of `__after_deadline/` was renamed with a * at the beginning and was then copied back to `Downloads/cw1/`, so that it can be processed by `unzip_submissions`: the tool will not treat punctual or late submissions differently (`test_and_grade` will, instead).



```
Downloads
  cw1
    __after_deadline
      *cw1_asfffas4_attempt_2018-06-04-08-12-58.zip
      *cw1_mbaxtmp4_attempt_2018-06-02-02-14-12.zip
      *cw1_mpjhfaw2_attempt_2018-06-02-09-14-48.zip
      *cw1_mpjhfaw2_attempt_2018-06-09-12-37-05.zip
    __late_subs_w_prev_attempts
      cw1_mbaxtmp2_attempt_2018-06-03-17-45-21.zip
    __old_submissions
      cw1_mbaxtmp2_attempt_2018-06-01-12-40-37.zip
    *cw1_asfffas4_attempt_2018-06-04-08-12-58.zip
    *cw1_mbaxtmp4_attempt_2018-06-02-02-14-12.zip
    *cw1_mpjhfaw2_attempt_2018-06-02-09-14-48.zip
    *cw1_mpjhfaw2_attempt_2018-06-09-12-37-05.zip
    cw1_aintzip0_attempt_2018-05-28-22-36-29.zip
    cw1_archsdc4_attempt_2018-05-01-15-35-12.zip
    cw1_formatt2_attempt_2018-06-01-23-09-42.zip
    cw1_hascbpy2_attempt_2018-05-28-22-36-29.zip
    cw1_mbaxtmp2_attempt_2018-06-01-17-45-21.zip
```

```
Downloads
  cw1
    __after_deadline
      *cw1_asfffas4_attempt_2018-06-04-08-12-58.zip
    __late_subs_w_prev_attempts
      cw1_mpjhfaw2_attempt_2018-06-09-12-37-05.zip
    __old_submissions
      cw1_mbaxtmp2_attempt_2018-06-01-12-40-37.zip
      cw1_mbaxtmp2_attempt_2018-06-01-17-45-21.zip
    *cw1_asfffas4_attempt_2018-06-04-08-12-58.zip
    cw1_aintzip0_attempt_2018-05-28-22-36-29.zip
    cw1_archsdc4_attempt_2018-05-01-15-35-12.zip
    cw1_formatt2_attempt_2018-06-01-23-09-42.zip
    cw1_hascbpy2_attempt_2018-05-28-22-36-29.zip
    cw1_mbaxtmp2_attempt_2018-06-03-17-45-21.zip
    cw1_mbaxtmp3_attempt_2018-05-28-22-36-29.rar
    cw1_mbaxtmp4_attempt_2018-06-02-02-14-12.zip
    cw1_mbaxtre2_attempt_2018-05-31-13-55-12.zip
    cw1_mbsfawg2_attempt_2018-05-20-19-23-52.zip
    cw1_mpjhfaw2_attempt_2018-06-02-09-14-48.zip
```

Figure 5.1: Resulting output directories after two runs of *select_last_submission* with different deadline dates, 1st June and 3rd June.

Figure 5.1 shows how the `Downloads/cw1` directory should look after running the script `select_last_submission` with two different deadline dates, 1st and 3rd of June. The yellow rectangles indicate the late submissions that have no punctual previous attempt (and therefore will be processed as late submissions, with all applicable penalties).

Notice how submissions from user *mbaxtmp2* (in red) were classified differently in the two cases: in the first (deadline 1st June), the archive with date 2018-06-03 is late, but there are two more submissions attempted on 2018-06-01 that are valid, of which the oldest was moved to `__old_submissions/` and will be skipped.

A similar situation happens in the second execution (deadline 3rd June, at midnight), except the latest submission by *mbaxtmp2* was submitted before the deadline, so it is the one that will be assessed. Similarly, the latest submission by student *mpjhfaw2* (in blue) is late, so the existing previous attempt (2018-06-02) will be assessed instead. Again, submission *asfffas4* (in yellow) is late and has no previous attempts, so it will be treated as late submission.

Finally, to verify that the extracting script works correctly, the testing function checks that the corresponding extracted directory exists in the location specified. A few misnaming defects, due to logic problems in the code that parses the zip file name and builds the target directory, were detected like this.

The required format for the submission archives is `.zip`, so a few test cases have other types of archives as target, including an example of “corrupted” `zip` file (it was obtained by compressing using the `rar` format and then changing it to `zip` – a situation that happened over the years).

Grading scripts

The various functions used in the grading scripts were largely verified through unit testing (see 5.4.1). Instead, the tasks performed by the four `test_and_grade` scripts have the aim to put together all the small functions in a logical order, so to verify whether the scripts produce the desired output, at this stage they were tested as a whole. A set of sample submissions are necessary for this type of testing, however, as explained in Chapter 3.9, real student submissions from past years could not be used for privacy reasons.

The solution that we adopted is a collection of sample submissions derived from the author’s own, which were developed as she attended COMP61511 2017/18. This gave a chance to create submissions with specific problems or errors and test particular paths of the grading scripts. For example, among the features that the derived submissions include, we can have:

- a WC script that does not compile;
- various kinds of misnamed or missing files;

- scripts that do not follow the required format when printing (e.g. they print on multiple lines);
- scripts with a directory structure that does not follow the one suggested (e.g. `wc.py` is in the main folder, `doctest_wc.py` and the test input files are in one folder called `tests/`;
- scripts with prohibited libraries, or with modules imported in various formats (see examples in Listing 5.2);
- scripts passing and failing different categories of tests (e.g. not accepting binary files, counting actual lines instead of `\n characters` leading to off-by-one errors, with wrong counts in general);
- scripts not following some given requirements (e.g. incorrect error messages, flags not implemented when they should be, ...).

```
import sys
import sys, argparse
import sys.argv
from sys import argv
from os.path import join
import sys as system
```

Listing 5.2: Different ways to import a module in Python that the grading tools recognise

A note about sample submissions

To overcome the privacy issues explained in Chapter 3.9, the author's own WC assignments were taken as base models, from which a set of sample submissions to be used for tests was derived.

Of course, this process could not guarantee much variety in some of the resulting marking metrics. For example, since the programme structure and the test cases were the ones originally built by the author, the results captured display similar rates in the amount of successful tests or in the coverage rates, or again in the number of LOC. For many submissions, she tried and diversify the results, for instance removing a number of tests from a sample submission's suite to reduce the coverage. However, the differentiation cannot be expected, and is not intended, to approximate the variety of results that is normally observable in a class context.

On the other hand, in order to test all aspects and logic paths of our tools, creating sample submissions with different types of errors and defects was necessary, and having a model was an acceptable starting point to derive submissions with specific features, like the ones explained in the previous section.

In conclusion, testing the tools on submissions that all look similar, and yet have one or two characteristics that are used to test a particular condition of the graders, was an effective method to verify the assessment programmes in a systematic way, without getting "lost" in a large group of submissions that have presumably very different features.

The real submissions, however, were used in a second phase of testing, when the main structure of the graders had been defined, and the results were used to improve the tools, so that they could cover a few particular error-prone cases that the original tools did not consider, too.

5.4.3 Regression testing

Whenever some change was introduced in the code, the existing test suites were run again. This is known as regression testing and its objective is to ensure that no new changes introduced errors in the code or somehow caused problems in any existing functions [57].

This was often the case when modifying the directory structure, for example by moving the submission archives one level down so that they were grouped by coursework number: in this and other cases, any test that contained the old paths failed and had to be updated.

5.5 Release and System testing

Release testing, that is verifying and validating a version of the software product outside the normal development process, normally together with the system's final users or customers [57], was performed whenever the author met the instructor to update the project progress.

Initially, that is before the author could present a working set of tools, she only explained any new features she had been implementing since the previous meeting. Once the tools were completed with at least their most basic functionality, the release testing consisted of an execution of the tools using real assignments on the instructor's machine. This is why release testing also involved an overall system testing, since of course it was the only time to test the whole system of scripts with real assignments.

Typically, a release testing session consisted of an overall run of the tools, for example through the command `doit` or by executing the corresponding script. At the beginning, when the automated functionality had not been set yet, the tool stopped running whenever some exception that the programme did not catch was triggered, so the only option to continue the execution was to fix the problem and retry. However, thanks to PyDoit's `--continue` functionality, the tools could eventually run up to the end, and any error caused when assessing some submissions was saved to a log file so that the programme could continue.

The author would then work on a solution to fix the bug and then include any newly-discovered error in a new sample submission, which would represent a test case for that patched error. This activity represented several opportunities to discover bugs and other code defects that the author alone had not been able to catch and process properly.

For example, initially the author had named the derived sample submissions following the model `username_cwX_01-01-18.zip` (as the submissions look like from the student's perspective, plus the date). However, BlackBoard automatically attaches other data to the file names, specifically the coursework name, the username and the attempt date, then the name the student gave to the submission. Running the tools on the instructor's machine caused errors when parsing the username and attempt date from the file names, as the names were formatted differently. This gave the author the chance to fix that bug in the code.

5.5.1 Requirements-based testing

The requirements of a software product should be designed so that they can be tested, and requirements-based testing aims at providing evidence that the requirements have been satisfied – so it falls under validation testing [57].

A set of requirements was defined from the beginning, to guide the author to develop a suitable set of tools; however, she was given some levels of freedom so that she could focus on the basic aspects of the tools and possibly add further functionalities if the time would allow. Also, new minor requirements were added or, more frequently, refined from time to time during the development process. Examples include the use of a local database to store marks and other data about the submissions, adding meta-data and student errors, introducing ideas for new test cases, the use of PyDoit itself to automate the execution of the scripts.

Table 5.1 lists a few macro- and micro-requirements for the COMP61511 project (including some that were not implemented) and explains how these were tested.

Requirement description	**Status**	**Details about implementation and testing**
Filtering out late subs	Implemented	A function was built to recognise late submissions or submissions with more recent punctual attempts.
Unzipping submissions	Implemented	The unzip function extracts the submission archives to a target directory and provides error handling for errors related to the extracting activity (corrupted archives, wrong file formats...)
Support for customising test cases, evaluation metrics and grading scales	Implemented	The scripts come with four JSON file containing max marks and grading parameters that the instructor can change as he likes.
Saving computed marks to log files and database.	Implemented	All marks and other data are saved to an ongoing log file and, once computed, to a local database.
Submissions that require manual checks should be moved to a separate dir	Implemented	Bad submissions either do not compile or have some important file missing that does not allow many tests to be run. The instructor can then check the submissions to try and fix the problem, and add penalties if necessary.
Checking that only permitted Python modules were used in the script	Implemented	A function checks that no prohibited modules were used, with support for the various formats for module import.
Recursive search for required submission files	Implemented	Initially the tools assumed a rigid structure for the submissions and searched for the files only and the top level. With a recursive search, instead, it does not matter at what level the file is located.
Plagiarism and collusion detection	Partially implemented	The grading tools highlight any similar errors made by the students or common sets of test cases failed, which may or may not be indicators of collusion — it is up to the instructor to judge. The MOSS tool [59] for identifying similar code chunks in student submissions was installed and tests were performed, but the implementation of a working collusion detector is at an early stage only. There is currently no support for checking similarity with online sources (e.g. Stackoverflow, or WC clones built by other developers).
Support for CI (Continuous Integration)	Not impl.	The goal would be setting up a grading system that makes used of Continuous Integration [60]. With such support, students could have tests performed on their code whenever they commit, which would represent an even faster feedback model (see Chapter 6.4.1).

Table 5.1: Requirements-based testing

5.6 User testing

In the last phase of testing, the objective is to verify the product from the end user's perspective in a real, or as close as possible to real, working environment [57]. In case of the automated grading tools developed for COMP61511, the course instructor represents the main "customer" the programme will have. Also, in this case the software product is relatively small, so the user testing stage was partially covered by the release testing.

Since we had the students' submissions and actual grades of previous years available, the tools were tested on the past coding exercises and the marks attributed were compared against the actual ones. The goal was to verify that the new tools' grading metrics would not be significantly distant from the previous ones, in order not to give unbalanced grades. Of course, the resulting grades varied due to the different weights assigned to various aspects of a submission, however we would not expect important differences between the metrics.

Also, the grades computed need to be reasonable above all, and the grade distribution in the class should not be excessively skewed, meaning that the grading criteria were too hard, or too tolerant. Of course a student who followed the structural requirements and passed all or most tests will get a high mark, and similarly a very low mark will be given if a submission contains important structural defects or failed most tests.

In case the instructor believed that the marks that some students received from the tool were not adequate, for example if it becomes evident from the class performance that many people faced similar difficulties, the actions he could do to mitigate this circumstance would involve adjusting the weight of the marking parameter or the grade scale used. This can be easily done by modifying the JSON configuration files.

5.7 Benchmark results

The final version of the tools was eventually timed, in order to have an idea of how long the grading process takes when using our tools. Again, the Unix TIME utility [33] was used when calling the grading scripts from the command line, and the Python module TIME [61] to measure how single chunks of the code performed. The results are provided in this section.

The execution time was monitored for all the main functions, considering both the total running time and the time each sub-task required. Table 5.2 shows how much time the unzip function takes on average, run by using `doit` or by calling the script `unzip_submissions.py`. We separated the performances for each coursework, although the unzip tool does not change according to the coursework (like `test_and_grade` does). What does change is the number of sample archive submissions that were used, and the more submissions are processed, the larger is the total amount of time required.

The difference between `doit`'s and the script's performances is also worth noticing: the reason why this happens is that `doit` works in a way that requires running a new instance of the programme for every submission archive found in the directory (see for example Listing 4.6), whereas calling the script processes all archives in a single run. This requires a significant amount of time. However, the archive extraction is a process that is completed much more quickly than `test_and_grade`, so in this case a worse performance is worth the other advantages of `doit`.

Command	**Avg time required**	**No. of Submissions**	**Avg time per submission**
doit unzip_cw1	9.4s	17	0.55s
python unzip cw1	0.67s		0.04s
doit unzip_cw2	7.7s	10	0.77s
python unzip cw2	0.75s		0.07s
doit unzip_cw3	6.1s	9	0.67s
python unzip cw3	0.77s		0.08s
doit unzip_cw4	8.9s	11	0.81s
python unzip cw4	1.23s		0.11s

Table 5.2: Execution time of the unzip function as run by *doit* or by calling the scripts.

Concerning the performance of the `test_and_grade` tools, Table 5.3 shows how well `doit` performed versus the standard script call. Note that the number of sample submissions decreased for cw1 and cw2, because the samples included some archives that could not be extracted (e.g. damaged zip files).

As anticipated, `test_and_grade` scripts required a considerably greater amount of time, due to the complexity of the sub-tasks executed. Additionally, as we go forward with the coursework, the parameters to evaluate increase in number and complexity, so there is a relevant difference between the average time needed for cw1 (which only includes tests for `miniwc`) and the others.

Again, `doit` takes slightly more, that is between 2 and 4 seconds per single submission on average, and again the reason behind this is how `doit` creates and handles the processes.

Finally, the individual average time refers to a set of "complete" submissions, that is submissions with all needed files that compiled and whose assessment caused no error that would lead the process to stop before completion: if this happened, of course the process would take less time.

We also report the execution performance of `test_and_grade_cw4` grading a sample submission due for CW4, which is the coursework with most sub-tasks because of the many metrics adopted.

Command	**Avg time required**	**No. of Submissions**	**Avg time per submission**
doit grade_cw1	1m02s	13	4.8s
python grade cw1	40s		3.1s
doit grade_cw2	4m15s	9	28.3s
python grade cw2	3m50s		25.6s
doit grade_cw3	4m24s	9	29.22s
python grade cw3	4m18s		28.7s
doit grade_cw4	4m51s	11	26.4s
python grade cw4	4m33s		24.8s

Table 5.3: Execution time of *test_and_grade* functions as run by *doit* or by calling the scripts.

Table 5.4 summarises the amount of time in seconds that each sub-task of the grading tool required on the author's machine.

Task name	**Time required (s)**	**Test cases present**
Initial tasks and structure tests	0.05	-
Coverage tests	9.48	30
miniwc tests	4.15	20
wc tests	11.53	64
files0-from tests	1.49	6
help-version tests	0.69	4
stdin tests	2.74	15
Printing report and saving to DB	0.38	-
Total	30.72	

Table 5.4: Execution time of *test_and_grade_cw4* sub-tasks for a sample CW4 submission, and number of test cases present.

Knowing how the programme works, it is intuitive that the larger the number of cases that need to be tested the more time the assessment task will take: in fact, for each test case, which consists of an input command, the script will create a subprocess [29] that executes the given command and then processes the output obtained. This is evident in particular for the `wc` test cases, which starting from CW2 occupy the greatest amount of resources.

The most variable parameter, however, is the performance of coverage tests, depending on the number of test cases written by the student. For example, the author's final doctest script contained 216 test cases: computing the code coverage for so many test cases occupied about 50% of the entire time.

Also, the number of students in a class represents the most critical factor in the performance. Assuming that the computation of a grade for a CW4 submission takes 30 seconds on average, the tools would need about 40 minutes to obtain the grades for a class of 80 people. However, considering how much time manual grading such amount of assignments would take, this still represents significant time savings.

Additionally, grading an assignment is an activity that allows concurrency, as it represents an independent process: if two or more processes were to run at the same time, they would not interfere with each other. PyDoit allows this kind of multi-thread execution, which is likely not to halve the processing time, but should guarantee some time saved.

5.8 Threats to validity

As already mentioned, the tools were tested using both the script invocation form (e.g. calling `python test_and_grade_cw1.py ...`) and the automated version (i.e. PyDoit) on a set of past submissions on the instructor's machine. Multiple iterations were made, since any exception caused when running the grader was used to identify bugs and missing features in the code and fix them.

The biggest interrogatives are, of course: will the tools be able to assess the student submissions with the minimum possible intervention by the instructor, or a human in general? Do the grades computed by the tools reflect those that a human would give for the same goals?

5.8.1 Grade distortion and sensibleness

Concerning the changes introduced in the coursework, their real effectiveness will be ultimately tested only when the course is taught in the coming years. In fact, we considered running a simulation of the improved teaching material in a realistic classroom context; for example, by selecting a group of MSc-level students having the necessary requisites to attend the COMP61511 course and willing to follow the lectures and do the lab exercises to verify the effectiveness of the new course structure. However, this would require much time and resources, and yet would maybe not provide useful and definitive information about the success of the changes.

We can, nonetheless, analyse how the tools assign grades, to verify they are reasonable for the objectives achieved. Note that the grades computed by the new tools for the old submissions are in some cases inevitably lower that they would normally be, because of some grading criteria that were introduced. For example, the past requirements of the coursework did not define specific names for files like the `doctest` suite in CW1, or the `stdin` functionality was not tested in CW4.

Consider the following Table 5.5, which lists the marks obtained for each grading parameters by three sample submissions, obtained by a "weak", an "average" and a "skilled" student respectively. The parameters refer to CW4 and the maximum marks obtainable are specified in parentheses.

Parameter	“Weak” student	“Medium” student	“Good” student
script present (max=0.5)	0.5	0.5	0.5
doctest present (0.5)	0	0.5	0.5
unittest present (0.5)	0	0	0.5
required libraries (0.5)	0.5	0.5	0.5
coverage (2)	0	1	2
miniwc tests (2)	1	1	1
wc tests (7)	2	4	6
files0-from tests (2)	0	1	2
help & version tests (2)	0	1	2
stdin tests (3)	0	2	2
Final grade (max=20)	4	11.5	17

Table 5.5: Comparison of marks per parameter obtained by students with different levels of skills.

The first student gained marks for having the `wc.py` script, which allowed Python modules exclusively, but passed only a small percentage of the functionality tests (the basic `miniwc` tests and tests with flags and files). Regardless of what test cases the script passed or failed, such marks indicate that the student most likely still lacked of an adequate understanding of the specifications (e.g. no tests were found, functionalities like the `--files0-from` flag and `stdin` seem to be missing,...) and, in general, had significant problems in building the correct functionality. The grade that the tool gave is therefore reasonable.

The “medium” student, instead, included most required files in his submission, however the rate of tests passed was around 50% in all parameters, indicating that the `wc` functionality was not implemented completely, or contains some faults. There is a wide combination of points that a student can obtain to place around the *pass* range, which is appropriate considering that many students tend to score in the medium range.

To reach higher grades, that is *merit* and above, the students need to pass as many tests as possible. The “skilled” student in the example did so and was given high marks in both the structural (existing files, allowed modules, coverage) and the functional tests.

Note that according to the British university grading system [62], any mark above 70% belongs to the maximum range that a student can obtain, so a mark like 17/20, which is 85%, is considered quite a high grade. So, again, the grade suggested by the automated tools looks reasonable according to the goals obtained. Additionally, in case the instructor realises that the class’ grade distribution is too skewed, meaning that the tool was too strict, or too indulgent, he could easily change the weight assigned to each parameter.

Chapter 6

Conclusion

6.1 Chapter overview

This chapter sums up the activities that were done for the COMP61511 auto-grader project and discusses any functionalities that were not achieved, but that could be implemented as future work.

6.2 Project summary

The initial motivation for this project was to analyse the current state of the COMP61511 coursework and think of ideas to improve the teaching, the evaluation process and hopefully the learning outcomes. Having been herself an attendee of the course, the author decided to focus on one main aspect of it – that is the timing of the feedbacks, which were released quite late and because of that were not as useful as they should be.

For the practical part of the project, a set of tools were designed and built using Python, the programming language that students must use for the assignments: their objective is to perform all the most repetitive tasks of grading a large set of assignments in an automated way. At the end of the coursework, students should have implemented a clone of `wc` using a re-engineering approach, and the said programme is particularly suitable for automated testing: there is a minimal ambiguity for the output correctness (after splitting the output to remove spaces, it is sufficient to compare the counts) and the `wc` original tool can be used as model to verify if the student's programme behaves accurately. Additionally, custom grading functions could be developed, for example to check that only allowed libraries were used, or to compute the test suite's code coverage.

A specific grading script was created for each of the four stages of the coursework, from an initial `miniwc` with basic functionality up to a one-to-one version of `wc`. The four scripts assess the student submissions according to various criteria that can be personalised by the instructor, for example by changing the maximum amount of marks that passing tests can receive. Also, we developed tools that, from a set of submission archives downloaded from BlackBoard, select the ones that were committed on time and, in case multiple submissions are allowed, are the latest attempt. The tools also extract the archives to prepare them for grading and finally print out the result of the tests, as well as create a set of log files containing grades and feedback to be uploaded back to BlackBoard.

The tools were tested using a set of sample submissions derived from the author's one, both to avoid manipulating the real assignments made by past students and to have submissions with different parameters passed or failed, in order to test all possible branches of the code. Real assignments were used at a later time to perform a complete system testing, and to verify that the grading scripts cover as many cases as possible.

6.3 Issues encountered

No problems that could seriously undermine the success of the project were encountered.

Speaking of programming tools and techniques, the author had some previous, yet not deep, experience with Python that allowed her to design and develop the tools in almost complete autonomy. Occasionally, she followed the instructor's suggestions whenever he recommended some Python libraries and frameworks that would provide specific functionalities or ease the developer's job, and of which the author was not aware.

Not being allowed to perform tests directly on real submissions was moderately challenging at the beginning, for the reasons explained in section 5.4.2. However, eventually the creation of sample submissions proved a valuable challenge for the author, as she was stimulated to think of possible submission outcomes, and even errors that students made, that she had not considered in the first place.

In conclusion, the project had a number of complex objectives, some of which could unfortunately not be realised due to the limited time, but it represented a valuable challenge for the author to deal with the difficult task of grading student assignments and implement a set of tools that will hopefully ease the instructor's job in the following years.

6.4 Future work

Building auto-graders for the COMP61511 coursework included several complex objectives, of which not all could be implemented within the time of the project. Below we present what we believe are the most interesting aspects to work on in the future.

6.4.1 Continuous Integration

Continuous Integration (CI) is a coding practice that involves frequent code integration into a repository, up to several times per day [63]. Typically, CI is a technique used when there are many developers taking part in a project and its main objectives are to solve conflicts and quickly and integration problems to automate the software build, which consists of code compilation, packaging and testing [64].

So far, to assess the student assignments, the instructor had to receive all the submissions, run the tests and then upload the results to the university's system. Thanks to the automated tools, this process should now take considerably less time. However, the feedback that students receives still cannot be considered immediate. For the aims of COMP61511, of course immediacy is not essential, but it would be interesting if the assessment could be even faster and require even less manual effort from the teacher.

A CI system, for example set up through GitLab's framework [60], would constitute a valid tool for this purpose: students could have tests performed on their code and receive a feedback every time they commit. Of course, the feedback model should be studied carefully, for example to avoid sharing too much information about the test cases used and prevent students from programming by trial-and-error. In any case, CI represents a promising and innovative idea to use in automated grading and it should definitely be explored as future work.

6.4.2 Better integration with Blackboard

Performing offline assessment and then uploading the results was the only mean to communicate with BlackBoard, whose functionality in terms of customised grading is very limited. We do not know if any future version of the system will allow such option. However, with a working CI environment set up, we should be able to move most of the grading process onto a system like Gitlab, which is conceived for programming tasks and should provide much more freedom and customisation in that sense. For example, the CI environment would work as submission manager too, by simply processing the last version of the programme that the students released. In this way, only the aspect of the grade publication would remain on BlackBoard.

6.4.3 Tool generalisation and Further grading metrics

Knowing that the instructor will probably want to modify some aspects of the coursework in the future, the tools allow some levels of customisation of the metrics, such as which flags and functionalities should each phase of the coursework contain or how many marks should be given for each parameter. We are aware, nonetheless, that there is a limit to the changes the instructor can introduce without the need to change the grading tools too.

In fact, we saw that WC works quite well as the target of an automated tool, because it produces a well-defined set of outputs that can be easily compared to the model's ones. It is obvious, though, that the type of tools we developed cannot be applied to all kinds of programming assignments (even assessing a programme other than WC that accepts files and flags and produces some console output would require some customisation), so building a general grader is definitely not a trivial task.

As future work, we would like to investigate this topic, that is to what extent it is possible to automate the assessment of programming assignments, which is a promising subject in the field of Computer Science Education and, again, represents significant savings in time and effort for software engineering teachers.

Appendix A

Relevant code snippets

```
$ wc --version
Usage: wc [OPTION]... [FILE]...
  or:  wc [OPTION]... --files0-from=F
Print newline, word, and byte counts for each FILE, and a total line if
more than one FILE is specified.  A word is a non-zero-length sequence of
characters delimited by white space.
With no FILE, or when FILE is -, read standard input.
The options below may be used to select which counts are printed, always in
the following order: newline, word, character, byte, maximum line length.
  -c, --bytes            print the byte counts
  -m, --chars            print the character counts
  -l, --lines            print the newline counts
      --files0-from=F    read input from the files specified by
                          NUL-terminated names in file F;
                          If F is - then read names from standard input
  -L, --max-line-length  print the maximum display width
  -w, --words            print the word counts
      --help     display this help and exit
      --version  output version information and exit
GNU coreutils online help: <http://www.gnu.org/software/coreutils/>
Full documentation at: <http://www.gnu.org/software/coreutils/wc>
or available locally via: info '(coreutils) wc invocation'
```

Listing A.1: wc –*help* command.

```
$ wc --version
wc (GNU coreutils) 8.28
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses...
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Written by Paul Rubin and David MacKenzie.
```

Listing A.2: wc –*version* command.

```
$ wc file.txt test.txt
wc: file.txt: No such file or directory
2      6       56      test.txt
2      6       56      total
```

Listing A.3: wc errors: No such file or directory.

```
$ wc Downloads/*
44     642     1256    Downloads/archive.zip
wc: Downloads/app: Is a directory
0      0       0       Downloads/app
235    1256    125512  Downloads/killer-queen.txt
279    1898    126768  total
```

Listing A.4: wc errors: Is a directory.

```
$ wc -k test.txt
wc: invalid option -- 'k'
Try 'wc --help' for more information.
```

Listing A.5: wc errors: invalid option.

```
$ wc --wordz test.txt
wc: unrecognized option '--wordz'
Try 'wc --help' for more information.
```

Listing A.6: wc errors: unrecognized option.

```
$ wc --files0-from=files0 -
wc: extra operand '-'
file operands cannot be combined with --files0-from
Try 'wc --help' for more information.
```

```
$ wc --files0-from=files0 test.txt
wc: extra operand 'test.txt'
file operands cannot be combined with --files0-from
Try 'wc --help' for more information.
```

```
$ wc  test.txt --files0-from=-
wc: extra operand 'test.txt'
file operands cannot be combined with --files0-from
Try 'wc --help' for more information.
```

Listing A.7: wc errors: extra operands.

```
$ wc -kwc --wordz test.txt
wc: invalid option -- 'k'
Try 'wc --help' for more information.
```

```
$ wc --wordz -k test.txt
wc: unrecognized option '--wordz'
Try 'wc --help' for more information.
```

Listing A.8: wc error precedence: invalid and unrecognized option.

```
{
  'required_files': {
    'script'      : "miniwc.py",
    'test_script': "doctest_miniwc.py",
    'tests_dir'   : "test_files"
  },
  'allowed_modules': ["sys"],
  'marks_structure': {
    'script_present': 0.5,
    'tests_present': 0.5,
    'required_libraries': 1
  },
  'marks_correctness': {
    'correct_formatting': 1,
    'text': 5,
    'binary': 1,
    'unicode': 1
  },
  'stats': {
    'correct_line_percent': 0,
    'correct_word_percent': 0,
    'correct_byte_percent': 0,
    'correct_filenames_percent': 0,
    'correct_tuples_percent': 0
  },
  'output': [
    {'lines': "0", 'words': "1", 'bytes': "100", 'file': "CW1/cw1_test_files/100B_file.txt"},
    {'lines': "0", 'words': "1", 'bytes': "10000", 'file': "CW1/cw1_test_files/10KB_file.txt"},
    {'lines': "0", 'words': "1", 'bytes': "1000000", 'file': "CW1/cw1_test_files/1MB_file.txt"},
    {'lines': "0", 'words': "0", 'bytes': "0", 'file': "CW1/cw1_test_files/empty.txt"},
    ...
  ]
}
```

Listing A.9: Configuration JSON file for CW1.

```
{
  'required_files': {
    'script': "wc.py",
    'doctest_tests': "doctest_wc.py",
    'unittest_tests': "unittest_wc.py",
    'tests_dir': "test_files",
    'argparse_script': "wc_argparse.py",
    'unit_script': "wc_unit.py"
  },
  'allowed_modules': ["sys", "doctest", "unittest", "argparse"],
  'flags': ["l", "w", "m", "c", "L"],
  'marks_structure': {
    'script_present': 0.5,
    'doctest_present': 0.5,
    'unittest_present': 0.5,
    'argparse_script_present': 0.5,
    'unit_script_present': 0.5,
    'required_libraries': 0.5,
    'coverage': 2
  },
  'marks_correctness': {
    'miniwc_tests': 2,
    'simple_wc_tests': 5,
    'complex_wc_tests': 4
  },
  'marking_params': {
    'coverage_ranges': [35, 85]
  },
  'output_miniwc_tests': [
    {'lines': "0", 'words': "1", 'bytes': "100", 'file': "CW1/cw1_test_files/100B_file.txt"},
    {'lines': "0", 'words': "1", 'bytes': "10000", 'file': "CW1/cw1_test_files/10KB_file.txt"},
    ...
  ],
  'output_wc_tests_simple': [
    {
      'command': "-lwmcl test_files/*",
      'output': [
        {
          'lines': "0",
          'words': "1",
          'chars': "100",
          'bytes': "100",
          'max-line': "100",
          'file': "test_files/100B_file.txt"
        },
        {
          'lines': "14",
          'words': "44",
          'char': "418",
          'byte': "436",
          'max-line': "222",
          'file': "test_files/pdf-sample4.pdf"
        },
        ...
        {
          'lines': "892",
          'words': "6980",
          'chars': "1065557",
          'bytes': "1071442",
          'max-line': "1000000",
          'file': "total"
        }
      ]
    },
  ...
}
```

Listing A.10: Configuration JSON file for CW3.

Appendix B

Project metadata

B.1 Package structure

```
/
├── CW1/
│   ├── cw1_test_files/
│   ├── test_cw1_correctness.py
│   ├── test_cw1_structure.py
│   └── wc_output_and_marks_cw1.json
├── CW2/
│   ├── test_cw2_correctness.py
│   ├── test_cw2_structure.py
│   └── wc_output_and_marks_cw2.json
├── CW3/
│   ├── test_cw3_correctness.py
│   ├── test_cw3_structure.py
│   └── wc_output_and_marks_cw3.json
├── CW4/
│   ├── test_cw4_correctness.py
│   ├── test_cw4_structure.py
│   ├── wc_output_and_marks_cw4.json
│   └── stdin_unittest.py
├── Downloads/
│   ├── cw1/
│   │   └── cw1_sample_submission_1.zip
│   ├── cw2/
│   ├── cw3/
│   └── cw4/
└── sql/
    ├── create_table.py
    ├── db_comp61511
    ├── insert.py
    └── select.py
```

```
 ├── lib/
 │   └── utilities.py
 ├── unittests/
 │   ├── test_files/
 │   └── unittest_utilities.py
 ├── dodo.py
 ├── select_last_submissions_before_deadline.py
 ├── extract_results.py
 ├── test_and_grade_cw1.py
 ├── test_and_grade_cw2.py
 ├── test_and_grade_cw3.py
 ├── test_and_grade_cw4.py
 └── unzip_submissions.py
```

B.2 System information

OSs

- MacOS High Sierra Version 10.13.6
- Ubuntu 18.04.1 LTS

IDE and writing tools

- PyCharm 2018.1 Professional Edition
- Atom 1.28.2

Software and Frameworks

- GitLab 11.1
- PyDoit 0.31
- Python 3.6.5
- SQLite 3.24.0
- WC 8.28

B.3 Lines of Code

Script name	Lines of Code	No. of sub-functions
test_and_grade_cw1.py	99	2
test_cw1_correctness.py	146	4
test_cw1_structure.py	44	1
test_and_grade_cw2.py	99	2
test_cw2_correctness.py	195	7
test_cw2_structure.py	80	1
test_and_grade_cw3.py	99	2
test_cw3_correctness.py	211	7
test_cw3_structure.py	67	1
test_and_grade_cw4.py	97	2
test_cw4_correctness.py	253	8
test_cw4_structure.py	62	1
stdin_unittest.py	95	15
lib/utilities.py	713	55
unittest/unittest_utilities.py	379	45
sql/create_table.py	138	5
sql/insert.py	138	7
sql/select.py	310	15
dodo.py	263	24
extract_results.py	42	1
select_last_submissions_before_deadline.py	127	5
unzip_submissions.py	93	3
Total	3750	213

Table B.1: Number of Lines of Code and sub-functions per script file.

Bibliography

[1] University of Manchester: School of Computer Science, *COMP61511: Software Engineering Concepts in Practice course syllabus*, `http://www.cs.manchester.ac.uk/study/taught-masters/courses/courseunitsyllabus/?courseunitcode=COMP61511`, [accessed 03-09-2018].

[2] The Linux Information Project, *The wc Command*, `http://www.linfo.org/wc.html`, [accessed 03-09-2018].

[3] E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, 1990.

[4] B. Beizer, *Software testing techniques*. Van Nostrand Reinhold, 1990, p. 550.

[5] S. Benford *et al.*, "Courseware to support the teaching of programming," in *Developments in the teaching of computer science*, University of Kent at Canterbury, 1992, pp. 158–186.

[6] A. Zeller, "Making students read and review code," in *Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in CS education - ITiCSE '00*, vol. 32, New York, USA: ACM Press, 2000, pp. 89–92.

[7] B. Cheang *et al.*, "On automated grading of programming assignments in an academic institution," *Computers & Education*, vol. 41, no. 2, pp. 121–131, Sep. 2003.

[8] D. Jackson, "Using software tools to automate the assessment of student programs," *Computers & Education*, vol. 17, no. 2, pp. 133–143, Jan. 1991.

[9] ——, "Computer-Based Evaluation of Student Software Quality," in *2nd Conf. Software Engineering in Higher Education (SEHE92)*, Southampton, UK, 1992, pp. 93–104.

[10] ——, "A software system for grading student computer programs," *Computers & Education*, vol. 27, no. 3-4, pp. 171–180, Dec. 1996.

[11] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[12] K. Blaha *et al.*, "Do students recognize ambiguity in software design? a multi-national, multi-institutional report," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, IEEE, pp. 615–616.

[13] D. Jackson and M. Usher, “Grading student programs using ASSYST,” in *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education - SIGCSE ’97*, vol. 29, New York, USA: ACM Press, 1997, pp. 335–339.

[14] M. H. Goldwasser, “A gimmick to integrate software testing throughout the curriculum,” in *Proceedings of the 33rd SIGCSE technical symposium on Computer science education - SIGCSE ’02*, vol. 34, New York, USA: ACM Press, 2002, p. 271.

[15] J. Spacco *et al.*, “The Marmoset project,” in *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA ’06*, New York, USA: ACM Press, 2006, p. 669.

[16] G. Tremblay *et al.*, “Oto, a generic and extensible tool for marking programming assignments,” *Software: Practice and Experience*, vol. 38, no. 3, pp. 307–333, Mar. 2008.

[17] G. Tremblay and P. Lessard, “A marking language for the oto assignment marking tool,” in *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education - ITiCSE ’11*, New York, USA: ACM Press, 2011, p. 148.

[18] D. M. de Souza *et al.*, “ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities,” in *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, IEEE, May 2011, pp. 1–10.

[19] D. M. de Souza *et al.*, “Towards the use of an automatic assessment system in the teaching of software testing,” in *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, IEEE, Oct. 2014, pp. 1–8.

[20] S. H. Edwards and M. A. Pérez-Quinones, “Web-CAT: Automatically grading programming assignments,” in *Proceedings of the 13th annual conference on Innovation and technology in computer science education - ITiCSE ’08*, vol. 40, New York, USA: ACM Press, 2008, p. 328.

[21] ——, “Experiences using test-driven development with an automated grader,” *Journal of Computing Sciences in Colleges*, vol. 22, no. 3, pp. 44–50, 2002.

[22] S. H. Edwards, “Rethinking computer science education from a test-first perspective,” in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA ’03*, New York, USA: ACM Press, 2003, p. 148.

[23] ——, “Improving student performance by evaluating how well students test their own programs,” *Journal on Educational Resources in Computing*, vol. 3, no. 3, pp. 1–24, Sep. 2003.

[24] GNU Free Software Foundation, *GNU Coreutils*, `www.gnu.org/software/coreutils/manual/html_node/wc-invocation.html`, [accessed 03-09-2018].

[25] The Linux Information Project, *Standard Input definition*, `http://www.linfo.org/standard_input.html`, [accessed 03-09-2018].

[26] Jukka "Yukka" Korpela, *Unicode Spaces*, `http://jkorpela.fi/chars/spaces.html`, [accessed 03-09-2018].

[27] Python Software Foundation, *The Python Standard Library*, `https://docs.python.org/3/library/index.html`, [accessed 03-09-2018].

[28] ——, *doctest – Test interactive Python examples*, `https://docs.python.org/3.6/library/doctest.html`, [accessed 03-09-2018].

[29] ——, *Python – Subprocess management*, `https://docs.python.org/3.6/library/subprocess.html`, [accessed 03-09-2018].

[30] ——, *argparse – Parser for command-line options, arguments and sub-commands*, `https://docs.python.org/3/library/argparse.html`, [accessed 03-09-2018].

[31] ——, *unittest – Unit testing framework*, `https://docs.python.org/3/library/unittest.html`, [accessed 03-09-2018].

[32] The Linux Information Project, *Standard Input, Standard Output, Standard Error*, `http://www.linfo.org/stdio.html`, [accessed 03-09-2018].

[33] The Linux Programming Interface, *Time – Linux User's manual*, `http://man7.org/linux/man-pages/man1/time.1.html`, [accessed 03-09-2018].

[34] E. Soloway *et al.*, "Cognitive strategies and looping constructs: an empirical study," *Communications of the ACM*, vol. 26, no. 11, pp. 853–860, Nov. 1983.

[35] K. Fisler, "The recurring rainfall problem," in *Proceedings of the tenth annual conference on International computing education research - ICER '14*, New York, USA: ACM Press, 2014, pp. 35–42.

[36] M. Guzdial, "Exploring the dual nature of computing education research," *Communications of the ACM*, vol. 54, no. 2, p. 37, Feb. 2011.

[37] Simon, "Soloway's Rainfall Problem Has Become Harder," in *2013 Learning and Teaching in Computing and Engineering*, IEEE, Mar. 2013, pp. 130–135.

[38] O. Seppälä *et al.*, "Do we know how difficult the rainfall problem is?" In *Proceedings of the 15th Koli Calling Conference on Computing Education Research - Koli Calling '15*, New York, USA: ACM Press, 2015, pp. 87–96.

[39] M. De Raadt, *Teaching Programming Strategies Explicitly to Novice Programmers*. University of Southern Queensland, 2008.

[40] D. McCandless *et al.*, *Codebases - Millions of lines of code*, `https://informationisbeautiful.net/visualizations/million-lines-of-code/`, [accessed 03-09-2018].

[41] M. G. Limaye, *Software testing: principles, techniques and tools*. Tata McGraw-Hill Education Private Ltd, 2009, p. 216.

[42] R. Patton, *Software Testing*. Indianapolis: Sams Publishing, 2001, p. 57.

[43] M. Shahbaz, "Reverse Engineering and Testing of Black-Box Software Components / 978-3-659-14073-0 / 9783659140730 / 3659140732," PhD thesis, 2012, pp. 6–8.

[44] C. Costa-Soria *et al.*, "An approach for teaching software engineering through reverse engineering," in *2009 EAEEIE Annual Conference*, IEEE, Jun. 2009, pp. 1–6.

[45] M. Weiser, *Program Slicing*. Association for Computing Machinery, 1984, p. 232.

[46] S. H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," *ACM SIGCSE Bulletin*, vol. 36, no. 1, p. 26, Mar. 2004.

[47] R. Adcock *et al.*, "Curriculum Guidelines for Graduate Degree Programs in Software Engineering," Stevens Institute of Technology, Tech. Rep., 2009.

[48] J. Kelleher, "Employing git in the classroom," in *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*, IEEE, Jan. 2014, pp. 1–4.

[49] E. Bonakdarian, "Pushing Git and GitHub in undergraduate computer science classes," *Journal of Computing Sciences in Colleges*, vol. 32, no. 3, pp. 119–125, 2002.

[50] A. Rosenbloom *et al.*, "GIT: Pedagogy, Use and Administration in Undergraduate CS," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '17*, New York, USA: ACM Press, 2017, pp. 82–83.

[51] S. Mohan and S. Chenoweth, "Teaching Requirements Engineering to Undergraduate Students," Tech. Rep., 2011.

[52] W. J. Wolfe, "Online student peer reviews," in *Proceedings of the 5th conference on Information technology education - CITC5 '04*, New York, USA: ACM Press, 2004, p. 33.

[53] K. Reily *et al.*, “Two Peers are Better Than One: Aggregating Peer Reviews for Computing Assignments is Surprisingly Accurate,” Tech. Rep., 2009.

[54] E. Schettino, *Doit Automation Tool*, `http://pydoit.org/index.html`, [accessed 03-09-2018].

[55] S. H. Edwards *et al.*, “Comparing Effective and Ineffective Behaviors of Student Programmers,” Tech. Rep., 2009.

[56] Project Management Institute, *A guide to the project management body of knowledge*, 6th edition. 2017.

[57] I. Sommerville, *Software engineering*, 10th edition. Pearson, 2016, pp. 227–252.

[58] *IEEE Standard Glossary of Software Engineering Terminology.* IEEE, 1990.

[59] A. Aiken, *MOSS – A System for Detecting Software Similarity*, `http://theory.stanford.edu/~aiken/index.html`, [accessed 03-09-2018].

[60] GitLab, *GitLab Continuous Integration and Deployment*, `https://about.gitlab.com/features/gitlab-ci-cd/`, [accessed 03-09-2018].

[61] Python Software Foundation, *time – Time access and conversions*, `https://docs.python.org/3/library/time.html`, [accessed 03-09-2018].

[62] University of Manchester: School of Computer Science, *Handbook for Masters Programmes*, `https://studentnet.cs.manchester.ac.uk/pgt/2017/handbook/MScHB.pdf`, [page 25, accessed 03-09-2018].

[63] P. M. Duvall *et al.*, *Continuous integration: improving software quality and reducing risk.* Addison-Wesley, 2007, p. 283.

[64] M. Fowler, “Continuous Integration,” Tech. Rep., 2006.