IJCSI

International Journal of Computer Science Issues

Automated style feedback for student programmers

By Abdulaziz Alsulami

Volume 4, 2019
ISSN (Online): 1694-0814

IJCSI PUBLICATION
www.IJCSI.org

AUTOMATED STYLE FEEDBACK FOR STUDENT PROGRAMMERS

Copyright © 2019 by Abdulaziz Alsulami

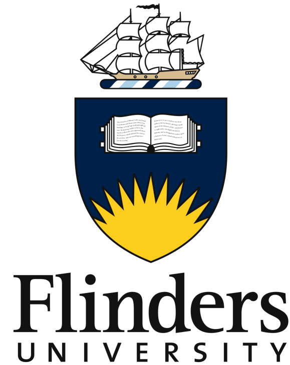
All rights reserved. No part of this thesis may be produced or transmitted in any form or by any means without written permission of the author.

ISSN(online) 1694-0814

IJCSI PUBLICATION 2019

www.IJCSI.org

Automated style feedback for student programmers



Supervisor Dr. Paul Calder

Submitted to the School of Computer Science, Engineering, and Mathematics in the Faculty of Science and Engineering in partial fulfillment of the requirements for the Master's degree program of Computer Science at Flinders University South Australia, Australia

By

Abdulaziz Alsulami - 2125070

Academic Integrity Declaration

I certify that this work does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university and that, to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where due reference is made in the text.

Signature .... Date.......................................

ACKNOWLEDGMENT

I would like to express my gratitude to my supervisor Paul Calder for the useful comments, remarks and engagement through the learning process of this master thesis. Furthermore, I would like to thank my parents and my wife who support my and motivate me through my study. Also, I like to thank the participants in my survey, who have willingly shared their precious time during the process of interviewing. Also, I like to thank my friends for sharing knowledge and information with me.

ABSTRACT

Computer-based tools are used to provide automated feedback to student programmers. Students appreciate being able to receive immediate feedback on their code, and teaching staffs often use the tools to ensure that submitted work fully satisfies program specifications. Many tools exist to check the functional aspects of code; however, few tools aim to assess programming style. This thesis investigates techniques that would allow for automatic assessment of novice students' programming style capabilities. The thesis describes a prototype automatic assessment tool which can provide programming style feedback on several widely used programming languages. The tool has been designed to check and provide feedback on a range of aspects of accepted "good style", including indentation, choice of names, efficiency, documentation, and complexity. The tool feedback has been evaluated by conducting an experiment and survey. The targeted participants were academic staff who have experience in teaching programming, so they are able to provide feedback about the techniques used by the prototype tool and identify additional techniques that have not been covered. Collecting feedback from teachers through the questionnaire helped to reveal disadvantages of the tool feedback and suggest missing assessment factors that need to be included.

TABLE OF CONTENTS

Academic Integrity Declaration ii
ACKNOWLEDGMENT iii
ABSTRACT iv
TABLE OF CONTENTS v
LIST OF TABLES vii
LIST OF FIGURES viii
CHAPTER 1 1
1 Introduction 1
1.1 Background 1
1.2 Static and dynamic analysis 2
1.3 Purpose of Study 4
1.4 Research Objectives 4
1.5 Problems 4
1.6 Contributions 5
2 Literature Review 6
2.1 Program Education Tools 6
2.2 Feedback 8
2.3 Automated assessment tools 10
2.3.1 Benefits of automated assessment tools 14
2.4 Automated-style assessment tools 17
2.4.1 Measurements of automated style assessment 19
CHAPTER 3 23
3 Design 23
3.1 Overview 23
3.2 Design requirements 23
3.3 Different languages 24
3.4 Code Structure 25
CHAPTER 4 28
4 Analysis 28
4.1 Assessment Factors 28
4.1.1 Code Layout 28
4.1.2 Variables and Operators Spacing 28
4.1.3 Blank line 31
4.1.4 Line length 32
4.1.5 Indentation 34
4.2 Name Choice 38
4.3 Method Extraction 42
4.4 Complexity 44
4.5 Documentation 46
4.6 Efficiency 47

4.7 Feedback 49
CHAPTER 5 52
5 Validation 52
5.1 Study description 52
5.2 Significance of the study 52
5.3 Experiment 52
5.3.1 Experimental procedure 52
5.4 Survey 56
5.5 Study results 56
CHAPTER 6 65
6 Conclusion and Future work 65
Appendix 67
References 70

LIST OF TABLES

Table 1: Participants recent experience (last 2 years) in teaching programming 56
Table 2: Participants medium-term experience (last 5 years) in teaching programming 57
Table 3: Participants long-term experience (last 10 years) in teaching programming 57
Table 4: Participant’s opinion about Indentation 60
Table 5: Participant’s opinion about code complexity 60
Table 6: Participant’s opinion about choice of names 61
Table 7: Participant’s opinion about Static efficiency 61
Table 8: Participant’s opinion about documentation 62

LIST OF FIGURES

Figure 1 ASSYST assessment process 14
Figure 2 Rees marking system (Rees 1982) 18
Figure 3 Sample of student programming exercise 30
Figure 4 The AFSA feedback regarding space between words and operators errors....30
Figure 5 AFSA feedback for blank-line errors 32
Figure 6 AFSA feedback for blank-line errors 32
Figure 7 Sample of student-programming exercise 33
Figure 8 The AFSA feedback for line-length errors 33
Figure 9 Sample of a student-programming exercise..36
Figure 10 Indentation of 2 spaces 37
Figure 11 AFSA feedback for indentation...38
Figure 12 sample of student programming exercise 41
Figure 13 The AFSA's feedback for name-choice errors 41
Figure 14 The AFSA's feedback on complexity code 46
Figure 15 AFSA feedbak based on category...50
Figure 16 grade scale...51
Figure 17 AFSA overall feedback 51
Figure 18 unannotated code sample 54
Figure 19 Annotated code sample...55

CHAPTER 1

1 Introduction

When it comes to learning programming languages, one of the most effective ways is for students to work through a series of increasingly difficult exercises. This method of learning would be incomplete unless students receive quick and accurate feedback regarding the value of their solutions to these exercises. Since leaving tedious and sometimes redundant assessment processes to instructors is not very efficient, it is proposed in this thesis that implementation of automated-style feedback through an autonomous system would eliminate the potential of error related to human factors and also enhance the assessment experience for students and instructors alike.

1.1 Background

As has been noted by (Vihavainen et al. 2013) students require a regular amount of practice (exercise in the discipline) in order to master the subject they are studying. Additionally, instructors should be aware of students' level of progress. However, instructors are human and considering the large number of computing students, the burden on a single instructor may become heavy if adhering strictly to manual-assessment methods. Assessment by humans also can lead to subjectivity and inconsistencies (Auffarth et al. 2008).

The use of automated-assessment systems can be traced back to the 1960's. Such systems have proven helpful in terms of reducing instructors' workload and improving the quality of assessments being made. Automated-assessment systems

operate through a set of predetermined measurement values, which enable them to perform a value-by-value comparison between the model (optimal) solution and an individual student's solution (Gupta and Dubey 2012). It also is important to note that automated systems are non-biased and fatigue-free compared to manual assessments undertaken by humans, allowing for accurate and reliable results. However, for such systems to work, the tasks received by students should be clearly defined, and codes should be error-free for the system to be able to determine that a program is working correctly.

1.2 Static and dynamic analysis

Automated assessments can be undertaken through either a static or dynamic program-analysis approach. The difference lies in the fact that static analysis does not rely on the compilation or execution of a program code (making it especially viable for programs that are unable to be compiled due to inherent errors), while dynamic analysis is performed at the same time as executing a program based on certain test data (Gupta and Dubey 2012).

Additional difficulties lie in differences between the various programming languages (and their respective syntaxes) as used by the students (Auffarth, López-Sánchez et al. 2008). The understanding of basic programming syntax is crucial. A typical complication faced by students is that syntax-based glitches and program-code mistakes may be hard to detect (especially for the beginner). This results in a frustrating **‘rinse-and-repeat’** process of recompiling a program only to repeatedly face the same error. From a learning-process perspective, this process may prove discouraging to

students, thus the total number of attempts needed to recompile a potentially faulty code should be reduced. Even if only a single aspect of programming eludes a student, the entire process becomes hard to understand. Since the volume of knowledge needed is comparatively large, there's no guarantee that instructors will be able to impart a programming subject in its entirety. Consequently, reliance on automatic-assessment systems is especially recommended when it comes to syntax-based errors and analysis.

An automated-assessment system should be able to understand the specifics of the programming style and language used in order to analyze written code. Measurements of code style can be divided into the following categories: Modularity, typography, clarity, independence, effectiveness, and reliability. Additionally, a code-style assessment also makes use of code metrics, such as maintainability index, cyclamate complexity, structural complexity, depth of inheritance, the amount of code lines, and class coupling.

According to (Chen et al. 2011), a combination of the learning approaches of exercise-based programming languages coupled with an independent automated system capable of assessing and appropriately scoring the performance of students is considered as one of the more effective approaches, especially for non-computer science students. According to the research by (Pettit et al. 2015), automated-assessment tools have proven very helpful in improving the performance of both students and the teaching experience of instructors; while automatic assessment generally is considered as reliably accurate.

1.3 Purpose of Study

The value of automated-assessment tools as means of providing feedback to students is a considerable area of research. Many tools exist to check the functional aspects of code. However, this thesis aims to use static analysis to promote good programming-style habits by novice programmers. Additionally, it is aimed at helping students and instructors by providing them with a style-based automatic assessment tool, which would make use of the most efficient automatic-assessment techniques. To that end, the scope of this study will cover only a selected range of programming languages.

1.4 Research Objectives

Research objectives consist of investigating techniques that would allow for automatic assessments of students' current programming style capabilities; thus developing a prototype automating-assessment tool, which would be able to provide feedback on several widely-used programming languages; and validating such feedback through a selected, sample range of student programming exercises is crucial.

1.5 Problems

There are some problems that relate to the scope of this thesis. The first is defining the assessment factors that cover the many aspects of code style. The second is the analysis and design tools that assess code style and generate useful feedback. The third is recognizing the structures of the various programming languages that have been

chosen to generate useful feedback. In the following section, methods regarding how these problems can be solved will be discussed.

1.6 Contributions

In order to solve the first problem, the selected design tool defines the assessment factors that assess various aspects of code style. These factors are indentation, code complexity, choice of names, static efficiency and documentation. Regarding the second problem, which is building a tool that assesses and generates feedback about code style, (the tool being built using Java programming language). Regarding the third issue, which recognizes how certain languages differs from others in order to analysis them, the tool uses a different algorithm for each language.

The structure of the thesis as the following chapter 2 presents literature review of automated assessment tools. Chapter 3 introduces the design of the prototype that use to assess the code style. Chapter 4 describes the implementation of the prototype. Chapter 5 evaluates the tool feedback that presented in chapter 4. Chapter 6 highlights the summary of the thesis and the future work.

CHAPTER 2

2 Literature Review

2.1 Program Education Tools

The introduction of computers and the continual release of the different software used in computing have created ongoing changes in the field. New tools have enhanced teaching and learning in the field. (Deek and McHugh.1998) Programming learning tools have been classified into four groups: Programming environment, debugging aids, intelligent-tutoring systems and intelligent-programming environment, with each group including many types of tools. Novices and experts alike are increasing their learning skills through the introduction of such tools. When students learn programming, they use tools that have been made available to them by their academic institution. Computers are the essential platforms used when programming subjects are learned. Consequently, improved software technology has led to an increase in the number of tools used to teach programming subjects (Salleh et al. 2013). There are a number of different skills that need be applied when learning programming as an academic subject. Some of these include planning, testing, designing and debugging. The programming syntax is another essential aspect that programming students should understand. This is because students need to know the basics so that they can create programs.

It is devastating to students when creating a program that will eventually fail due to lack of understanding the proper procedures in creating a program. Practical sessions should be undertaken in association with the theoretical. Theory must be well

understood so that it can be put into practice during lessons. Students may find it hard to learn programming since it is considered to be highly complicated. Many researchers have endeavoured to come up with the best and simplest tools that can be used to ease the difficulties of learning programming at an academic institution. There are various learning tools such as those by Alice, BlueJ, Jeliot, Scratch and Greenfoot that have been used to aid in the programming learning process. Meanwhile, instructors also have experienced difficulties teaching classes (Salleh, Shukur et al. 2013). Programming instructors found it important to have a comprehensive understanding of students' attitudes towards programming. Learning strategies, such as telling stories and gaming activities help capture students' attention in class. Consequently, there have been improvements in the field of programming and many students have benefited through various new teaching and learning tools.

Programming comprises three aspects: The program itself, the programming tools, and the programming language. Development and implementation of programming is supported by tools, which are essential. Programs give instructions through programming tools. Programming tools also support the implementation and testing of programs. It is in the tools where aspects such as programming language, syntax, and logic are mentioned, and programmers rely highly on tools. Also, programming skills are developed through programming tools. The increased levels of technology have promoted the creation of different tools used in programming. Through universities and the computer market, many programming tools are available, but only a very few will be adopted for learning and teaching. Even though the programming field

has been increasingly supported by the rise in the level of technology. Consequently, challenges are faced by those employed in the programming field. To overcome these, programmers endeavor to make his or her needs known so that software developers can work towards meeting their specific requirements. At times, it is difficult for programmers to understand the underlying concepts of software programs, such as error messages and the complexities of a software interface. These create challenges that, consequently, have reduced the number of programming tools used in the current teaching environment.

2.2 Feedback

In the past, faculty staff and lecturers provided feedback to student programmers in scheduled laboratory sessions and classes. Further, the amount of individual attention and time a learner received was haphazard. At present, lecturers may need to give inline feedback on the scholar's code to help them improve on programming skills. Analysis of learner's coding tasks is done using criteria like the style, design, and functionality. Pieterse posits that offering exercises as opportunities to practice is essential, and they become more valuable if instructors provide accurate and fast feedback (Pieterse, 2013). According to Koyya et al., a response to student programming assignments on quality is a laborious and tedious task for the tutor. Markedly, it is hard to spot comments added on the hundreds of lines of code. Today, online submission applications have been designed to address the needs of student programmers by offering instant and automatic feedback to their commitments and efforts, and such solutions have reduced the administrative loads for participating

institution employees such as lecturers required to provide results to the learners (Koyya et al. 2013).

Recently, a typical scenario with programming students involves emailing supervisors with copies of their codes. The emailed program is often incomplete and corrupted with errors. In effect, programmers need an annotation mechanism that would allow them to include comments and queries asking for help. In a similar manner, the approach should enable tutors to assess, analyse, and reply to the questions in a structured and straightforward way. Accordingly, students can get timely and frequent responses to the codes. On the other hand, faculty staff can offer their comments effectively, which improves overall learning.

Today, several computer-aided tools have been created for examining coding assignments and providing feedback (Ala-Mutka 2005). In many programming courses and units in colleges and universities, the use of automated assessment and feedback has proved useful through the deployment of these software applications. Moreover, learners and tutors have observed several benefits of this approach. Firstly, Pieterse mentions consistency, speed, availability, and objectivity of the assessment. Computer-aided tools provide immediate evaluation reports for students who can benefit from early disambiguation of errors and misconceptions. Moreover, online feedback tools have the potential to facilitate learning for trainees who get feedback from any location and at any time. Such systems allow student programmers to be held to higher standards and to meet them.

Significantly, some factors contribute to the efficient use of automatic assessment for student developers. Firstly, the quality of the tests is crucial to the success of a learner. A significant part of successful programming courses is correct tasks. Traditional and manual assessment may fail to recognize this factor. In contrast, automatic assessment focuses on the pedagogical design of codes. Secondly, students should clearly formulate tasks for the application of computer-aided tools. In effect, learners take additional care to avoid vagueness. Thirdly, this approach allows the use of test cases. Accordingly, wrong codes identified as correct through manual assessment can be established. Ultimately, computer-aided tools allow students to resubmit corrected and improved software in response to the instructor's feedback. The prompt fulfilment upon such improvements strengthens the overall training intended by the approach where learners submit assignments and tutors provide feedback. Multiple submissions support iterative studying. On the contrary, in manual and traditional assessment, the trend of multiple submissions may cause negative behaviours. Eventually, deploying automatic assessment using computer-aided tools offers an experience where student programmers understand the course in a more professional manner, with more focus on robust and accurate codes over quick and erroneous solutions (Pieterse, 2013).

2.3 Automated assessment tools

With computer-science education and programming-learning systems, it is important to give detailed and comprehensive assignments to students. This way, students will increase their knowledge of programming languages if they are given well-designed assignments. Software development and implementation also pose

challenges to the students, and much has to be done to increase their understanding and upgrade programming skills. Instructors and lecturers, therefore, are required to dedicate their time and resources to ensure that they have completed enough training to be able to devise and assess problems and assignments (Rahman and Nordin 2007). Assignments given to students must satisfy certain requirements, and it is up to the instructor and the lecturer to ensure that assignments meet these needs. This task becomes difficult when most students are physically removed from their educators and sometimes instructors may lack adequate time to assess student work. There has been much academic research undertaken on automated-assessment systems that can evaluate the progress of programming students.

Assessment systems are grouped in generations; where the first generation is the oldest. Hollingsworth undertook the initial first-generational automated assessment in 1960 (Douce et al. 2005). With this system, students were given exercises on programming, which they submitted on punched cards. The students' answers were run on graded programs that provided two outputs; namely wrong answer and program complete. The main advantage of the system was that it was efficient, and it helped many students in understanding and learning programming. The second generation, conversely, tested the working of the program. Isaacson and Scott utilized this system in 1989, and it checked if the program functioned properly and if the programming style used was done sensibly. Other systems in this generational assessment were the TRY system, which is Unix-based system code submission. The TRY allowed students to test the programs they had created with a test program. The tester then provided

students with results from the program, and the attempt was recorded (Ala-Mutka 2005). Finally, there were the third generation systems that used complicated testing techniques and web technology. These systems allowed students, instructors, developers and tutors to become involved, and test the program in different ways. It will test the design format of the program, the complexity, the execution efficiency and finally the operation and function of the program. This system is referred to as the course marker, and a tutor can change the number of users, edit the content and revise the course in the system (Higgins et al. 2005).

. One of the commonly used automated-assessment tools is CourseMarker (Douce, Livingstone et al. 2005). This is an automated-assessment tool that is used to mark programs. This assessment tool was developed at Nottingham University and supports four different users. It can support the student, the tutor, the teacher and the developer. This program has been built on Ceilidh systems (Benford et al. 1995). The original Ceilidh systems was used to give simple and short answers as marks for work that had been submitted. CourseMarker later was been used to give better feedback to students. It also gave the students an alphabetical scale that showed the percentage and the feedback tree. Through this, the student was able to identify errors and see where marks had been lost (Higgins et al. 2005). There is now existed a provision of tools that enabled an administrator to change user profiles. New users could be added while the old could be removed.

Another assessment system that currently is being used is the BOSS (Joy et al. 2005). This system has continued to develop since it was made, and currently has a

web-server component. Through the web-server component, an instructor can review submissions that have been done by students using a web browser. BOSS also has the ability to detect plagiarism in work submitted by students. There are two different phases that this system uses. These are the open-source and the implementation. These are open-source works have been used on all Java platforms by being installed. The implementation format only is used at Warwick University.

Another useful assessment tool is RoboProf (Douce, Livingstone et al. 2005). Daly and his colleagues at Dublin City University developed this in the year 2004. It uses a browser platform where students are provided with an online box where they can write their program. When the program is complete, it will be submitted to the assessment tool that will compile it, run it, and return the results. After the program has been validated, the student is given a chance to proceed to the next programming task. Tasks offered by this system are provided in levels, and the student will move to the next level after completing the previous level. The feature of students being able to move on to the next level by students has increased their chances of better progress in programming (Truong 2007).

Another development undertaken in the field of automated assessment is the Automated System for Assessment of Programming (ASAP). This system was developed at Kingston University and was focused on the Java programming language. Innovative improvements have been made with this system, and various features have been installed. For example, ASAP is able to detect plagiarism, ready submission and

provide results through the program. ASSYST is another program that has been used to relieve tutors from the burden of assessing the many programs used by students. It offers a graphical interface, and has an increased level of automated-assessment criteria. ASSYST provides three stages of assessment, the management of student-exercise submission, the management of student exercises, and the management of the directories and files stored for submission. It also reports on assessment tasks and the grading associated with a report (Jackson & Usher). Figure 2 shows the assessment process of ASSYST system.

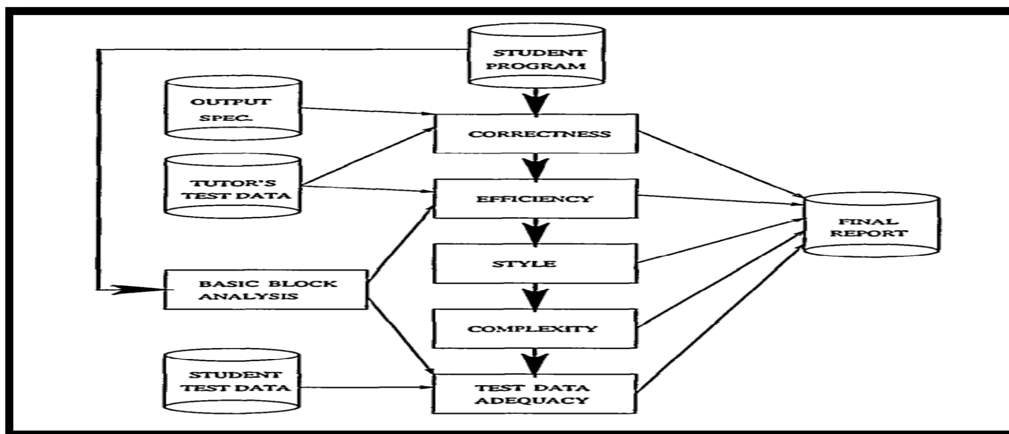


Figure 1 ASSYST assessment process

2.3.1 Benefits of automated assessment tools

Automated-assessment systems have been used in many different areas. As previously mentioned, the first generation was used in 1960, and many different systems since have been used. The automated assessment tools and systems have improved student understanding of the programming language. Consequently, many of

the educational institutions where students learn programming, instructors lack the time to adequately assess each student. With the use of automated systems, students are able to have their exercises properly assessed in a timely manner thus improving professional skills in the programming field.

Quality assignments and the clear formulation of tasks can be expected when automated assessments are undertaken. For any course to be completed successfully, assignments must be well performed. Consequently, when working with programming tasks that will be assessed using automated systems, students must have a clear definition of the programming codes since any error will result in an incorrect answer. In the first generation of the automated systems, the assessment tool gives two answers; namely wrong answer and program complete. For a student assignment to be complete, the program must run effectively and execute correctly. As a consequence, students will have to work hard and improve their programming skills so that their programming will be error-free.

During the marking of student assignments, automated systems can work on some assignments, while other assignments can be assessed by humans. Manual assessment requires the instructor to spend much time with students who may not be present at the required time in some institutions. When students use their own tester programs, many more programs can be submitted and this will increase the number of students who will learn good programming skills (Pieterse 2013).

Regarding teaching programs, assessment is a very important tool for better progress. Assessment can have a strong and positive effect on learning. Students learn

better when they are frequently assessed, and proper feedback is given. On the other hand, the assessment may become hard work for tutors and lecturers alike (Pieterse 2013). Creating exercises and ensuring that the students have completed them takes time and the majority of tutors and instructors fail to have enough time to devote to this task. Tutors will use this time outside doing other constructive work that will enable them to earn money rather than teaching (Haley et al. 2007). Academic institutions have, therefore, adopted computerized marking methods and, in this case, they use automated assessment systems. While such systems are expensive to install, they are highly advantageous compared to human-assessment procedures.

When human are marking, they get fatigued after marking for a long time, and may mark differently. The marking order may also get them confused, and the assessment will not be as effective. For instance, when a human marker comes across a brilliant answer in the first exercise, he or she may become biases when assessing the other exercises. Personal feelings by instructors towards students also can affect the marking criteria sometimes leading to poor assessment (Ribeiro and Guerreiro 2009). Remaining unbiased is one of the reasons why automated assessment has become very important in the field of programming. Students need to be clear in every aspect. Since programming not only deals with the theoretical, the practical part also is very important especially in software designing where error codes may affect the entire program. The results obtained from the automated assessment tools are reliable and will work for long periods without fatigue. The number of students that can be assessed through the use of these systems will be higher when compared to those of human

markers (Rahman and Nordin 2007). There will be no bias, no personal feeling. The results that are obtained from the automated assessment systems will remain exactly the same regardless of the order in which the answers are presented. Therefore, these systems are accurate and can be relied upon (Enström et al. 2011).

2.4 Automated-style assessment tools

As mentioned before, automated-assessment tools provide two types of programming assessment, function analysis and static analysis. There also exist tools that generate feedback about these two types of assessment, including ASSYST, CourseMaster, Web-CAT and Automatic Exams. Other tools such as Style++ and Fortran analyzer generate feedback on the static aspects of programming.

Rees developed one of the early-style tools (Rees 1982); his tool was built to assess Pascal programs. This automated tool suggested 10 measurements to check code style. Five of these measurements were related to layout and other measurements concerning identifiers. As shown in Figure 2, this tool defines five assessor areas and, by testing every line, the tool calculates measurement values. To achieve full marks, the value has to be between two values, which are called lotol, and hitol. If the value falls between lo and lotol or hitol and hi, only part of the maximum mark will be given; while less than lo and more than hi a mark of zero will be given. This tool inspired Berry and Meekings (1985) to work with a tool that can check C code style. Their tool was later the basis of one of the best-known automated assessment tools which was Ceilidh (Benford, Burke et al. 1995).

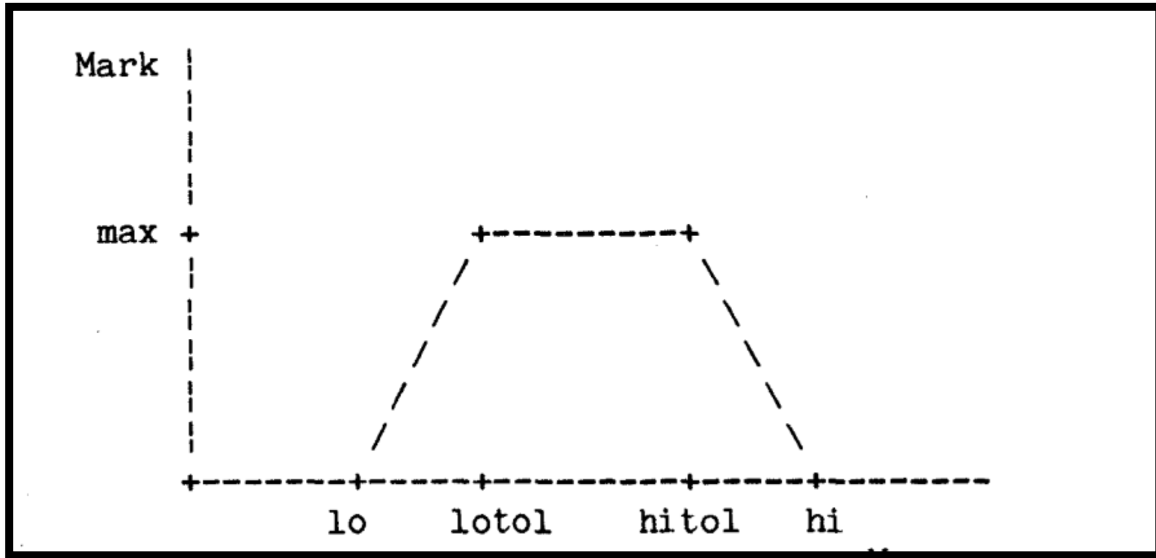


Figure 2 Rees marking system (Rees 1982)

Some years later (Redish and Smyth 1986), the Fortran analyzer was created. Fortran analysed style based on a different approach. The Fortran analyzer focused on stylistic factors of programming. It proposed six measurements to assess code style. The measurements are modularity, simplicity, economy, structure, layout and documentation.

Style++ was developed in 2004 and has been used in the C++ programming for the UNIX environment, which is the standard environment where programming course work is done. Style++ uses six metrics for program style. These metrics are modularity, typography, clarity, independence, effectiveness and reliability. The program style was designed to have output that resembles the compiler. Controlling the output and the content of the program will happen when different options are given to the style-

assessment tool. For instance, the outputs from the style assessment can give scores and comments on the entire program. Comments also can be about those areas in the program that are defective and need to be changed (Ala-Mutka et al. 2004).

One of the current currently created to check program style is Checkstyle. Programming languages such as Java use a Checkstyle-assessment style that helps programmers to write program codes in Java. This tool is used to check the Java codes that have been used in a computer program (Burn 2003).

Checksyle is highly coded, and so can support various coding standards. Common features of Checkstyle are the ability to check aspects of the source code, determine method-design problems and class-design problems. Code layout and formatting issues can be displayed in this style, which improves the ability of students to increase their programming skills.

2.4.1 Measurements of automated style assessment

There are the varied approaches of automated-assessment system that assess different features. Al-Mutka (2007) classified these tools according to their functionality into Dynamic assessment, and Static assessments. Dynamic assessment is an approach that assesses programming code after its execution while static assessment assesses a source code without executing it.

Programming style is the manner in which the students use a different programming language to write program codes. Programming style is a very important tool in the field of programming. A program that has been created using the wrong

codes will be hard for other programmers to understand. Additionally, the outcome of such a program will fail. When a program tester runs against such a program, errors will be displayed, and the execution protocol will fail. Programming languages like the C++ will need special attention when coding because the coding is language dependent. Errors in this programming language are intensely avoided to ensure that the program will be able to run. It is, therefore, a requirement that programmers follow the required guidelines and rules that reduce the chances of the errors that eventually will make the program fail.

Different programming-style assessment has been used to ensure that learning in programming is efficient. Care and attention should, therefore, be paid when developing assessment practices to be used across the different courses. In the Tampere University of Technology, most of the programming was done using the C++ programming language. Therefore, an automated C++ assessment style was developed to ensure that the students in this university followed the rules in this programming language (Ala-Mutka et al. 2004). Other programming' languages will have different assessment styles since coding in the various languages have been found to be quite different from each other.

There are different categories that the coding-practices measurement features are based on. These are modularity, typography, clarity, independence effectiveness and reliability. Modularity needs for inherited instructors and destructors in the inherited classes. This category also includes the use of friends, general summarization, pointers

and private data (Ala-Mutka, Uimonen et al. 2004). Typography comprises of the commenting practices, issues pertaining the layout of the code and naming conventions. Clarity and simplicity also is determined by length of the different codes, the short circuit statements, and the inherent blocks and braces. Independence deals with return values that are correct and avoid numerical literals. Effectiveness is based on the size of the variable scopes, which must be small (Ala-Mutka, Uimonen, & Järvinen, 2004). Finally, regularity is based on setting the pointer to zero if there is deletion of the memory blocks. Instructors who are assessing programming assignments should bear in mind the features of the program style, the acceptance levels of each of feature, verbal feedback messages and the weighting factors given to each feature.

Style assessment also includes measurements of a code combining the use of code metrics. Complexity of a software code is one of the factors that affect how reliable and maintainable a code is. Code-metric measure complete programs, which helps in the evaluation and testing of the code thus establishing where a code needs revision or a complete rewrite. Code metrics create better understanding of the types and methods that need revising within a program.

Some code metrics include; maintainability index, which determines how easy it is to maintain a code. The measure ranges between 0 and 100 where the highest value represents better maintainability. Another code metric is cyclomatic complexity, which determines the code's structural complexity. Structural complexity is about the flow of a

code. The number of paths of flow in a program using a complex flow is difficult to control, and requires many tests to ensure maximum coverage with low maintainability. Depth of inheritance is another code metric. This measures how well one understands a certain code and depends on the number of class definitions extending to class hierarchy root. It is more difficult to understand a code with a deep hierarchy in comparison to a code that endless defined and redefined fields or methods. Lines of code is a common term to all programmers but they do not know it is a code metric. A large number of lines in a method or type indicate that the particular method is loaded with too much work and requires splitting. Large number of lines also indicates difficulty in the maintenance. Class coupling is another code metric. Low coupling and high cohesion is highly recommended for a good software design. High coupling makes a design difficult to maintain and reuse due to several interdependencies.

CHAPTER 3

3 Design

3.1 Overview

In order to assess a novice program, style it is important to define what is good programming style. There are many studies that developed measurements to define what good style is and some of them are presented in the literature. One of these studies is Rees (1982). Rees presented 10 measurements that were based on readability. Consequently, after that (Oman and Cook 1990) proposed taxonomy and guidelines for a good code style, which involved four elements: general practices, typography, flow control and information structure style. (Mäkelä and Leppänen 2004) suggested four measurements to assess the quality of code style. These measurements are visual aspects of the code, program structure, semantic and logical. However, there is an ongoing debate over what metrics should be chosen to determine the quality of code style. In general, it is obvious that, the goal of using these measurements is to make the code readable and able to maintain

3.2 Design requirements

In order to investigate programming style feedback, I designed and built a prototype Automated Feedback for Style Assessment (AFSA) tool. Since there is no perfect standard or one guideline for code style, AFSA strives to adopt the most common convention and provide a consistent style guideline for programmers. The goal of designing the tool is using the principle of programming style guidelines in order to enhance the readability of the code. AFSA use these assessment factors that been

used to judge the style aspect of codes. These factors are layout, name choice, complexity, documentation and efficiency. The implementation of these factors will be discussed in the analysis chapter. Java programming language was the language that used to implement this tool. AFSA tool is designed to detect errors related to code style and complexity. Some old measurements that had been checked by previous tools are no longer applicable since they now are self-assessed by the compiler. Thus, a variable start with capital letters will be detected by the compiler and also will start with special characters. The AFSA doesn't detect the compile-time errors because the compiler already would detect these types of errors. It is important that the code already be compiled successfully because the AFSA relies on this. If the code is not compiled correctly, then the AFSA may not work properly. For example, the AFSA uses open and close brackets to extract the block of methods and if the bracket count is incorrect. Conversely, the AFSA uses the programmer's code at a textual level to analysis the style of the code and generates feedback on it.

3.3 Different languages

Tools are designed to check and analyse the style of code for different languages, including C, C++, C# and Java. There are many cases where different approaches needed to be considered for each computer language. Some languages are object-oriented and contain classes and methods, but some languages like C does not contain a class, so it was necessary to consider both structures. Each language also has a different set of reserved words that needed to be considered when analysing names of the variables. At the start of running the AFSA, it reads the file name from a given

location and determines the language it is written in by considering the file extension. It then applies the relevant settings for that specific language to be considered and also used in other parts of the analysis.

3.4 Code Structure

Code structure consists of different packages and classes based on the functionality of each part. The following are descriptions:

- **Main class**: This is responsible for creating all the necessary objects ensuring they are called in the right order and given the right inputs in order to provide the necessary results for correctly undertaking the procedure of reading files. Also for processing the context, evaluating the code, and returning the results and feedback.
- **Constants**: This is a class for defining all the constants required throughout the code. Rather than repeating these variables they are stored in one location as static variable that can be used anywhere in the code. Examples of these constant values include:
  - Messages to be displayed for the different errors,
  - The necessary values (such as min, max) to be used in some of the evaluations.
  - Reserved words in each programming language that should be ignored during part of the evaluation.

- **Language**: is an enum class that defines and represents different languages and some related properties, such as file extensions for each language used to find the relevant language of code written in the given files.
- **FileHandler**: Handles file reading and writing to scan code from the input files and write the evaluation/feedback to the output files. After reading the file from a given path, it then is returned as a map of line number to the line text. This helps in the future process of reading the content line-by-line and evaluating it with feedback on each line. FileHandler also contains a method for reading all files within a folder, so it could simplify the given file path for each file.
- **MethodBlock**: This class is used to represent a method block by holding information about the method name, content (body), and start line. This mostly is used when evaluation is required per each method, such as the code complexity, or method documentation.
- **JazzySpellChecker**: This class is provided from an external library ('jazzy-core-0.5.2.jar' obtained from (Idzelis 2003) http://jazzy.sourceforge.net/), which is called the Jazzy Library and is used as an English spelling check of words in order to evaluate if the variable names have some meaning. There also is a dictionary file added ('data/dic/dictionary.txt') that is used by the class. Additionally, there are extra files added to the data folder for testing purposes.
- **Feedback package**: Contains classes for providing feedback. These classes include:

- **ErrorImpact**: This is an enum defining the impact of an error for the purpose of providing feedback, because different may be of different importance.
- **ErrorType**: This is an enum defining the type or category of errors for the sake of providing better overall feedback of a number of errors for each category. Some of the types include: White space, indentation, complexity, variable name, and documentation.
- **ErrorMsg**: This is a class that represents an error, and includes the error message text, the error type, and the error impact, as well as the line number that the error occurred in order to provide a better feedback.
- **GradingScale**: This is an enum so as to define the different range of scores into a specific category of High Distinction, Distinction, Credit, Pass, or Fail.

- **Evaluator**: This is the most important and largest class that uses logic for assessing student-code style. It contains various methods for checking different factors for code style such as white spaces, indentation, naming, documentation, complexity, and efficiency. In the following chapter, each of these aspects is explained in further detail.

CHAPTER 4

4 Analysis

4.1 Assessment Factors

The AFSA tool considers a number of assessment factors to analyse the style of each code. Factors considered to check the style are as follows:

- Variables and Operators Spacing
- Blank line
- Line length
- Indentation
- Name choice
- Complexity
- Documentation
- Efficiency

In the following sections, each of the assessment factors is described and the approach taken to implement checking and analysing code based in them is explained.

4.1.1 Code Layout

One of the key factors that used to assess the code style is the layout of the code. By having an organized layout, programmer will be able to follow and understand the program. The code layout can be examined from different aspects such as: space between words and operators, blank line, line length and indentation.

4.1.2 Variables and Operators Spacing

Convention, dictates that spaces should be used to separate binary operators and not be used with unary operators (Oracle, 1999). The AFSA uses one space to separate operators, which is defined in the Constants class. This can further be modified if more spacing is required. The algorithm goes through each line of code and checks how

much spacing exists between variable names and operators. The text for single line and block comments are ignored when checking spacing as this is only considered for the actual code.

The code line is then converted to display an array of characters plus the white space character is counted between each character. Spacing should not be too much or too little. The algorithm checks that there is not a large gap (more than one spacing) between the variables and the operators. It also checks that the operators contain one space between the adjacent variables. If the amount of space is incorrect, then an error message is created to indicate in which line the spacing error is.

There are some exceptions for operator characters that are joined such as ‘++’ or ‘>=’ and with these, there should not be any spacing between the operators. Figure 4 shows feedback generated from AFSA system that checks spacing between words and operators of student samples in Figure 3. At line 33, there is no space before and/or after the equal sign ‘=’ and also on the same line, there is a statement i<argc; with no space between the operators <. The same error occurred again at line 37 and 42. This symbol ‘ | ‘ refers to the start of indentation.

```
23: |  int _value;
24: |};
25: |#include <iostream>
26: |#include <string>
27: |#include <math.h>
28: |using namespace std;
29: |int main(int argc, char** argv)
30: |{
31: |  char* s;
32: |  bool british;
33: |  for (int i=0; i<argc;i++)
34: |  {
35: |   if (argv[i][0]=='-')
36: |   {
37: |    for (int j=1; j<sizeof(argv[i]);j++)
38: |    {
39: |     switch (argv[i][j])
40: |     {
41: |      case 'b':
42: |       british=true;
43: |       break;
44: |      default:
45: |       break;
46: |     }
```

Figure 3 Sample of student programming exercise

```
- Line 33: (White Space error) Not enough space for the operator. There should be 1 space between operators.
- Line 37: (White Space error) Not enough space for the operator. There should be 1 space between operators.
- Line 42: (White Space error) Not enough space for the operator. There should be 1 space between operators.
```

Figure 4 The AFSA feedback regarding space between words and operators errors

4.1.3 Blank line

A well-styled code should have a good balance of empty lines between code lines. Using a blank line makes the code both readable and understandable. Conversely, too many blank lines between the code lines make it difficult to read and understand the code. Therefore, the AFSA algorithm checks the code to ensure that the blank lines do not exceed the configured constant number.

The way an algorithm works is by going through each line and using the String *trim()* method to remove spaces and tabs. It then compares the trimmed string to see if it equals the empty string. If the line is empty, then the number of blank lines is incremented. Then, the following line is checked. If the line is not empty, the counter is reset to zero otherwise it continues incrementing. After each increment, the counter is checked to see if it exceeds the configured constant number, which in this case is set to two. Also, if the number of blank lines is more than the constant then a style error is created to prompt the user. Figure 6 shows the AFSA feedback about the blank line of the student sample Figure 5. There are three lines empty, which styles the code layout as disorganized.

```
6:  |class Wordnum {
7:  |public:
8:  |     Wordnum();
9:  |     Wordnum(int n) {
.0: |         value_ = n;
.1: |     }
.2: |     friend Wordnum operator+(const Wordnum& n1, const Wordnum& n2) {
.3: |         return Wordnum(n1.value_ + n2.value_);
.4: |     }
.5: |     friend Wordnum operator-(const Wordnum& n1, const Wordnum& n2) {
.6: |         return Wordnum(n1.value_ - n2.value_);
.7: |     }
.8: |     friend Wordnum operator*(const Wordnum& n1, const Wordnum& n2) {
.9: |
0:  |
1:  |
2:  |         return Wordnum(n1.value_ * n2.value_);
3:  |     }
4:  |     friend Wordnum operator/(const Wordnum& n1, const Wordnum& n2) {
5:  |         return Wordnum(n1.value_ / n2.value_);
6:  |     }
7:  |     friend std::ostream& operator<<(std::ostream&, const Wordnum& n);
8:  |     friend std::istream& operator>>(std::istream&, Wordnum& n);
9:  |private:
0:  |     int value_;
1:  |};
```

Figure 5 Sample of student programming exercise

```
- Line 21: (White Space error) There are 3 empty lines
```

Figure 6 AFSA feedback for blank-line errors

4.1.4 Line length

As a convention for code style, it is good to keep the length of the code lines up to a certain length so it is easier to read (Android 2015). When the code length is too long it moves off the screen and the programmer has to scroll to the left and write when trying to read and understand the code.

The AFSA tool goes through the lines of code one by one and compares the length of the line with the configurable constant value, which in this case is set to 160 characters. If the length of the line is bigger than this constant, then the style error is created to act

as a prompt to the user. As shown in **Figure 7** the line length of line number 47 exceeded the limit of the screen view, which forced the programmer to scroll to the right to see the rest of the code line. Figure 8 is illustrates the feedback that generated from the tool about this bad style

```
45: |#include <iostream>
46: |using namespace std;
47: |string singular_ [] = {"zero", "one", "two", "three", "four", "five", "six",+"seven", "eigh
"nine", "ten", "eleven", "twelve", "thirteen", "fourteen",  +"fifteen", "sixteen", "seventeen",
"eighteen", "nineteen"};
48: |
49: |string tenprefix_ [] = {"void", "void", "twenty", "thirty", "forty",
50: |    +"fifty", "sixty", "seventy", "eighty", "ninety"};
51: |string triprefix_ [] = {"void", "hundred", "thousand", "million"};
52: |Wordnum::Wordnum(int n) {
53: |    value_ = n;
54: |}
55: |int string_into_int(string s) {
56: |    int result = 0;
57: |    int signconverter = 1;
58: |    size_t pos;
59: |    for (int i = 0; i < s.length(); i++) {
```

Figure 7 Sample of student-programming exercise

```
- Line 47: (Indentation error) The length of line is 215 which is bigger than the standard line length of 160
```

Figure 8 The AFSA feedback for line-length errors

4.1.5 Indentation

Indentation refers to whitespace added at the beginning of each code line in order to suggest code structure. There are different conventions in regards to what size the indent should be and this may vary between one language and another. For example, here are the conventions of the code-indentation size for Drupal and Python languages. Drupal.org and JavaScript use two spaces as a guideline for indentation (Gallagher 2000), whereas Python considers four spaces to be the standard regarding indentation (Guido van Rossum. 2001).

Regardless of the indentation size, it is important there be consistency throughout the code. With AFSA, the approach is to try to determine what is the most common indentation size used by a programmer in the code file.

This is accomplished by testing the provided code with different indentation sizes ranging from a configurable minimum and maximum indentation, and finding the value that has the least number of errors for indentation. In this case, the minimum and maximum indentations are configured as 1 and 20. For each integer within this range the code is analysed to see how different the actual code indentation is from the current value (representing the indentation). Algorithms add the number of lines that are not aligned with the current indentation size. At the end, the indentation size that has the least number of misaligned lines is selected as the final indentation size that is considered best for examining a given code.

The chosen indentation is checked to ensure it is within the allowable indentation size range that is configured as 2 to 8. If the chosen indentation size is out of this range an error with a high-impact ratio is created to indicate that the indentation is not good.
After finding the correct indentation size, the algorithm goes through each line of code and checks if that line is aligned or misaligned with the indentation and adds up the number of lines that are not following the indentation. The comments and empty lines are not considered in this calculation and are ignored.

The algorithm counts the number of empty space character by character to determine the number of indentations. An issue that needs to be considered is that some empty spaces might occur because of the tab insertions. A tab is considered to be a single character but is equivalent to four spaces. Moreover, a tab has different interpretations over different environments. Therefore, the tab characters are temporarily replaced with a space for checking the indentation.

The code indentation would change based on methods or different code syntax such as loops and conditional statements. In order to determine the correct indentation in the code, the algorithm considers the number and state of open/close curly brackets. For example, the code inside of a class should be indented one level higher than the class. Similarly what comes inside of a method block should be indented a level higher that the class-level code. So every time an open bracket is detected, the following codes should be indented a level higher repeatedly until the matching closing bracket is reached. Some special cases need to be considered because of the location of the open bracket at the end of the same line or the start of the following. Figure 9 shows

student sample that has zero indentation errors. The calculated indentation for this sample was 1 space, even though this indentation size does not generate any indentation errors but it is not desirable. Because using one space for indentation affects the readability of the code.

```
130:  |void atobi(char *szBigInt, int *a_iBigInt) {
131:  | int iLength = strlen(szBigInt), iCount = 0;
132:  | for (iCount = 0; iCount < iLength; iCount++) {
133:  |  a_iBigInt[iCount] = szBigInt[iCount] - 48;
134:  |  a_iBigInt[iCount + 1] = -1;
135:  | }
136:  |}
137:  |void bitoa(char *szBigInt, int *a_iBigInt) {
138:  | int iCount = 0;
139:  | memset(szBigInt, '\0', 255);
140:  | while (a_iBigInt[iCount] >= 0) {
141:  |  szBigInt[iCount] = a_iBigInt[iCount] + 48;
142:  |  szBigInt[iCount + 1] = '\0';
143:  |  iCount++;
144:  | }
145:  |}
146:  |void itobi(int *a_iBigInt, int iNormalInt) {
147:  | char szNormalInt[255];
148:  | sprintf(szNormalInt,"%i", iNormalInt);
149:  | atobi(szNormalInt, a_iBigInt);
150:  |}
151:  |void bigint_increment(int *a_iAnswer, int iAmount) {
152:  | int a_iAmount[255], a_iIntermediate[255];
153:  | memset(a_iAmount, -1, 255);
154:  | memcpy(a_iIntermediate, a_iAnswer, 255);
155:  | itobi(a_iAmount, iAmount);
156:  | bigint_add(a_iAnswer, a_iIntermediate, a_iAmount);
157:  |}
158:  |int bigint_iszero(int *a_iBigInt) {
159:  | int iReturn = 0;
160:  | bigint_trim(a_iBigInt);
161:  | if (a_iBigInt[0] == 0) {
162:  |  iReturn = 1;
163:  | }
164:  | return iReturn;
165:  |}
```

Figure 9 Sample of a student-programming exercise

Figure 10 is an example of bad indentation. The calculated indentation for code is four spaces, which generate many errors that shown in **Figure 11**. At line 104 of **Figure 10** the line indented by four spaces, which is the calculated indentation size. So the next line, which is 105, should keep the same size of indentation because there is no open

bracket that increases the level of indentation. The programmer has the same level of indentation for the next lines, 106 and 107 which means incorrect indentation. Following the correct indentation level will result in nested indentation, which increases the code readability.

```
100: |     string thousands[] = {"","M","MM","MMM"};
101: |     string output[10];
102: |     int count = 0;
103: |     while(count < 3){
104: |         cin>> roman;
105: |     output[count]= roman;
106: |     char c;
107: |     for(int i = 0; i <= roman.length()-1; i++){
108: |         roman[i] = toupper(roman[i]);
109: |     }
110: |     int length = 0;
111: |     arabic = 0;
112: |     while(true){
113: |     for(int t = 3; t >= 1; t--){
114: |         length=thousands[t].length();
115: |         if(thousands[t]==roman.substr(0,length)){
116: |             arabic += t*1000;
117: |             roman.erase(0,length);
118: |             if (roman == "") goto print;}}
119: |     for(int t = 9; t >= 1; t--){
120: |         length=hundreds[t].length();
121: |         if(hundreds[t]==roman.substr(0,length)){
122: |             arabic += t*100;
123: |             roman.erase(0,length);
124: |             if (roman == "") goto print;}}
125: |     for(int t = 9; t >= 1; t--){
126: |         length=tens[t].length();
127: |         if(tens[t]==roman.substr(0,length)){
128: |             arabic += t*10;
129: |             roman.erase(0,length);
130: |             if (roman == "") goto print;}}
```

Figure 10 Indentation of 2 spaces

```
- Line 105: (Indentation error) The indentation is incorrect to start at 4, it should start at 8
- Line 106: (Indentation error) The indentation is incorrect to start at 4, it should start at 8
- Line 107: (Indentation error) The indentation is incorrect to start at 4, it should start at 8
- Line 108: (Indentation error) The indentation is incorrect to start at 8, it should start at 12
- Line 109: (Indentation error) The indentation is incorrect to start at 4, it should start at 8
- Line 110: (Indentation error) The indentation is incorrect to start at 4, it should start at 8
- Line 111: (Indentation error) The indentation is incorrect to start at 4, it should start at 8
- Line 112: (Indentation error) The indentation is incorrect to start at 4, it should start at 8
- Line 113: (Indentation error) The indentation is incorrect to start at 4, it should start at 12
- Line 114: (Indentation error) The indentation is incorrect to start at 8, it should start at 16
- Line 115: (Indentation error) The indentation is incorrect to start at 8, it should start at 16
- Line 116: (Indentation error) The indentation is incorrect to start at 12, it should start at 20
- Line 117: (Indentation error) The indentation is incorrect to start at 12, it should start at 20
- Line 118: (Indentation error) The indentation is incorrect to start at 12, it should start at 16
- Line 119: (Indentation error) The indentation is incorrect to start at 4, it should start at 16
- Line 120: (Indentation error) The indentation is incorrect to start at 8, it should start at 20
- Line 121: (Indentation error) The indentation is incorrect to start at 8, it should start at 20
- Line 122: (Indentation error) The indentation is incorrect to start at 12, it should start at 24
- Line 123: (Indentation error) The indentation is incorrect to start at 12, it should start at 24
- Line 124: (Indentation error) The indentation is incorrect to start at 12, it should start at 20
- Line 125: (Indentation error) The indentation is incorrect to start at 4, it should start at 20
- Line 126: (Indentation error) The indentation is incorrect to start at 8, it should start at 24
- Line 127: (Indentation error) The indentation is incorrect to start at 8, it should start at 24
- Line 128: (Indentation error) The indentation is incorrect to start at 12, it should start at 28
- Line 129: (Indentation error) The indentation is incorrect to start at 12, it should start at 28
- Line 130: (Indentation error) The indentation is incorrect to start at 12, it should start at 24
```

Figure 11 AFSA feedback for indentation

4.2 Name Choice

Names of variables and methods are extracted for analysis if they have a good name. A good name is considered as a name that is meaningful and descriptive, and is the right length, meaning that it is not too long, or too short. Names at the higher-level scope, such as class-level variables, it is more important to follow this rule.

Also, names may consist of other combined names. Convention dictates that, when joining words together to make a longer name, they are separated by a '_' or by making

the first letter of following word uppercase (Camelcase). The program checks that the name of variables, classes, and methods are meaningful. It does this by breaking the word into its sub-words and using the SpellChecker from Jazzy library to see if the word exists in the dictionary of words.

Regular expressions are used frequently to find relevant words from the code text.

An algorithm works as follows:

To loop through each line of code, do the following for each line of code:

a. Temporarily remove the comment and text within quotations
b. Trim the line to remove empty spaces at the both ends and ignore the empty lines
c. Use a regular expression to split the code line statement based on ' ', ';', '[', '(', '{', '=', ‘*’, ‘&’ in order to find names, because these characters are usually placed next to a name (variable, method, class), when they are mentioned for the first time. It considers different languages such as C where the (‘*’, and ‘&’) are used for pointers:

```
String[] wordsInLine = trimedLine.split(" |;|\\[|\\(|\\{|\\=|\\,|\\*|\\&");
```

d. For each word in the line, do the following:
    i. If the word is already checked, continue on to next word
    ii. Check if the word is not a reserved word for the language of this code (that’s already determined in other parts of the code).

iii. Use a regular expression to check if the word contains numbers or alphabetical characters:

```
Pattern pattern = Pattern.compile("[a-zA-Z0-9_]{1,}");
matcher = pattern.matcher(name);
if (matcher.matches()) ...
```

iv. If the previous condition is true, it means that the program has reached a word that is a feasible name

v. Check the length of the name to ensure it is not too short or too long based on the configured values

vi. Then the name is split into its sub-names (if any exists) based on the separator or camelcase naming, using the following regular expression:

```
String[] nameSlices =
 name.split("(?<!(^|[A-Z]))(?=[A-Z])|(?<!^)(?=[A-Z][a-z])|_|[0-9]");
```

vii. For each sub-name, the JazzySpellChecker is used to check that each sub-name exists in the dictionary. If not, it will add the word to a list that keeps track of the misspelled algohm.

For every misspelled word add an error message for wrong naming. The feedback about name- choice factors is depicted in Figure 13 for the sample that was selected in Figure 12. There is a class variable at line 4, which is WrongNaming. The

system didn’t detect any errors for this variable after breaking it down into two words. The words wrong and naming are found in the dictionary file. If the second word naming is not capitalized the system feedback will ask to restructure the variable. At line 11 and 24 there are misspelling variables. The tool only generated feedback on the first error that occurred and ignored the second one.

```
01: |package data.currentSample;
02: |
03: |
04: |public class WrongNaming{
05: |
06: |    int j=3+2;
07: |    private String godoada[];
08: |    private int nunber;
09: |    private int aaAAppple;
10: |
11: |    public WrngNaming(int j)
12: |   {
13: |           int a = 0;
14: |        int classname = 0;
15: |        int number = 0;
16: |        int name = 0;
17: |          if ((a=(n/(1)))==(b=j)) {}
18: |
19: |          if (nunber == 2) nunber = 3;
20: |          j++; j--;
21: |   }
22: |
23: |   // hi
24: |   public WrngNaming() {
25: |          int ladada= digit1a;
26: |
27: |          // TODO Auto-generated constructor stub
28: |          System.out.println (" // dadfadak ");
29: |
30: |          for (int i = 0; i < godoada.length; i++) {
31: |
32: |          }
```

Figure 12 sample of student programming exercise

```
- Line 07: (Variable Name error) The spelling godoada is not correct or not structured well
- Line 08: (Variable Name error) The spelling nunber is not correct or not structured well
- Line 09: (Variable Name error) The spelling aaAAppple is not correct or not structured well
- Line 11: (Variable Name error) The spelling WrngNaming is not correct or not structured well
- Line 14: (Variable Name error) The spelling classname is not correct or not structured well
- Line 25: (Variable Name error) The spelling ladada is not correct or not structured well
```

Figure 13 The AFSA’s feedback for name-choice errors

4.3 Method Extraction

In order to analyse some aspects of code, it is necessary to determine which parts of text are related to a method. For example, knowing the content and location of s method is important when measuring code complexity, the existence of method documentation, and efficiency of code. These assessments will be covered in the following sections. This section only talks about how methods are extracted and determined from the code text.

The program extracts method blocks and stores them as a List of MethodBlocks, which then can be fed to other methods to individually assess each method. The MethodBlock contains information, such as the name and content of a method, and the starting line number. The methods are extracted through the following procedures:

The program goes through each line of code to determine the start and end of a method by looking for special indicators. The most important information is found by determining where the method is through using open and close curly brackets '{'. Also, '}', which are common indicators for the start and end of the methods in the various languages examined in this project. Although, using brackets as an indicator for determining method structure is a good approach, one thing to be considered is that not all curly brackets indicate the start or end of the method. Some indicate the statements, loops or class structure. Therefore, it is important to consider the level of the brackets as well, which means the position of the nested bracket (the brackets within the other open brackets).

For different languages, the method-level brackets are at different levels. For example, in object-oriented language such as Java and C#, the first open bracket usually indicates the start of a class (or interface, enum, etc.), and the method is started with the second nested open bracket. The constructor could also be considered as a method or referred to as one for simplicity. In the procedural languages such as C, there is no class-level bracket (as there is no class). Therefore, the first open bracket often indicates the start of a method.

The program considers the type of language in order to know what level of nested brackets to use for measuring the start of a method. For example, the procedural language of C uses the first-level bracket as method, and object-oriented programs of Java and C# use the second-level bracket to determine the methods.

To simplify the process of finding methods, unnecessary texts within the code is ignored, such as comments or text within the quotations. Then, depending on the type of language, the program goes through each line and searches for open and close brackets. It also keeps the count of the level of bracket. The level starts with 0, then is incremented by 1 every time a '{' is detected and decremented by 1 when a '}' is detected. Then as the bracket-level count changes, the program checks level of bracket. If the bracket-level is changed from a higher level down to the method-level, then it recognises that it has entered a method so it begins saving the content of the data being read, until the bracket-level count changes from method level to the higher level (e.g. class-level). At this point it senses that it has come out of the method and

therefore creates a new MethodBlock object to store the recorded content of the method, before saving the object to the List of MethodBlocks.

4.4 Complexity

There are many different approaches for calculating the complexity of code. In this case McCabe's Cyclomatic Complexity approach is used to measure flow complexity.

One of McCabe's original applications was to limit the complexity of routines during program development (McCabe 1976). This is measured by counting the complexity of modules that been developed, and splitting them into smaller modules whenever the Cyclomatic complexity of the module exceeded 10.

The flow complexity of a method is measured by counting 1 for each place where the flow changes from a linear flow (Swartz 2007), such as: conditional statements (e.g. if, else, case), loops (for, while, do-while, break, and continue), operators (e.g. &&, ||, ?, :) and returns.

In order to measure the complexity of each code module, first it is required to identify/extract the modules. The code modules are different in each specific language depending on whether the language is object-oriented. In object-oriented languages (such as Java and C#) the code module is the methods within a class. In the procedural language (such as C) there is no class so the modules are simply the methods. For "multi-paradigm" languages (such as C++), which can be both object oriented or procedural, extra consideration is required to ensure code blocks are considered correctly.

The algorithm for measuring complexity first extracts the code modules (as explained in the Method Extraction section) and then stores the information in a map of code-module identifiers (e.g. method name) within the module content. The code module is represented by the MethodBlock class as described in the previous section, which contains the content, identifier, start line and method complexity.

Once the code blocks or modules (e.g. methods) are identified, each of them is passed into the method for measuring complexity, which does the following:

- Temporarily remove comments and quotation text, and forces lines to ignore comments and empty lines
- Initialize a counter for complexity starting with value of 1
- Use regular expression to split the code line around the end of line or characters of ‘;’, ‘[’, ‘{’ ‘(’, ‘)’, ‘=’, ‘*’, ‘,’.
- For each splitted word, check if the work is a reserved word related to McCabe’s approach, increment the complexity counter by one if it is.

Figure 14: Shows the way the tool provides feedback about the complexity of each block. The tool also shows the line number of blocks and provides feedback on complexity.

```
Measuring the complexity of code...
-- The complexity of the method  Transpose() on line 19 is ok
-- The complexity of the method  ReadFromString() on line 23 is ok
-- The complexity of the method  Equal() on line 40 is ok
-- The complexity of the method  PlusEqual() on line 45 is close to high
-- The complexity of the method  MinusEqual() on line 58 is close to high
-- The complexity of the method  MultiEqual() on line 68 is ok
-- The complexity of the method  DivideEqual() on line 87 is ok
-- The complexity of the method  Greater() on line 110 is very high
-- The complexity of the method  GreaterEqual() on line 120 is ok
-- The complexity of the method  EqualN() on line 123 is ok
-- The complexity of the method  *ToString() on line 132 is very high
-- The complexity of the method  Transpose() on line 170 is ok
-- The complexity of the method  *GetInput() on line 177 is ok
-- The complexity of the method  IsOperator() on line 181 is ok
-- The complexity of the method  main() on line 184 is very high
```

Figure 14 The AFSA's feedback on complexity code

4.5 Documentation

The program checks for the existence of documentation regarding the methods. The documentation could be single-line documentation as indicated by '//' or a block comment as indicated by '/*' or '/**', and '*/'. This part of assessment takes advantage of the Method Extraction process. One of the initial approaches taken was to count the number of methods extracted, and compare it with the number of block comments. However, this approach did not seem to be effective, as it did not ensure that the block comments were for the corresponding method, because the block comment could be added to any part of the code. By academic convention, the comments for methods should be placed immediately before the method definition. Therefore, a different approach was considered to ensure that each method was commented on. The procedure for this was as follows:

The method-extraction procedure is known as the ‘get the list of MethodBlocks’ in the code. The MethodBlocks contain the start-line number for the method. Therefore, the program begins from the start of the method, then goes backwards to check the previous lines to see if any indicator is found for the single-comment line or the end of the block comment immediately prior to the method. If there are empty lines or space in the previous lines prior to the method, the program ignores them and continues to the lines before the previous one. It stops as soon as it reaches a line that does not have a comment indicator, which means that the method being assessed was without documentation.

4.6 Efficiency

The efficiency is that the number of resources that been used in order to solve a certain task (Carmichael 2002). With this tool the efficiency of code is measured by counting the total number of lines of code written by the student and then comparing it with the line of code written by the instructor. Lines are defined by the lines that finish with ';'. The reason for this consideration is that some lines might be segmented and placed on the following lines for visual purposes. The aim is to measure the lines of code that have some values or importance in terms of use of resources. Therefore, the comment lines or empty lines, or what is in quotations for print statements is ignored. Also, the curly brackets are removed because often they occupy the whole line for the sake of styling the code as shown below, and the code style should not affect the efficiency. For example:

```
if (condition) { ... }
```

or

```
if (condition)
{
        ...
}
```

The above codes are the same in terms of efficiency and use of resources, but the second one occupies two more lines than the first one because of the brackets. Also, for simplicity, it was decided to remove what is inside of the curved brackets '(' and ')' of *for* loop. That is because different implementation could affect the efficiency count as the program is written to count ';' as a new line and some implementation of *for* loop has different number of ';'. For example, the first implementation of *for* loop has two semicolons, whereas the second implementation has 0 semicolons but, in terms of the number of resources, they both use a single variable:

```
for (int i = 0; i < MAX; i++) // has 2 semicolons

for (Integer i : values) // has 0 semicolon
```

After removing all the unnecessary parts of code, the code lines for each file is counted and added to the total sum of all the files provided by the student. The same procedure is done with the code provided by the instructor and the total line is calculated. Then the student measurement is compared to the tutor's code and the difference calculated. Finally, the difference is checked against some defined ranges to provide clear feedback to the student.

4.7 Feedback

Providing constructive feedback is very important in order to help the student or the instructor understand exactly what the inherent problems are in providing feedback. The ErrorMsg class was created to store information on the errors such as the type and impact of error, the error message to be displayed and the line number, in which the error occurred.

The line number and error message serve the purpose of directing the user to where the problem is and what the issue is so it can be assessed or fixed.
The impact of error is used to determine the severity of error and the amount of marks that should be deducted. Most of the errors are of the basic type. Some examples of when the higher impact error types are used include the situation when the code complexity of methods is above the highest defined threshold, or when the indentation is distant from the allowed range.

The error types are defined as the assessment factors such as indentation, whitespace, variable name, complexity, and documentation. This type of error is used as a way of grouping errors with similar issues. This allows for the giving of an overall mark for a different category or aspect of assessment.

The feedback is provided in two ways. Firstly, each line of code that is issue-based in the code assessment is listed to indicate what the issue is as shown in figure 15.

There are 17 errors in the code:

- Error: -- The complexity of the method geTCalculat() on line 38 is very high

- Line 05: (Variable Name error) The spelling WrngNaming is not correct or not structured well

- Line 06: (Documentation error) -- The method WrngNaming() on line 6 doesn't have documentation.

- Line 09: (White Space error) Not enough space for the operator. There should be 1 space between operators.

...

- Line 95: (Indentation error) The indentation is incorrect to start at 0, it should start at -4

Figure 15 AFSA feedbak based on category

Another manner in which feedback is provided is by showing the total number of errors for each category of assessment (e.g. Indentation, Complexity, etc.). The total number of errors for each category is measured and a percentage is calculated in order to determine a specific score for each category. The denominator used for calculating the percentage is different depending on the type of error. For example, for the case of the indentation, the denominator is the total number of lines in the code, whereas in the case of complexity, the denominator is the number of method blocks because it is measured on a per method block basis.

The score then is checked against some defined range of values to determine the mark depending on which range it falls into. The defined ranges include High Distinction, Distinction, Credit, Pass, or Fail as defined by **Figure 16**.

Score	Grade
0-50	Fail
50-64	Pass
65-74	Credit
75-8	Distinction
85-100	Hi Distinction

Figure 16 grade scale

A sample output of the program's feedback is shown below, where it shows the number of errors for the different categories.

```
The category of 17 errors include:

  - 1 errors of Complexity type, mark is 75, grade is Distinction - very good.
  - 10 errors of Variable Name type, mark is 89, grade is High Distinction - Excellent!
  - 1 errors of Documentation type, mark is 75, grade is Distinction - very good.
  - 3 errors of White Space type, mark is 96, grade is High Distinction - Excellent!
  - 2 errors of Indentation type, mark is 97, grade is High Distinction - Excellent!

 -------------> Your overall mark is 86.4%
```

Figure 17 AFSA overall feedback

The overall mark is calculated by taking scores for each category and then averaging them.

The following chapter evaluates the feedback provided by the prototype by using sample of student programs.

CHAPTER 5

5 Validation

5.1 Study description

A study was conducted in order to validate the tool feedback and assess the effectiveness of the programming style feedback of the tool. In addition, the study assesses the difference between manual style marking and automated style marking. The study contains tow parts experiment and survey that was conducted in papers. The targeted participants were academic staffs that have a programming background and teach programming. Six tutors out of twelve agreed to participate in the study.

5.2 Significance of the study

The targeted participants were the academic staffs who has experience in teaching programming, so they are able to provide feedback about the techniques used by the prototype tool and identify technique that have not been covered. Moreover, The targeted participants are familiar with assessing the code manually, which means they have the ability to determine which techniques need to be investigated.

5.3 Experiment

5.3.1 Experimental procedure

Participant was given four scenario packages, each of which contains printed copies of the following information:

- A description of a sample student-programming task that used for a practical activity in a programming topic.

- A code listing of sample student code that submitted as a solution for the programming task.
- A code listing of the student code annotated with style feedback based on analysis from the automated style feedback tool

For each scenario, first, the participant was asked to examine unannotated code listing and consider what feedback he would provide about each of the style attributes. Secondly, the participant was asked to to examine the annotated code listing and consider how accurately the code-generated feedback matches his view of the feedback that would best assist the student. Lastly, the participant was asked to consider any other feedback that would provide if he were manually assessing the code style.

Participant was free to withdraw from the experiment at any stage during the experiment and also free to question about the material. Figure 18 is an example of unannotated code sample that was given to the participant and Figure 19 is annotated code sample that was used to show the participant the generated tool feedback.

```
01: |package data.currentSample;
02: |
03: |
04: |public class WrongNaming{
05: |
06: |    int j=3+2;
07: |    private String godoada[];
08: |    private int nunber;
09: |    private int aaAApple;
10: |
11: |    public WrngNaming(int j)
12: | {
13: |        int a = 0;
14: |        int classname = 0;
15: |        int number = 0;
16: |        int name = 0;
17: |      if ((a=(n/(1)))==(b=j)) {}
18: |
19: |      if (nunber == 2) nunber = 3;
20: |      j++; j--;
21: | }
22: |
23: | // hi
24: | public WrngNaming() {
25: |      int ladada= digit1a;
26: |
27: |      // TODO Auto-generated constructor stub
28: |      System.out.println (" // dadfadak ");
29: |
30: |      for (int i = 0; i < godoada.length; i++) {
31: |
32: |      }
33: | }
34: |
35: |
36: |
37: | private void   geTCalculat() {
38: |
39: |      if ( digit1a == 1) {
40: |           System.out.println();
41: |      } else {
42: |           System.out.println();
43: |      }
44: |
45: |      if ( digit1a == 2) {
46: |           System.out.println();
47: |      } else {
```

Figure 18 unannotated code sample

```
01: |package data.currentSample;
02: |
03: |
04: |public class WrongNaming{
05: |
06: |    int j=3+2;
07: |    private String godoada[];
08: |    private int nunber;
09: |    private int aaAAppple;
10: |
11: |    public WrngNaming(int j)
12: | {
13: |        int a = 0;
14: |        int classname = 0;
15: |        int number = 0;
16: |        int name = 0;
17: |      if ((a=(n/(1)))==(b=j)) {}
18: |
19: |      if (nunber == 2) nunber = 3;
20: |      j++; j--;
21: | }
22: |
23: | // hi
24: | public WrngNaming() {
25: |      int ladada= digit1a;
26: |
27: |      // TODO Auto-generated constructor stub
28: |      System.out.println (" // dadfadak ");
29: |
30: |      for (int i = 0; i < godoada.length; i++) {
31: |
32: |      }
33: | }
34: |
35: |
36: |
37: | private void   geTCalculat() {
38: |
39: |      if ( digit1a == 1) {
40: |           System.out.println();
41: |      } else {
42: |           System.out.println();
43: |      }
44: |
45: |      if ( digit1a == 2) {
46: |           System.out.println();
47: |      } else {
```

the class is not documented

The identifier (j) is too short

The spelling godoada is not correct or not structured well

The spelling nunber is not correct or not structured well

The spelling aaAAppple is not correct or not structured well

The spelling WrngNaming is not correct or not structured well

The spelling classname is not correct or not structured well

Not enough space for the operator. There should be 1 space between operators.

The spelling of ladada is not correct or not structured well

Not enough space for the operator. There should be 1 space between operators.

There are 3 empty lines

The spelling of geTCalculat is not correct or not structured well

There are more than 1 space between words

doesn't have documentation

High complexity

Figure 19 Annotated code sample

5.4 Survey

After the experiment stage, participants were requested to complete the questionnaire. Questionnaire is the most suitable method to collect primary data for the current research as it allows the researcher to receive direct feedback and additional ideas for improvement of project tool. The aim of the survey was to collect ideas and features that the teaching staff rated as the most important for automated feedback tool. The researcher carefully reviewed the analysis of these responses in order to evaluate the tool feedback that generated from the tool. 'See the appendix to see the survey'

The survey consists of 10 closed-ended questions and 2 open-ended questions. Questions were grouped into three logical blocks: "Use of Automated Feedback", "Evaluation of Feedback" and "Additional Information.

5.5 Study results

Use of Automated Feedback

	None	A little	Moderate	Extensive
Introductory Students	1	-	1	4
Intermediate Students	1	-	3	2
Advance Students	1	2	1	2

Table 1: Participants recent experience (last 2 years) in teaching programming

	None	A little	Moderate	Extensive
Introductory Students	1		2	3
Intermediate Students	1	-	3	2
Advance Students	1	1	2	2

Table 2: Participants medium-term experience (last 5 years) in teaching programming

	None	A little	Moderate	Extensive
Introductory Students	1		3	2
Intermediate Students	1	3	2	
Advance Students	1	3	2	

Table 3: Participants long-term experience (last 10 years) in teaching programming

Table 1 shows that majority of participants claimed to have recent extensive experience (last 2 years) teaching programming to introductory students: Four participants chose this option in the first question whereas one participant has a recent moderate experience and one participant does not have a recent experience in teaching programming to introductory students. The second choice in the same question was "moderate experience with intermediate students". Five out of six participants have a moderate and extensive experience whereas one participant has no experience in teaching programming for intermediate students. Finally, with advance student three participants claimed to have moderate and extensive experience of teaching programming to advanced students over the last 2 years whereas the other either have no or little experience of teaching programming.

In the second question **Table 2**, the situation remained largely the same with few changes exceptions. In question 3 which was, how much long-term experience (last ten years) do you have teaching programming? **Table 3** shows that five participants answered that they had moderate and extensive experience of teaching programming to introductory student over the last 10 years. One respondent replied that he had no teaching experience with introductory students. On the other hand, It been noticed that none of the participants have an extensive experience with long term.

Table 1, **Table 2** and **Table 3** show the participants experience in teaching programming to different students levels. It is obvious that the predominant majority of respondents claimed to have moderate and extensive experience teaching programming to students in medium and short terms.

Regarding the participant experience with automatic programming tool assessment, the predominant majority of teaching staff stated that they are aware of automatic programming tools. In particular, five participants claimed to be familiar with "try" program; 1 respondent reported that he is not familiar with "try". And regarding CodeRunner(quiz-based function testing in FLO), all the participants agree that they familiar with it. One respondent have used it in the past and two planning to include the software in their future teaching activities.

Regarding automated style assessment tools in specific, only two respondents are planning to use checkstyle (open-source style checking) in the future. Interestingly, these respondents either have or had moderate or extensive experience of teaching programming to introductory students. That may reflects the important of using automated style tool for novice programmers. The other participants check the code style manually by using common standard.

The majority of respondents who claimed to teach introductory students believe that there is substantial extent to which automated tools can replace manual assessment of code style. Whereas the majority of participants believe that the extent is partial in the case of intermediate students and advanced students correspondingly.

Evaluation of feedback

Indentation	Strongly disagree	Somewhat disagree	Neutral	Somewhat agree	Strongly agree
Feedback is accurate				3	3
Feedback is appropriate				5	1
Feedback is easy to understand			1	1	4
Feedback is helpful to learning				2	4

Table 4: Participant’s opinion about Indentation

Code complexity	Strongly disagree	Somewhat disagree	Neutral	Somewhat agree	Strongly agree
Feedback is accurate			2	3	1
Feedback is appropriate			3	2	1
Feedback is easy to understand		1	3	1	1
Feedback is helpful to learning			2	3	1

Table 5: Participant’s opinion about code complexity

Choice of names	Strongly disagree	Somewhat disagree	Neutral	Somewhat agree	Strongly agree
Feedback is accurate		1	3	1	1
Feedback is appropriate				6	
Feedback is easy to understand		1		5	
Feedback is helpful to learning			1	5	

Table 6: Participant's opinion about choice of names

Static efficiency	Strongly disagree	Somewhat disagree	Neutral	Somewhat agree	Strongly agree
Feedback is accurate			1	4	1
Feedback is appropriate			3	2	1
Feedback is easy to understand			2	3	1
Feedback is helpful to learning			2	2	2

Table 7: Participant's opinion about Static efficiency

Documentation	Strongly disagree	Somewhat disagree	Neutral	Somewhat agree	Strongly agree
Feedback is accurate				3	3
Feedback is appropriate				3	4
Feedback is easy to understand				2	4
Feedback is helpful to learning		1		3	2

Table 8: Participant's opinion about documentation

In terms of indentation, **Table 4** indicates that all respondents agree that AFSA feedback is good in relation to accuracy, appropriateness, easy of understanding, and contribution to learning. Three of respondents somewhat agree with the remaining part strongly agree. It is been noticed that a participant was Neutral toward this statement "Feedback is easy to understand". He mentions that he prefers a different way of providing feedback about indentation. So instead of giving just the number of correct indentation size add a message to the number of indentation.

In terms of code complexity, the AFSA use McCabe's Cyclomatic Complexity approach to measure flow complexity. The tool checks the complexity of every method and provides feedback to the programmers, see the complexity feedback in Figure 14. Academic staff evaluated the tool feedback. Half of them were neutral toward the

appropriateness of tool feedback. Some of them claimed that the provided feedback does not tell the student the way that the tool measures the complexity **Table 5**.

In terms of choice of names, there was a more united attitude towards AFSA feedback for name choices. The majority of participants somewhat agree about accuracy, appropriateness, easy of understanding, and contribution to learning of tool feedback. But some of them were not confident about the accuracy of using an embedded dictionary in order to check the meaning of identifiers **Table 6**.

Table 7 shows that respondents have different opinions toward tool feedback in terms of static efficiency. The tool uses a tutor sample to check the efficiency of student code. Some participants argued that there are different ways of solving a certain exercise. So, it is difficult to judge the code efficiency of student against one tutor sample. Consequently, some of them were neutral toward some aspects of the feedback.

Table 8 shows that all the majority of participants were satisfied about the tool feedback for indentation.

Additional Information

Among other parameters that participants suggest to be included in the automated feedback tool are: indentation aspect such as use new-line use/spacing (66%), also the style of the braces position of braces/code structure (84%), in addition, a participant suggests using minimum space between methods, duplication of code, check 'CamleCase' for classes and functions/methods. One participant made an additional comment and suggested highlighting in-line comments.

Results of the survey clearly indicate the predominant majority of programming teaching staff is already familiar with or used automated assessment tools to provide valuable insights on students' code but most of them are not familiar with automated style assessment. In general, the study shows that the majority participants consider the tool feedback is helpful to learning. On the other hand, collecting feedback from participants through the questionnaire helped to reveal disadvantages of tool feedback and consider missing assessment factors that need to be included. In addition, the result reflects some strong aspect of the tool feedback. Regarding the evaluation of feedback, all respondents recognize that feedback is important for programmers.

CHAPTER 6

6 Conclusion and Future work

Thesis research developed a prototype tool that automatically assesses programming style for student programmers. The factors that been use to assess the code style are indentation, name choice, complexity, efficiency and documentation. These factors have been chosen to increase the readability of programs. The tools provide feedback for several common programming languages, such as Java, C++, C# and C. These programming languages have different syntax so the tool uses different algorithms to deal with differences such as reserved words. In addition, this thesis conducted a study to evaluate the tool's feedback. The targeted participants were academic staff who had experience in teaching programming. The result of the study showed that the majority of participants considered the tool feedback was helpful to learning.

Although much work was done on assessing code style, given the time constraints, it wasn't possible to finish all aspects in full detail. In particular it was not possible to consider all the 'on the edge cases', so the focus was on the most common aspects of this research. Here are some examples of what could further be explored:

- **Naming issue**: The variables with a high scope (e.g. class-level) should have a longer length of characters. Also the process of extracting variables could be refined more so it does not rely on checking a list of reserved words.

- **Efficiency**: Instead of comparing the student code efficiency against tutor code efficiency, other factors could be considered towards measuring code efficiency.
- **Consistency:** There are many acceptable styles for the position of braces, but some programmers use different styles in the same program, which is not desirable. Also, it is important to have a consistent number of blank lines between methods.
- **Other Languages**: In this project, the focus has been on the languages of C, C#, C++ and Java. In future, the work could be extended to include other languages such as Python, PHP, and more.
- **Graphical User Interface**: It would be useful to provide feedback using a GUI, so it is easier and more convenient for the programmer to retrieve errors.

Appendix

Academic questionnaire

Providing Automated Style Feedback for Student Programmers

Survey & Feedback

Use of Automated Feedback

1. **How much recent experience (last 2 years) do you have teaching programming?**

	None	A little	Moderate	Extensive
Introductory students	⬭	⬭	⬭	⬭
Intermediate students	⬭	⬭	⬭	⬭
Advanced students	⬭	⬭	⬭	⬭

2. **How much medium-term experience (last 5 years) do you have teaching programming?**

	None	A little	Moderate	Extensive
Introductory students	⬭	⬭	⬭	⬭
Intermediate students	⬭	⬭	⬭	⬭
Advanced students	⬭	⬭	⬭	⬭

3. **How much long-term experience (last 10 years) do you have teaching programming?**

	None	A little	Moderate	Extensive
Introductory students	⬭	⬭	⬭	⬭
Intermediate students	⬭	⬭	⬭	⬭
Advanced students	⬭	⬭	⬭	⬭

4. **For each of these automatic programming assessment tools, indicate which you are familiar with, which you have previously used in your teaching, and which you intend to use in future.**

	Familiar	Past use	Future use
“try” program (command-line function testing)	⬭	⬭	⬭
CodeRunner (quiz-based function testing in FLO)	⬭	⬭	⬭
Other function testing (specify …..……………………..)	⬭	⬭	⬭

Checkstyle: open-source style checking	⬭	⬭	⬭
Other style checking (specify ……………………….)	⬭	⬭	⬭

5. **To what extent do you think an automated tool such as this can replace manual assessment of code style?**

	Not at all	Partially	Substantially	Completely
Introductory students	⬭	⬭	⬭	⬭
Intermediate students	⬭	⬭	⬭	⬭
Advanced students	⬭	⬭	⬭	⬭

Evaluation of Feedback

The following questions relate to different aspects of style on which the tool provides feedback. For each aspect, indicate your opinion of the tool's feedback in relation to accuracy, appropriateness, easy of understanding, and contribution to learning.

6. **Indentation**

	Strongly disagree	Somewhat disagree	Neutral	Somewhat agree	Strongly agree
Feedback is accurate	⬭	⬭	⬭	⬭	⬭
Feedback is appropriate	⬭	⬭	⬭	⬭	⬭
Feedback is easy to understand	⬭	⬭	⬭	⬭	⬭
Feedback is helpful to learning	⬭	⬭	⬭	⬭	⬭

7. **Code complexity**

	Strongly disagree	Somewhat disagree	Neutral	Somewhat agree	Strongly agree
Feedback is accurate	⬭	⬭	⬭	⬭	⬭
Feedback is appropriate	⬭	⬭	⬭	⬭	⬭
Feedback is easy to understand	⬭	⬭	⬭	⬭	⬭
Feedback is helpful to learning	⬭	⬭	⬭	⬭	⬭

8. **Choice of names**

	Strongly disagree	Somewhat disagree	Neutral	Somewhat agree	Strongly agree
Feedback is accurate	⬭	⬭	⬭	⬭	⬭
Feedback is appropriate	⬭	⬭	⬭	⬭	⬭
Feedback is easy to understand	⬭	⬭	⬭	⬭	⬭
Feedback is helpful to learning	⬭	⬭	⬭	⬭	⬭

9. **Static efficiency**

	Strongly disagree	Somewhat disagree	Neutral	Somewhat agree	Strongly agree
Feedback is accurate	⬭	⬭	⬭	⬭	⬭
Feedback is appropriate	⬭	⬭	⬭	⬭	⬭
Feedback is easy to understand	⬭	⬭	⬭	⬭	⬭
Feedback is helpful to learning	⬭	⬭	⬭	⬭	⬭

10. **Documentation**

	Strongly disagree	Somewhat disagree	Neutral	Somewhat agree	Strongly agree
Feedback is accurate	⬭	⬭	⬭	⬭	⬭
Feedback is appropriate	⬭	⬭	⬭	⬭	⬭
Feedback is easy to understand	⬭	⬭	⬭	⬭	⬭
Feedback is helpful to learning	⬭	⬭	⬭	⬭	⬭

Additional Information

11. **The tool provides feedback about indentation, complexity, choice of names, efficiency and documentation. What other aspects of coding style should be included in an automated feedback tool?**

……………………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………

12. **Do you have any additional comments or suggestions about the tool?**

……………………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………

References

Ala-Mutka, K. M. (2005). "A survey of automated assessment approaches for programming assignments." Computer science education **15**(2): 83-102.

Ala-Mutka, K., T. Uimonen and H.-M. Jarvinen (2004). "Supporting students in C++ programming courses with automatic program style assessment." Journal of Information Technology Education: Research **3**(1): 245-262.

Android. Code Style for Contributors. Retrieved 1 Nov, 2016, from https://source.android.com/source/code-style.html#java-language-rules

Auffarth, B., M. López-Sánchez, J. Campos i Miralles and A. Puig (2008). System for automated assistance in correction of programming exercises (SAC). International Congress University Teaching and Innovation (CIDUI).

Benford, S. D., E. K. Burke, E. Foxley and C. A. Higgins (1995). The Ceilidh system for the automatic grading of students on programming courses. Proceedings of the 33rd annual on Southeast regional conference. Clemson, South Carolina, ACM**:** 176-182.

Burn, O. (2003). " Checkstyle." SourceForge. net. Posted at http://checkstyle. sourceforge. net/(accessed October 16, 2003).

Carmichael, R. M. (2002). "Measures of efficiency and effectiveness as indicators of quality–A systems approach." Journal of Institutional Research Southeast Asia (JIRSEA) 1(1): 3-14.

Chen, W., X. Li and W. Liu (2011). Teaching computer programming to non-computer science students. Proceedings of the 3rd Asian Conference on Education (ACE). Katahira: IAFOR Publications.

Deek, F. P. and J. A. McHugh (1998). "A survey and critical analysis of tools for learning programming." Computer science education **8**(2): 130-178.

Douce, C., D. Livingstone and J. Orwell (2005). "Automatic test-based assessment of programming: A review." Journal on Educational Resources in Computing (JERIC) **5**(3): 4.

Drupal. (2016). CSS formatting guidelines. Retrieved 8 Nov, 2016, from https://www.drupal.org/docs/develop/standards/css/css-formatting-guidelines

Enström, E., G. Kreitz, F. Niemelä, P. Söderman and V. Kann (2011). Five years with kattis—Using an automated assessment system in teaching. Frontiers in Education Conference (FIE), 2011, IEEE.

Guido van Rossum, B. W., Nick Coghlan (2001). "https://www.python.org/dev/peps/pep-0008/." Python.org.

Gupta, S. and S. K. Dubey (2012). "Automatic assessment of programming assignment." Computer Science & Engineering **2**(1): 67.

Haley, D. T., P. Thomas, A. De Roeck and M. Petre (2007). "Seeing the whole picture: evaluating automated assessment systems." Innovation in Teaching and Learning in Information and Computer Sciences **6**(4): 203-224.

Higgins, C. A., G. Gray, P. Symeonidis and A. Tsintsifas (2005). "Automated assessment and experiences of teaching programming." Journal on Educational Resources in Computing (JERIC) **5**(3): 5.

Idzelis, M. (2003). "Jazzy." iEdit programmer's Text Editor.

Joy, M., N. Griffiths and R. Boyatt (2005). "The boss online submission and assessment system." Journal on Educational Resources in Computing (JERIC) **5**(3): 2.

Koyya, P., Y. Lee and J. Yang (2013). "Feedback for Programming Assignments Using Software-Metrics and Reference Code." ISRN Software Engineering **2013**.

McCabe, T. J. (1976). "A complexity measure." IEEE Transactions on software Engineering(4): 308-320.

Oracle. (1999). White space Retrieved 8 Nov, 2016, from http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-141388.html#475

Pettit, R., J. Homer, K. Holcomb, N. Simone and S. Mengel (2015). Are automated assessment tools helpful in programming courses. 2015 ASEE Annual Conference and Exposition.

Pieterse, V. (2013). Automated assessment of programming assignments. Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research, Open Universiteit, Heerlen.

Rahman, K. A. and M. J. Nordin (2007). A review on the static analysis approach in the automated programming assessment systems. Proceedings of the national conference on programming.

Redish, K. and W. Smyth (1986). "Program style analysis: A natural by-product of program compilation." Communications of the ACM **29**(2): 126-133.

Rees, M. J. (1982). "Automatic assessment aids for Pascal programs." ACM Sigplan Notices **17**(10): 33-42.

Ribeiro, P. and P. Guerreiro (2009). "Improving the automatic evaluation of problem solutions in programming contests." Olympiads in Informatics **3**: 132-143.

Salleh, S. M., Z. Shukur and H. M. Judi (2013). "Analysis of Research in Programming Teaching Tools: An Initial Review." Procedia-Social and Behavioral Sciences **103**: 127-135.

Swartz, F. (2007). Java: Computing Cyclomatic Complexity. Retrieved 10 Oct, 2016, from http://www.leepoint.net/principles_and_practices/complexity/complexity-java-method.html

Truong, N. (2007). A web-based programming environment for novice programmers, Queensland University of Technology. **PhD**.

Truong, N., P. Roe and P. Bancroft (2005). Automated feedback for "fill in the gap" programming exercises. Proceedings of the 7th Australasian conference on Computing education - Volume 42. Newcastle, New South Wales, Australia, Australian Computer Society, Inc.**:** 117-126.

Vihavainen, A., T. Vikberg, M. Luukkainen and M. Pärtel (2013). Scaffolding students' learning using test my code. Proceedings of the 18th ACM conference on Innovation and technology in computer science education, ACM.