Increasing students' code quality knowledge using automated formative feedback based on software metrics.

van den Aker, E

Award date:
2022

Link to publication

INCREASING STUDENTS' CODE QUALITY KNOWLEDGE USING AUTOMATED FORMATIVE FEEDBACK BASED ON SOFTWARE METRICS

by

Eddy van den Aker

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering
to be defended publicly on Monday June 20, 2022 at 13:00.

Student number:		
Course code:	IM9906	
Thesis committee:	dr. Ebrahim Rahimi (chairman and first supervisor),	Open University
	dr. ir. Fenia Aivaloglou (second supervisor),	Open University

Contents

List of Figures

LIST OF TABLES

Summary

Software maintenance is an important aspect of the development life-cycle. One of the critical success factors of software maintainability is the quality of the code. It is therefore important that students learn how to write high quality code. However, previous research by Keuning et al. [37] shows that student code often contains significant code quality issues that would lead to problems with maintainability. Additionally, these issues are rarely fixed.

To improve the quality of their code, students need high quality formative feedback. While much research has been done in the context of formative feedback in programming education. However, most of this research focuses on beginners (e.g. first year students). In this context, assignments are generally small and model solutions are often available. This study aims to provide automated formative feedback on code quality for open-ended assignments. These types of assignments are more typical for students further in their education.

In this study we have used software metrics as a basis for generating the feedback. To do this, a set of suitable software metrics have been selected. This has been done based on a literature review. Three suitable metrics were found, Cyclomatic Complexity (CC) [44], Lines of Code (LoC), and code duplication. Additionally, we have defined a number of design principles for automated formative feedback on code quality. These design principles are based on two focus group sessions involving students and teachers from the HBO-ICT program at Zuyd University of Applied Science, and on a number of general meta design principles found in the literature.

Finally, the design principles for formative feedback on code quality have been implemented in a prototype. This prototype has been used to evaluate the design principles defined in this study. For this evaluation, a number of observations have been done with students from the same program as the focus group sessions. Each observation consisted of a before interview, a programming session, and an after interview. Additionally, an exploration and discussion of the prototype and the design principles has been done with a senior teacher.

From the first evaluations, the design principles defined in this study look promising. All students found the feedback provided by the prototype helpful. However, further refinement and extension of both the prototype and the design principles implemented in said prototype, are needed.

SAMENVATTING

Softwareonderhoud is een belangrijk onderdeel van de development life-cycle. Een van de kritische succesfactoren die software onderhoudbaar maakt is de kwaliteit van de code. Het is daarom van belang dat studenten leren om hoge kwaliteit code the schrijven. Echter, uit onderzoek van Keuning et al. blijkt dat code van studenten vaak significante kwaliteitsproblemen bevat die de onderhoudbaarheid van software negatief zouden beïnvloeden. Daar komt bij dat deze kwaliteitsproblemen zelden opgelost worden.

Studenten hebben goede formatieve feedback nodig om de kwaliteit van hun code te verbeteren. Hoewel veel onderzoek is gedaan in de context van formatieve feedback met betrekking tot het programmeeronderwijs, is veel van dit onderzoek gericht op beginners (o.a. studenten in hun eerste leerjaar). In deze context zijn opdrachten vaak kleinschalig en zijn meestal uitwerkingen beschikbaar. Dit onderzoek heeft als doel om automatisch feedback te genereren op code kwaliteit in de context van open opdrachten. Dit soort opdrachten zijn gebruikelijk voor studenten die verder gevorderd zijn in hun opleiding.

In dit onderzoek is gebruik gemaakt van softwaremetrieken als basis voor het genereren van de feedback. Hiervoor is eerst een selectie gemaakt van geschikte softwaremetrieken. Dit is gedaan aan de hand van een literatuuronderzoek. Uit dit onderzoek zijn drie geschikte softwaremetrieken naar voren gekomen, Cyclomatische Complexiteit (CC), regels code (Lines of Code, ofwel LoC) en code duplicatie. Naast de softwaremetrieken zijn een aantal design principles gedefinieerd voor het automatisch genereren van formatieve feedback op code kwaliteit. Deze design principles zijn gedefinieerd op basis van twee focusgroepsessies met studenten en docenten van de HBO-ICT opleiding van Zuyd Hogeschool en een aantal generieke meta design principles uit de literatuur.

De design principles voor formatieve op code kwaliteit zijn geïmplementeerd in een prototype. Dit prototype is gebruikt om de design principles te evalueren. Voor deze evaluatie zijn een aantal observaties uitgevoerd met studenten van dezelfde opleiding als de focusgroepsessies. Iedere observatie bestaat uit een interview vooraf, een programmeersessie en een interview achteraf. Tevens een het prototype bekeken door een senior docent en is een discussie gevoerd of de gedefinieerde design principles.

De eerste resultaten met betrekking tot de effectiviteit van de in dit onderzoek opgestelde design principles op basis van de evaluatie zijn veelbelovend. Alle studenten vonden de feedback gegeven door het prototype nuttig. Echter is er ook ruimte voor verdere verfijning en uitbreiding van zowel het prototype alsook de design principles die in dat prototype verwerkt zijn.

1 INTRODUCTION

Software maintenance is becoming more important as reliance on software increases [7]. Additionally, with trends such as Agile development, a larger part of development work takes place in the maintenance phase [58]. A critical factor for the maintainability of software is the quality of the code [67]. It is therefore important that students, who will become future developers, learn how to write high quality, maintainable code. However, research shows that students' code contains significant quality issues that would impact maintainability [37]. Additionally, these issues are often not fixed. Overall, code quality should receive more attention in programming education.

For students to write more maintainable code, they first need to learn what high quality code is. For this, students need detailed formative feedback on the code they write. Providing this feedback takes a lot of time, which teachers often do not have. Automated generation of this feedback can be a solution. However, much of the research in this area focuses on novice students in the first year of their education. As a result, most approaches for feedback on code assume small assignments, often with available model solutions. These approaches are not suited for more advanced programming education, where assignments are often larger, more open-ended (i.e. no model solutions are available), and project based.

The aim of this research is to find an approach for automatically generating formative feedback on code quality for advanced programming students in undergraduate education. The goal of this feedback is to increase the knowledge of code quality among the students and empower them to write more maintainable code. To find an approach for generating formative feedback on code quality, we have defined the following research question: *How can software metrics be used to facilitate formative assessment on code quality in project-based, open-ended programming assignments?*

To answer the research question mentioned in the previous paragraph, a design based research approach has been used. During this research, a number of software metrics have been extracted from the literature. From these metrics, a number of suiteable metrics for generating formative feedback on code quality have been selected based on a number of criteria. The selected software metrics are combined with general design principles from the literature and with input from two focus group sessions to define a number of design principles specific to automated feedback on code quality. These design principles have been implemented in a prototype. This prototype has been used to test the design principles in a number of evaluations. For these evaluations, a qualitative approach has been used.

The contributions that this research aims to make are both theoretical and practical. The theoretical contribution of this research is a set of design principles for software metric based formative feedback on code quality. These design principles translate the more general design principles for feedback and learning dashboards into design principles specific to code quality and programming education. The design principles defined in this research can be used to guide the design of (software) tools that provide formative feedback on code quality. The practical contribution is a prototype based on the design principles that provides code quality feedback to students. In addition to the prototype itself, the design (including the design principles) of the prototype could also inform the design of other tools.

This thesis contains nine chapters, this chapter, the introduction, being the first. Chapter 2 gives an overview of the literature. Chapter 3 contains the problem analysis, which looks at why this research is relevant from both a theoretical and a practical point of view. Chapter 4 describes the research design as well as the setting in which the research hase been done. Additionally, it looks at a number of ethical considerations, since this research involves participants. Chapter 5 is a literature review with the goal of finding suitable metrics for automatically generating formative feedback on code quality. Chapter 6 looks at how the metrics data can be used to generate high quality feedback that is useful to students. The goal of this chapter is a set of initial design principles. The design principles defined in Chapter 6 are then implemented into a prototype and evaluated. This is described in Chapter 7. Chapter 8 discusses the results of the research, including the selected metrics, the outcomes of the focus group sessions, the design principles for formative feedback on code quality, the prototype, and the evaluation of the design principles. This chapter also discusses the limitations of the research and the threats to validity. Finally, Chapter 9 gives a conclusion of the research and gives suggestions for future research.

2
Related Work

This chapter gives an overview of the work related to formative code quality feedback on source code. Section 2.2 gives an overview of software maintenance and the importance of high quality code. Section 2.3 reviews the literature on programming education. Section 2.4 looks at feedback and assessment, the different types of feedback, the characteristics of good feedback, and finally the role of feedback in programming education. Section 2.5 gives an overview of learning dashboards. Finally, Section 2.6 summarizes and reflects on the related work.

2.1. Software Quality

Software quality is defined in ISO/IEC 25010:2011. This standard describes a number of attributes for software quality. These are categorized as: functional stability; performance efficiency; compatibility; usability; reliability; security; maintainability; portability. The previous version of this standard is ISO/IEC 9126-1:2001. In literature before 2011, this previous standard is used instead of the current standard.

The quality aspects of ISO/IEC 25010:2011 can be divided into internal quality aspects (maintainability) and external quality aspects (functional stability, performance efficiency, compatibility, usability, reliability, security, and portability). Internal quality describes the quality as measured from the viewpoint of the construction of the software. External quality is the quality as perceived from the perspective of the users. Internal quality is one of the critical factors in the successful maintenance of software [67]. The internal quality of the software also affects the speed at which changes can be made [9]. The maintainability aspect considers a number of sub-aspects: analysability; changeability; testability; stability.

2.2. Software Maintenance

Software maintenance is an important aspect of the software development life-cycle. Between 40% and 80% (60% average) of the costs of software are maintenance costs [7][27]. Additionally, it is important for organisations to be able to respond to changes (e.g. respond quickly to changes in the needs of users of a software system). If it is not possible to adapt software to these changes, it could lead to lost opportunities [7].

IEEE Std. 14764-2006 defines software maintenance as[1]:

> "*the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage*" ([34], p. 4)

While the current IEEE standard does include some pre-delivery activities, the main focus of software maintenance is on the post-delivery activities, when the system is already in production use.

The work done during software maintenance can be categorized into four different categories [7][41]. The first category is adaptive maintenance. This occurs when the environment of the software changes and the software needs to be adapted to this environment (e.g. when upgrading the operating system). Perfective maintenance occurs when new features need to be added. These new features generally come from user requests. The third category is corrective maintenance. This type of maintenance involves fixing bugs and errors. The final category is preventive maintenance, which aims to prevent future issues (e.g. refactoring a module to reduce complexity).

Bennett and Rajlich [6] defined a software life cycle model which describes the phases a software system goes through during its lifetime (see Figure 2.1). Their main goal was to show that maintenance is not a single phase, but consists of multiple distinct phases. All phases after initial development can be considered maintenance.

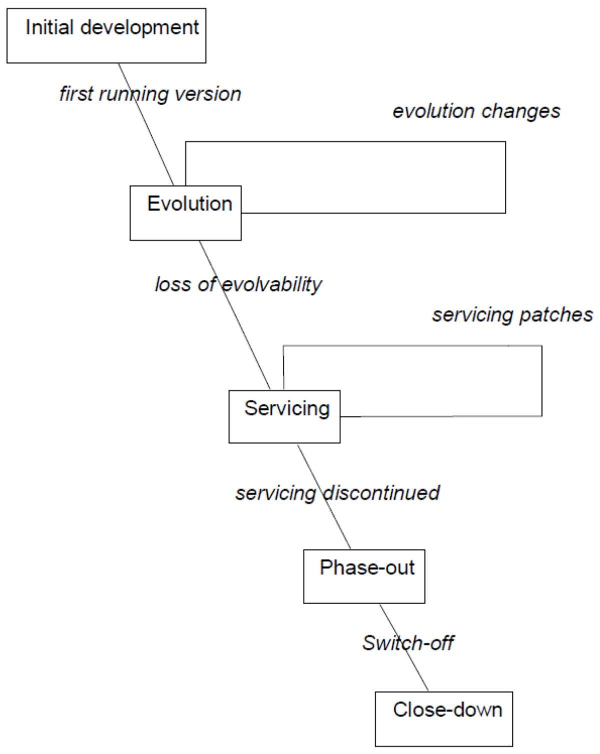


Figure 2.1: Staged software life cycle model [6]

The evolution phase is the phase directly after initial development (if initial development was successful) [6]. During this phase, new features can be added easily, as the architecture is still intact and the team has enough knowledge of the software system. However,

[1]Pre-delivery in this context mainly includes activities that aim to support the software post-delivery

without careful management, the structure of the software system will decay. The decay often starts with losing knowledge of the system (e.g. by staff turnover). When it is no longer possible to evolve the system (i.e. add significant new features), it enters the servicing phase. During this phase, bugs and issues are still fixed, but it is no longer possible to make major changes. Finally, even servicing becomes too expensive and the system goes into the phase-out phase followed by the close-down phase. While not formally confirmed, Bennett and Rajlich suspect that going back to a previous phase (e.g. from the servicing phase back to the evolution phase) is practically impossible.

The evolution stage has gained more importance over the years with the rise of iterative development processes such as Agile [58]. In a process called Evolutionary Software Development (ESD), the initial development stage is short and the bulk of the development work is done in the evolution stage.

Software Metrics

It is important in software engineering in general and for software maintainability specifically to have good, empirical knowledge of the state of a system. It is, for example, important for maintainability to know whether a system is getting more complex over time, and which modules are adding this complexity [67]. This is where software metrics come in.

According to Fenton and Bieman [24] measurement can be defined as:

> "*the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way so as to describe them according to clearly defined rules.* " ([24], p. 5)

Software metrics are this process of measurement applied to some (or multiple) aspect(s) of software engineering [24]. Software metrics can be extracted from three sources related to the software [32], from the software itself, from the software development process, or from supporting resources used during the development. Some examples of software metrics are lines of code, code duplication, and cyclomatic complexity.

Cyclomatic Complexity (CC) as defined by McCabe [44] is possibly one of the more well known software metrics. The goal of CC is to give a quantifiable way to describe the complexity of some part of code. CC does this by looking at the control structure of code as a graph and determining the number of unique paths through that graph. The "Hello, World!" example shown in listing 2.1 only has a single path through the code, so it has a CC of one. The example shown in listing 2.2 has a CC of two, since there are two paths through the code. One where the if statement is true, and another path where the if statement is false. There are many other examples of software metrics used to measure different aspects of a system (e.g. lines of code, coupling, inheritance) and the processes surrounding it (e.g. number of bug reports).

Listing 2.1: cyclomatic complexity example 1

```c
#include <stdio.h>

int main() {
  printf("Hello, World!");
  return 0;
}
```

Listing 2.2: cyclomatic complexity example 2

```
#include <stdio.h>

int main() {
    int num = 5;
    if (num % 2 == 0) {
        printf("num is even.");
    }
    return 0;
}
```

There are various software metrics that are used to measure how maintainable a software system is. Different models have been proposed for this purpose. One is the maintainability index by Oman and Hagemeister [52] and by Coleman et al. [16]. A more recent model is the SIG model proposed by Heitlager et al. [30].

The maintainability index uses a polynomal equation which uses several software metrics as input (such as lines of code) [52]. This equation leads to an index which indicates the overall maintainability of a system. The numbers used in the equation have been calibrated by comparing the outcomes with those of experts.

While the maintainability index does give an indication of how maintainable a system is, it has several issues. The first issue is that it only gives an overall score, which does not help identify why a system is easy or hard to maintain. Additionally, since it includes numbers which have been calibrated by comparing the results with experts, it is not always transparent how each metric influences the score. These critiques were one of the reasons for Heitlager et al. [30] to create a new model for maintainability that solves these issues.

The main idea for the SIG maintainability model is the mapping of software code properties onto sub-characteristics of maintainability from the ISO 9126 standard [30] (see Figure 2.2). Heitlager et al. defined a number of important requirements when developing their model [30]:

- Measures should work for different kinds of technologies, languages, and architectures.

- Measures should be well defined and easy to compute.

- Measures should be easy to understand and explain (including to non-technical staff) and should aid communication.

- Measures should enable root-cause analysis (they should provide basis for action).

For each of the source code properties Heitlager et al. [30] give a number of possible metrics. For volume they suggest lines of code and man years via backfiring function points. Lines of code can also be used to determine the unit size. Additionally, they suggest a number of secondary metrics that can give rough estimates, such as number of tables in the database, or number of screens, etc. Heitlager et al. suggest cyclomatic complexity as a measure for complexity per unit. Duplication can be measured by looking at blocks of at least six lines of code, then calculating the percentage of these blocks that occur more than once. Finally, the unit testing property can be determined by test coverage.

		source code properties				
		volume	complexity per unit	duplication	unit size	unit testing
ISO 9126 maintainability	analysability	x		x	x	x
	changeability		x	x		
	stability					x
	testability		x		x	x

Figure 2.2: ISO 9126 maintainability characteristics mapped to source code properties [30]

2.3. Programming education and software maintainability

While programming skills (e.g. writing high quality code, correctly modelling real life problems, etc.) are becoming more important, these skills are generally considered difficult to learn [60]. While this is not directly seen in failure rates in introductory programming courses [4][5][77] (although sources on failure/drop-out rates in programming courses rely on self-reporting, which might bias the results), computer science and programming courses are still perceived as difficult [60]. It should be noted that much of the research on programming education focuses on novice programming students, often in introductory courses (e.g. CS1).

As noted in Section 2.2, software maintainability is an important part of software development and engineering. As discussed, one of the key factors of maintainability is the quality of the source code. It is therefore important that students learn how to write maintainable, high-quality software. In an analysis of students' code, Keuning et al. [37] found that it contained significant issues that would make software less maintainable. Additionally, they show that students rarely fix the issues in their code, possibly because they have moved on to the next assignment.

A survey by Lahtinen et al. [40] looked at the difficulties faced by students in novice programming courses. Among these difficulties, aspects related to the structure of programs (such as designing a program to solve a certain task and dividing functionality into procedures) ranked high among both student and teacher respondents. A reproduction of the same survey by Piteira and Costa [56] found similar results. These are the type of difficulties that also influence the issues found by Keuning et al. [37].

Some efforts have been made to help students write better code. For example Stegeman et al. [68] designed a rubric that can be used to assess the quality of students' code. However, this approach requires teachers to manually review students' code. Keuning et al. [37] noted that automated code style and code quality issue detection tools (Checkstyle[2], PMD[3], and PatternCoder[4]) used by students in their data set do not significantly improve

[2]https://checkstyle.sourceforge.io/
[3]https://pmd.sourceforge.io/pmd-5.5.2/
[4]http://www.patterncoder.org/

the quality of the code. Keuning et al. suggest that better tools need to be built to resolve students' code quality issues.

Overall, more attention to code quality is needed in programming education[11][12][37][38]. Building better code quality feedback tools might be one possible solution. Software metrics (see Section 2.2) might be useful as an empirical method to help students understand code quality and maintainability. A large advantage of software metrics is that the extraction of these metrics from the code could be automated.

2.4. Assessment and Feedback

Feedback is one of the most influential parts of the learning process [28]. In an analysis of 12 previous meta-analysis, Hattie and Timperley found an average effect size[5] of 0.79 for feedback (the average effect of schooling in general is 0.40). The idea that feedback is a powerful tool is again confirmed in a later review by Winsiewski et al. [79]. However, both Hattie and Timperley [28] and Wisniewski et al. [79] noted that the feedback given and the way in which it is given are important. Hattie and Timperley define feedback as:

> "*Information provided by an agent (e.g. teacher, peer, book, parent, self, experience) regarding aspects of one's performance or understanding.* "
> ([28], p. 81)

Hattie and Timperley [28] also present a framework for feedback (see Figure 2.3). It states that the main purpose of feedback is to reduce the gap between current understanding or performance and the desired goal. Teachers and students can employ strategies to achieve this purpose. These strategies can vary in effectiveness.

According to Hattie and Timperley [28], effective feedback needs to answer three questions, "Where am I going?", "How am I going?", and "Where to next". The first question is about the learning goals. The second question is about how the student is progressing towards those goals. Finally, the third question answers how the student can make better progress. Hattie and Timperley call the answers to these three questions feed up, feed back, and feed forward.

Finally, feedback exists at four different levels [28]. First is the task level, which is feedback on a specific task (e.g. giving correct and incorrect answers). The process level gives feedback on how the student performs a task. The third level is the self-regulation level. This helps the student regulate their own actions for achieving their learning goals. Finally, the self level is about the student (e.g. praising a student for being a "good learner"). It should be noted that Hattie and Timperley included the self level because of its prevalence in classrooms, not because it is effective.

2.4.1. Summative Assessment and Formative Assessment

There are two main types of assessment and feedback. The first is summative assessment, also called assessment of learning. This type of assessment aims to judge the current progress of the student, possibly compared to some desired goal [70]. This type of assessment is mainly concerned with gathering evidence to judge the learning of a student. This

[5]The magnitude of the effect of an experiment/intervention (in this case, feedback) compared to a control group

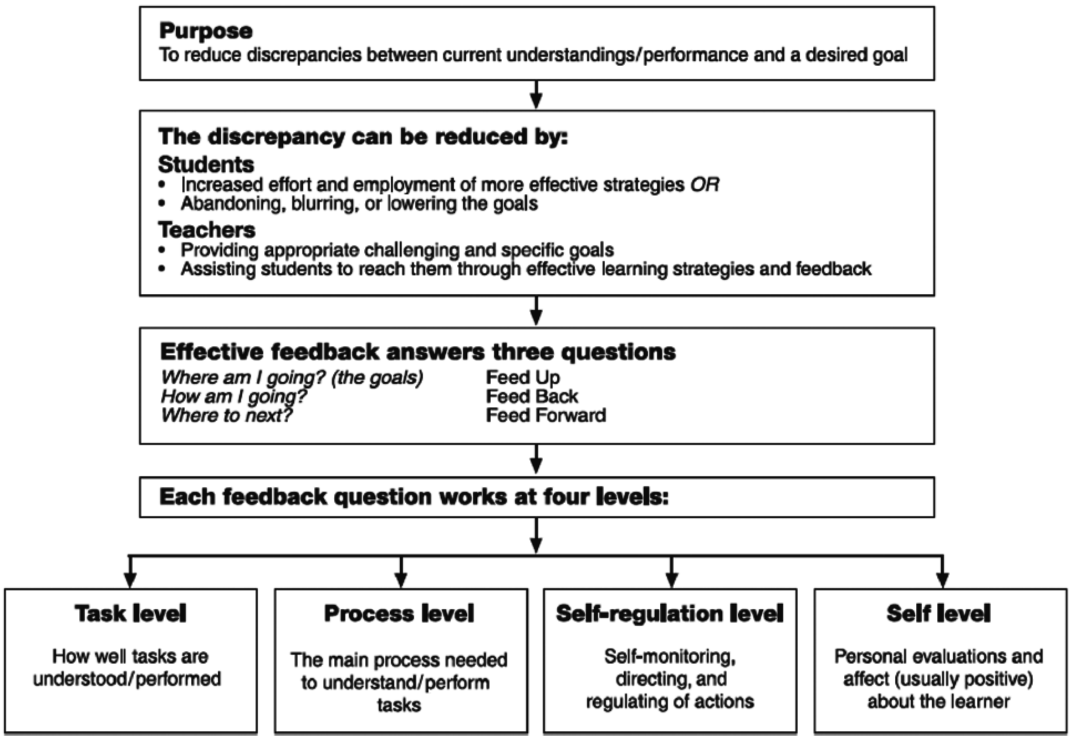

Purpose
To reduce discrepancies between current understandings/performance and a desired goal
The discrepancy can be reduced by:
Students
• Increased effort and employment of more effective strategies OR
• Abandoning, blurring, or lowering the goals
Teachers
• Providing appropriate challenging and specific goals
• Assisting students to reach them through effective learning strategies and feedback
Effective feedback answers three questions
Where am I going? (the goals) Feed Up
How am I going? Feed Back
Where to next? Feed Forward
Each feedback question works at four levels:
Task level
How well tasks are understood/performed
Process level
The main process needed to understand/perform tasks
Self-regulation level
Self-monitoring, directing, and regulating of actions
Self level
Personal evaluations and affect (usually positive) about the learner

Figure 2.3: Framework for feedback proposed by Hattie and Timperley [28]

often results in a grade. Summative assessments are often in the form of final projects, papers or examinations [20].

The second type of assessment is formative assessment, also called assessment for learning. Formative assessment not only aims to check the current progress of a student, but also to provide feedback on the gap between the students' current progress and the goal, how a student may reach their goal and how to make better progress toward that goal [70].

According to McMillan, the process of formative assessment occurs in a number of repeating steps [45]. It starts with gathering evidence of learning. This step seeks to measure the students' current understanding. This evidence will be evaluated to find gaps between the students' understanding and the desired goal. Feedback that aims to support understanding is given. Finally, the student and instructors can then use this feedback to make adjustments. After the adjustments have been made, the cycle starts again with gathering evidence of learning (see Figure 2.4).

2.4.2. Computer Assisted Feedback

According to Hattie and Timperley [28] computer assisted feedback is effective. In a comparative case study by Denton et al. [19] computer assisted feedback was rated higher by students than traditional written feedback. Important factors in this finding are clear structure of the feedback, legibility, information on gaps in student understanding, and identification of good aspects of student submissions. Additionally, computer assisted feedback was provided to the students faster than traditional feedback [19][31]. Another study by Faber et al. showed similar results, with students reporting higher motivation and performing better [23].

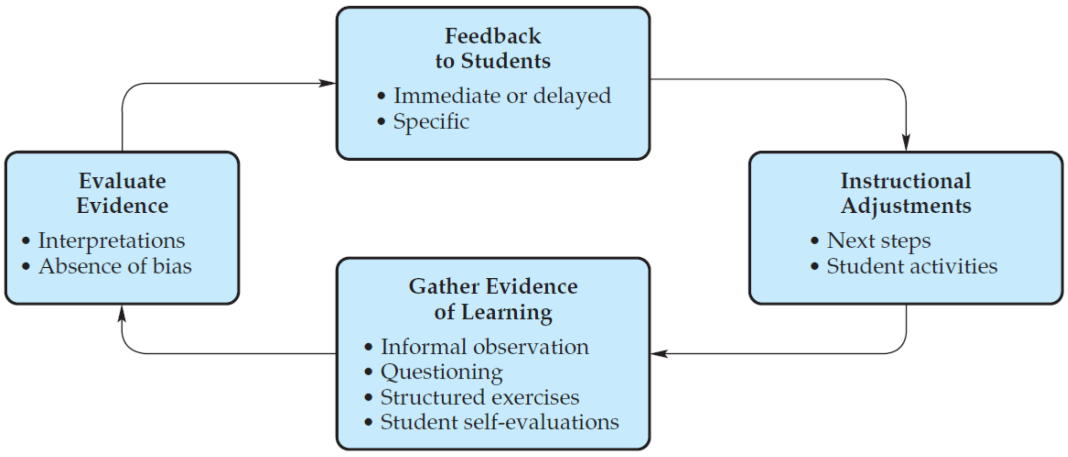


Figure 2.4: Formative assessment cycle [45]

2.4.3. Design Principles for Formative Feedback

One of the important findings of both Hattie and Timperley [28] and Wisniewski et al. [79] is that not all feedback is equally effective. To guide the process of giving effective feedback, Nicol and Macfarlane-Dick [51] have defined a number of principles for good formative feedback. A summary of the design principles can be found in appendix A.

FB1: Good feedback helps clarify what good performance is The first question in the feedback model defined by Hattie and Timperley is "Where am I going?" [28]. Good feedback should help answer this question. Feedback should guide students toward their learning goals by clarifying the criteria of these learning goals [51][53][57]. Similarly, techniques such as goal setting [43], which make (learning) goals specific and measurable, increase student achievement [21][49]. An example of a clear goal would be to have an average cyclomatic complexity of 10 or less.

FB2: Good feedback facilitates the development of self-assessment (reflection) in learning Feedback should enhance self-regulated learning. According to Mega et al. [46] self-regulation is one of the important predictors of academic achievement. An important aspect of self-regulated learning is self-assessment and reflection. Feedback should support students in assessing their own work [51]. This can be achieved by teaching students how criteria are applied and helping students use the data to improve performance [53]. For example, feedback should help students judge the quality of their code in future assignments.

Additionally, testing or grading (summative assessment) should not be confused with feedback [28]. Mixing formative feedback with grading can lead to students comparing themselves with other students instead of improving on a task [51]. Grades can also distract students from the formative feedback provided [31]. Brown and Glover [14] recommend letting students receive formative feedback without any summative component before submitting a final version for grading or testing. Additionally, Panadero et al [53] recommend not mixing feedback and grading in self-assessment scenario's, as this can harm the self-assessment (e.g. by incentivizing the student to be less critical of their own work).

FB3: Good feedback delivers high quality information to students about their learning The correct amount of feedback with enough detail should be provided [26][28]. The feedback should match the students' level of knowledge [28][78]. According to Nicol and Macfarlane-Dick, high quality information is:

> "*... information that helps students troubleshoot their own performance and self-correct: that is, it helps students take action to reduce the discrepancy between their intentions and the resulting effects.* " ([51], p. 208)

An example of feedback that might help students troubleshoot their performance for programming assignments might be to not only show parts of code that are incorrect, but also provide steps that the student could use to improve the assignment. In this case, writing incorrect code (or not knowing how to write correct code) is the discrepancy between the goal and the students' current understanding.

The timeliness of feedback is another important factor of high quality feedback [28][51][78]. Generally, faster feedback is better, as long as it does not decrease quality. However, Hattie and Timperley note that sometimes a delay is better. They speculate that this might be the case for more advanced students who are able to self-reflect on an assignment and figure things out for themselves. To increase the timeliness of feedback, it needs to be in regular, short intervals within a course [26].

FB4: Good feedback encourages teacher and peer dialog around learning Feedback should enable teacher and peer dialog [51]. This helps students better understand the feedback, as it encourages students to actively participate in the process instead of only being a receiver. A study by Schaaf et al. [72] found that feedback combined with teacher dialog was perceived as more useful by students than feedback alone.

FB5: Good feedback encourages positive motivational beliefs and self-esteem Positive motivation has a significant positive effect on the learning outcome of students [42]. Additionally, feedback should enhance student self-efficacy [28], which is also linked to student motivation. Feedback should aim to encourage these positive motivational beliefs in order to be effective [51].

FB6: Good feedback provides opportunities to close the gap between current and desired performance While feedback looks at past work of the student, the feedback should be relevant for future tasks [14]. It should allow students to close the gap between their current level and the desired level [51].

FB7: Good feedback provides information to teachers that can be used to help shape teaching The information gathered by providing feedback should be useful in guiding teachers [51]. The information should be used to enhance the teachers' pedagogical content knowledge (PCK). PCK is the combination of content knowledge and pedagogy [64]. This includes elements like knowing useful forms to represent ideas or concepts, good examples and explanations, available demonstrations, and knowing which specific topics are easy or hard for students to grasp [65]. For example, if a lot of students need feedback on the aspect of code coupling, it shows that this subject is hard for students. At the same time, it might also deliver good examples of how to deal with code coupling issues, providing the teacher with examples to use in lessons.

2.4.4. Feedback on Source Code

Various approaches have been used to improve feedback in the context of programming education. Approaches vary in terms of automation and computer assistance, from fully automatic (test cases, software metrics, and rule-based) feedback to fully manual (peer assessment and code annotation). Below, some of these approaches are explained.

Code annotation One approach for providing feedback on programming assignments is annotating the code [1][17][69]. With this approach some kind of annotation scheme is used to place feedback directly in the context of the source code.

Generally, these systems allow teachers or peers to place comments that are linked to the code and visible from within the code. Another approach is allowing teachers and peers to annotate the code with tags, which was used by Cummins et al. [17].

Peer assessment Another approach for generating feedback is peer assessment [18][33][66][76]. With this approach, students following a programming course will review each others work and provide feedback. Most studies provide the students with rubrics or some other criteria on which to base their peer reviews. By giving feedback to peers, students might discover mistakes in their own assignments. This process might make them more critical of their own work.

Test cases A common approach to generating feedback in programming education concerns using test cases, often in combination with testing tools also used for testing software [13][22][63][75]. These systems generally provide some input for the students' submissions, then check the output and compare it to some expected output.

Software metrics Software metrics have been used as a feedback mechanism in programming education. These metrics come from either the source code (e.g. cyclomatic complexity or lines of code)[35][39][47][55] or from the development process (e.g. the Git repository)[48][60][61]. Most research that involves software metric based feedback uses the metrics to inform and guide teachers giving feedback [39], but not giving students direct access to the metrics. This approach aims to assist a teacher in the process of providing feedback. Providing students with direct access to the metrics has been suggested by Pettit et al. [55].

Rules The final approach discussed in this thesis is the usage of rules to generate feedback (e.g "each method should have a comment") [2][73]. This method has some overlap with software metrics. For example, the Submit program [73] has a rule that methods should not be longer than 50 lines. This rule makes use of a software metric (lines of code) to determine the result. However, it is used as a rule, so it only specifies a pass/fail condition. In this case, 51 lines is bad, 49 lines is good. This is different from using the metrics directly, which can be used to indicate more granular differences in quality (i.e. 49 lines is slightly better than 51, but still quite long).

2.5. Presentation of Feedback

For feedback to be effective, not only the content is important, but also the presentation. Some important aspects of the presentation are a clear structure and timeliness. These aspects were an important part of why students preferred computer assisted feedback (see Section 2.4.2). These aspects also return in some of the design principles found in Section 2.4.3. Dashboards might be a tool well suited for presenting feedback while also considering the structure and timeliness.

A well-designed dashboard can be a powerful tool for presenting data and information while empowering users to interactively explore that data [25]. Dashboards have been suggested or used in several software quality related processes [8][29][54]. In education, both student-facing [10][62] and teacher-facing [50][80] dashboards have been used to present data and information to users. A dashboard can be defined as[6]:

> "... *a visual display of the most important information needed to achieve one or more objectives; consolidated and arranged on a single screen so the information can be monitored at a glance.* " ([25], p. 26)

2.5.1. Design Principles for Dashboards

To be effective the dashboard needs to be designed well. Few [25] has defined a number of design practices for digital dashboards. A summary of the design principles can be found in appendix A

DB1: Organize information to support its meaning and use Careful thought needs to go into the layout of a dashboard [25]. Elements on the dashboard should be organized in groups according to activities, entities and use (e.g. by class, methods, source file, etc.). The organization should also enable meaningful comparison (e.g. comparing the sizes of different classes) while discouraging meaningless comparison (e.g. the size of the class and the length of the name of the class). Finally, groups should be delineated from each other using the least-visible means.

DB2: Maintain consistency to enable quick and accurate interpretation If something is different, it should mean something [25]. Even small inconsistencies like the size of the labels can lead to users searching for meaning. These aspects should be as consistent as possible. Additionally, the choice of display media should be the same when the type of the data and the goal of the visualization are the same.

DB3: Put supplementary information within reach The main dashboard is not able to display all the needed and relevant data on a single screen, so it should be possible to get the needed supplementary information easily [25]. This supplementary information can be provided by pop-up windows or on separate screens. It is best to allow access to the supplementary data through interaction with the information available on the dashboard without cluttering the dashboard with a bunch of controls.

[6]When considering the point of a dashboard fitting on a single screen, Few notes that it can be useful to allow a user to "drill down" for more detailed data based on the initial data presented on the dashboard.

DB4: Expose lower-level conditions While the dashboard itself should only display a high-level overview, it should notify the user when something at a higher level of detail is of interest [25]. If this is the case, it should be easy to navigate to this level of detail.

DB5: Make the experience aesthetically pleasing While the main goal of dashboards is communication, it is still important to make viewing the dashboard a pleasant experience [25]. For this, the choice of colors should be carefully considered. All text and graphics should be high resolution and the font used for the text should be clearly readable. Additionally, content should be aligned when appropriate.

DB6: Prevent excessive alerting When designing a dashboard which uses alerts, one should take care that the alerts are meaningful and not excessive [25]. An excessive number of alerts can lead to users ignoring the alerts.

DB7: Keep viewers in the loop Care should be taken that a dashboard is not too automatic, as this might lead to the user not paying attention [25]. The user should be actively involved in monitoring the data presented.

DB8: Accommodate real-time monitoring The dashboard should consider the interval at which updates are required [25]. For some dashboards the intervals need to be very short (almost real-time), for other dashboards one update a day is enough. It should however always be clear what the timing is of these intervals and how old the data is that is being presented (e.g. by the use of timestamps).

2.5.2. Design Principles for Learning Dashboards

In addition to design principles for dashboards in general, it is important to consider design principles for dashboards used in education. It should be noted that most dashboards described in educational research describe learning analytics (LA) dashboards. However, these principles can still be informative for presenting other data in an educational setting. Charleer et al. [15] define a number of principles based on previous pilots. A summary of the design principles can be found in appendix A.

LD1: Abstract the LA data Often there is a lot of data that can be presented on a dashboard. This data should be abstracted and aggregated for more impact, but should also trigger a student to explore the data further [15]. This further exploration can be supported with interactivity.

LD2: Provide access to the learner artifacts By giving students and teachers access to the learner artifacts, the connection between the artifact and the resulting feedback can be retained [15]. For example, showing the code and the feedback side-by-side. This also supports further exploration and discussion based on the initial presentation.

LD3: Augment the abstracted data While abstraction can help focus the students attention, it also offers room to enhance the presented data further. Charleer et al. [15] give a number of suggestions, for example, providing an overview of students' code including the cyclomatic complexity per method and the number of lines.

LD4: Provide access to teacher and peer feedback Students can learn from feedback given to peers when it is available [15]. It can also help students find quality examples to learn from. One example could be showing parts of good source code (e.g. code with low complexity). By integrating this aspect into the dashboard, it becomes easily accessible. In the same vein, Charleer et al. recommend allowing peer feedback on student artifacts, which can further help students (both the students giving the feedback, as well as those receiving feedback).

LD5: Visualise the learner path An artifact with data only provides a limited view of the student's process and understanding. However, it is possible that feedback given on one assignment leads to changes in the next assignment. A dashboard should try to incorporate the artifact and the data into the bigger picture of the students' learner path [15].

LD6: Integrate the dashboard into the work-flow It is important that dashboards are well integrated into the learning work-flow and are easy to use [15][74]. When the dashboard is not well integrated into the work-flow of students and using the dashboard takes too much effort, it is likely that students will not use it [15]. This integration can work on a course level (e.g. when students need to write source code, the development environment also contains the dashboard) or on an environment level (e.g. the dashboard is integrated into the learning management system). T

LD7: Facilitate collaborative exploration of (learning analytics) data Allowing and supporting collaborative exploration of the data can trigger discussion, exploration, and (self-)assessment [15]. It can also help struggling students find peers who have solved problems they are facing, or find peers with similar problems.

2.6. Summary

One of the aspects of software quality is maintainability. On average 60% of all the costs of a software system are maintenance costs [7][27]. Additionally, maintainability is an important factor for organisations to be able to respond to changes [7]. It is therefore important that students learn to write high-quality and maintainable code. It is clear from the literature that much of the code written by students contains significant quality issues. Additionally, it is shown that these issues rarely get fixed. Overall, there is room for improvement when it comes to the quality of student code [37].

Students need feedback to improve the quality of their code. Feedback is an effective tool in education [28]. However, the quality of the feedback is important. Feedback design principles can inform the design of feedback to ensure the quality and effectiveness of students' learning process to write high quality code [51]. These principles will serve as meta principles for the design principles for software metric based feedback on code quality. Additionally, the design principles for formative feedback will also be used to guide the design and development of a prototype in addition to the design principles for software metric based feedback on code quality. This prototype will be used to evaluate the principles defiend in this research. Besides the content of the feedback, how it is presented is also important.

Dashboards can assist in showing the feedback in an effective manner. Just like the content, the quality of the dashboard is important. For this, dashboard design principles can be used [25]. Additionally, design principles for learning analytics (LA) dashboards can also provide guidelines for a feedback dashboard, even if it does not present LA data [15]. These principles will be used to inform the presentation of the software metric based feedback in the prototype.

3 Problem Analysis

Software plays an ever increasing role in our lives. As our dependence on software increases, it becomes more important than ever to ensure the usability, stability, and safety of this software. To achieve this, software must be able to change and evolve as our society changes and evolves. Additionally, it must be possible to fix defects found down the line. On top of that, iterative processes like Agile development have moved more development work from the initial development stage to the evolution stage [58]. In short, software must be of high quality and maintainable. One of the major factors effecting maintenance of software is the internal quality of the software and the quality of the code [9][67].

As future developers, students need to learn how to build high quality maintainable software [58]. Much of this area of research focuses on novice, first year students. However, there is little attention for senior undergraduate students. For this group of students, advanced and open-ended programming assignments are common.

Previous research shows significant code quality problems in student code [37]. Students rarely fix these issues. Additionally, current tooling is not sufficient to significantly improve the quality of the code students write. It is clear that more attention for code quality is needed in programming education [11][12][37][38].

To empower students to write high quality code, they need in-depth, detailed, and specific formative feedback [28]. However, providing this feedback requires a lot of time, which teachers often do not have. On top of this, students should be able to iterate on their feedback so they can fix issues in their code and improve over time.

From the perspective of teachers, it is important to know how the quality of the code students write evolves. Feedback given to students on code quality might give teachers insight into common issues that the students have. Additionally, it might also provide teachers with good examples from peers that can be used to help other students. In short, feedback can be used to enhance teacher pedagogical content knowledge.

To summarize, students need to learn how to write high quality maintainable software. However, many student programs contain significant issues that decrease the quality and maintainability. Students should be empowered to fix these issues and learn how to avoid them in future assignments. To do this, they need good formative feedback. Timeliness, how fast students receive feedback, is an important aspect of this feedback. The feedback might also be useful to enhance teacher pedagogical content knowledge.

4 RESEARCH DESIGN

As mentioned in Section 3, it is important for students to learn how to write high quality, maintainable code. For this to happen, students need to be aware of code quality in general and the quality of the code that they write. In addition they need guidance in improving their code. The objective of this research is to increase the understanding of code quality among advanced undergraduate students.

The general approach for this research is design based research. This is a common approach used in education which aims to combine theory and practice.

4.1. CONTEXT

This research is done in the context of second, third and fourth year undergraduate students. These students are passed the introduction phase of their education. This means that many of their assignments are larger, more open-ended, and often project-based. These aspects make model solutions, which are often used in introduction courses, difficult to provide.

The participants of this research are second, third, and fourth year undergraduate students of a HBO-ICT program from Zuyd University of Applied Science in the Netherlands. During the different courses, students receive open ended, project-based programming assignments. From the third year forward, these assignments are often provided by partner organizations of Zuyd University of Applied Science. Different groups in the same course work on different assignments. For example, for an AI course, one group of students might get a project to detect empty parking spaces on a parking lot, while another group might get a project to determine crop quality based on drone images. This combined with the open ended nature means that it is very difficult or even impossible to provide model solutions for the assignments. Both students and teachers from the same program have participated in this research.

4.2. DESIGN RESEARCH IN EDUCATION

Design research (also known as design based research or development research) is a problem-oriented interdisciplinary type of research that combines aspects from research and from design and development. Van den Akker et al. [71] give a number of related terms that describe similar approaches to design research: design studies/experiments/research, development(al) research, formative research/inquiry/experiments/evaluation, action research, or engineering research.

It is often hard to apply the insights and knowledge gained from traditional research in practice [71]. The results are often too narrow, too artificial, too superficial, and too late. Education is a highly complex environment and many interventions have a wide scope and are ill-defined. To implement good interventions, designers need applicable knowledge and a solid theoretical foundation.

Some researchers argue that an iterative approach which considers both research and design aspects would work better in an education setting [59][71]. Both policymakers and researchers want more direct contribution to education. Additionally interaction between practitioners and researchers might incentivize practitioners to take interest in and contribute to theoretical insight and knowledge.

While practical design and development and design research consist of many of the same activities, they are not the same. Van den Akker et al. [71] define a few key differences. First, design research involves more extensive preliminary investigation (e.g. literature reviews, case-studies, etc.) and general theoretical embedding. Additionally, design research emphasizes empirical evidence to show that interventions are effective. Finally, systematic documentation and analysis are important parts of design research, which is not usually the case in practical development work.

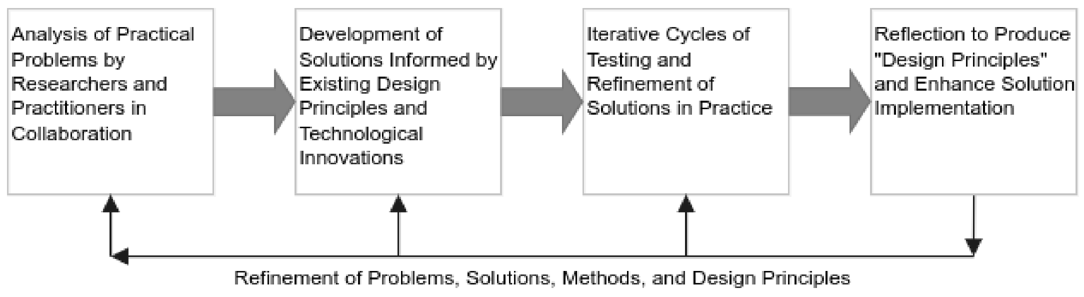
Another difference between practical design and development and design research can be found in the artifacts that are delivered by both. Where practical design and development is only concerned with the practical output (e.g. a tool), design research also produces some theoretical artifact in terms of design principles [71]. These design principles, rooted in educational theroies and validated through practice, can inform effective educational interventions.

4.2.1. Design Research Cycle

As mentioned, design research uses an iterative process. This process, as described by Reeves [59], starts with some practical problem, which is analysed by practitioners and researchers. A solution is then developed using existing theoretical knowledge (in the form of design principles) and technological innovations. This solution is developed and tested in an iterative manner in a practical setting. Finally, design principles are produced, refined, or adapted by reflecting on the previous phases. In this final phase, further improvements of the solution are also considered. After the final phase, the cycle starts again, either from the first phase or from one of the phases in between (e.g. it is possible to go from the final phase to the "iterative cycles of testing and refinements" phase). Figure 4.1 gives an overview of this cycle.

4.2.2. Design Principles

In the Design research cycle shown in Figure 4.1, the last phase mentions the formulation of design principles. Like design patterns in software engineering, design principles used in design research aim to capture knowledge that can be used in new design and development work [36]. These design principles generally use a format such as:


Analysis of Practical Problems by Researchers and Practitioners in Collaboration
Development of Solutions Informed by Existing Design Principles and Technological Innovations
Iterative Cycles of Testing and Refinement of Solutions in Practice
Reflection to Produce "Design Principles" and Enhance Solution Implementation
Refinement of Problems, Solutions, Methods, and Design Principles
Figure 4.1: Design Research Cycle [59]

> "*If you want to design intervention X [for the purpose/function Y in context Z], then you are best advised to give that intervention the characteristics A, B, and C [substantive emphasis], and to do that via procedures K, L, and M [procedural emphasis], because of arguments P, Q, and R.* " ([71], p. 9)

4.2.3. Design research in this study

In this research, design based research is used to generate a set of appropriate design principles. The goal of these principles is to guide the design and development of a prototype that provides automated code quality feedback based on software metrics. The (meta-)design principles for formative feedback (see Section 2.4.3) will inform the design principles generated in this research. The principles generated in this research serve as functional requirements for a prototype. Additionally, there are a number of design principles for dashboards in general and learning dashboards specifically (see Section 2.5.1). These design principles informed the specific design of how the system works and looks (non-functional requirements).

4.3. Research questions

The main research question is **"How can software metrics be used to facilitate formative assessment on code quality in project-based, open-ended programming assignments?"**. To answer this question, three sub-questions have been defined:

- **(RQ1)** Which software metrics can be used to generate code quality feedback?
- **(RQ2)** How can software metrics be used for provide formative feedback on code quality?
- **(RQ3)** How does software metric based formative feedback affect code quality knowledge among students?

4.3.1. RQ1: Which software metrics can be used to generate code quality feedback?

Since software metrics have not yet been used to automatically generate formative feedback on code quality in a similar context to this research, it is not clear which software metrics can be used to generate this type of feedback. The aim of this research question is to give an overview of which software metrics are available and can be used to generate formative feedback on code quality aspects of participants' programming assignments.

4.3.2. RQ2: How can software metrics be used for provide formative feedback on code quality?

Besides selecting the software metrics in RQ1, research is also needed into how these metrics can be used to generate the feedback. This research question aims to give a number of initial principles for the generation and design of the software metric based formative feedback. The aim is to use these principles to build a prototype that automatically provides formative feedback on code quality.

4.3.3. RQ3: How does software metric based formative feedback affect code quality knowledge among students?

After defining a number of principles in RQ2, it is important to test these principles in practice. The ultimate goal of this research is to improve knowledge of code quality among students. It is therefore important to evaluate the effects on code quality knowledge among students who used the prototype. Ideally, students have a better understanding of code quality after receiving the feedback from the prototype. However, it might also be the case that students do not understand the feedback or that students do not find the feedback useful.

4.4. Research methodology

During this research, a prototype will be designed and built which assesses the code quality of student code and provides the student with formative feedback aimed at improving the quality of their code. The process of developing and evaluating this prototype will use the phases as defined in the Design Research Cycle (see Figure 4.1)[59].

The problem analysis phase has been completed and has been described in chapter 3. RQ1 and RQ2 will be answered as part of the development of solutions phase. A single cycle of the testing and reflection phases will be done to answer RQ3.

4.4.1. RQ1: Which software metrics can be used to generate code quality feedback?

During the design and development of the prototype, we have reviewed the literature to identify software metrics that can be used to generate formative feedback on code quality. Two main aspects are important to consider for the software metrics. The first aspect is how well the metric correlates with code quality. The second aspect is how well a metric lends itself to the principles for good formative feedback (see Section 2.4.3). A metric needs to be well suited for both aspects to be useful for generating formative feedback. Additionally, there are some practical aspects that need to be considered as well. First, the metrics need to be easy to implement and calculate. Finally, the metric must support (i.e. be possible to calculate for) common programming languages used within programming education. In the context of this research this means specifically Python and C#.

To find software metrics, we have used the following query *("code measure" OR "code metric" OR "software measure" OR "software metric" OR "code measurement" OR "software measurement") AND ("code quality" OR "software quality" OR "maintainability" OR "maintenance")*. The ACM and IEEE databases have been used for this literature review. From the results, machine learning and AI based approaches were removed. This will be done based on the title, the keywords, and the abstract. Next we created a list of available metrics. To do this, papers presenting metrics are selected from the search results by scanning the title, abstract, and keywords. For each metric in the list of metrics, the search results have been scanned for related papers (including review papers). Where possible, the original source of the metric is retrieved. The original metric papers and the related papers are scanned for the metrics' relation to code quality. Metrics that have no relation to code quality will be discarded. The metrics that are left will be reviewed to check how well they fit with the principles for good formative feedback (See Section 2.4.3).

The output of this literature review is a set of software metrics which are relevant to code quality, fit well with the design principles for formative feedback, and are easy to implement and calculate in the given context. The answer to this question can be found in section 5.2 of chapter 5.

4.4.2. RQ2: How can software metrics be used for provide formative feedback on code quality?

To translate the meta design principles (see Sections 2.4.3 and 2.5) into specific design principles applicable to the context of code quality in advanced programming education, it is important to get student and teacher input. To collect this input, two focus group session have been done. The goal of these focus group sessions is to provide input for and help define a number of initial design principles for giving feedback on code quality. Both groups, which consist of entirely different participants, have been asked to brainstorm on the same set of questions. This is done to see if the two groups come up with similar results independently of each other.

Both sessions have been done in-person. In each session, four students and one teacher participated. The students were third or fourth year students of the HBO-ICT bachelor program at Zuyd University of Applied Science. Their ages were between 20 and 26. All students that participated in the focus group sessions were male[1]. The teachers were recruited from the same program as the students. Each session took about an hour and a half to two hours.

The two sessions both had the same setup. Each session started with a short explanation of code quality and the metrics selected in Chapter 5. Additionally, participants received a rubric based on the meta design principles. This rubric is shown in Table 4.1[2]. For this rubric, we have ordered the meta design principles found in Sections 2.4.3 and 2.5 into five categories. These categories are content, current level and expected level, motivation and self-regulation, and finally presentation. The participants were asked to have an open discussion on each of the questions listed in the rubric. In addition to the rubric, at the end of the session, participants were asked for any additional input that was not covered in the rubric. Finally, at the end of the second session, the participants were shown the concept principles that had been generated during the first session.

We combined the results from the focus groups with the meta design principles defined in sections 2.4.3, 2.5.1, and 2.5.2 and with the metrics selected in RQ1 to define a number of design principles for formative feedback on code quality. The results of the focus group sessions and the design principles based on these results can be found in section 6.3 of

[1]The population of the HBO-ICT program at Zuyd University of Applied Science is about 90% male, which could have influenced the fact that all student participants of the focus group session were male.

[2]The original rubric was in Dutch. It has been translated to English for this thesis.

#	*Question*	*Category*	*Related Meta Design Principles*
1.a	How should the software metric results be communicated to be useful as feedback?	Content	FB3, LD2
1.b	Which additional information is needed, besides the metric data, to use the feedback?	Content	FB3, DB3, LD3
2.a	How can the feedback help students understand their current level in the context of code quality?	Current level and expected level	FB6
2.b	How can the feedback help students understand the expected level of the quality of the code?	Current level and expected level	FB1
2.c	How can the feedback help students close the gap between the current level of the quality of the code and the expected level	Current level and expected level	FB6
3.a	How can the feedback affect the students' motivation in a positive manner?	Motivation and self-regulation	FB5
3.b	How can the feedback encourage the students' self-regulation?	Motivation and self-regulation	FB2
4.a	How should the feedback be presented?	Presentation	DB1, DB6, DB7, LD1, LD5, LD6
4.b	How should the feedback be organized?	Presentation	DB1, DB4

Table 4.1: Focus group rubric

chapter 6. These design principles have been implemented in a prototype.

4.4.3. RQ3: How does software metric based formative feedback affect code quality knowledge among students?

The feedback generation and presentation is implemented into the prototype using the metrics from RQ1 and the design principles defined in RQ2. Additionally, the content of the feedback is based on the information gathered during RQ2. A description of the prototype can be found in section 7.1. The technical details of the prototype can be found in appendix E.

For each of the student evaluations, the participants were asked to work on a programming assignment of their choice (e.g. an assignment from their current course). At the start of the programming session the students were asked to upload the current version of the code to the prototype and review the feedback. After working on the assignment for about half an hour the students were asked to upload the code once again and review the feedback. After the programming session, the students were asked a number of questions to see how useful they found the feedback given in the prototype (see Appendix F). The goal of the questions in the before interview was to gain an understanding of what participants know about code quality before using the prototype, to get an indication on how difficult they find it to learn new concepts, and whether they have experience with using other tools for their programming assignments. The goal of the after interview is to evaluate the usefulness of each design principle and to get possible areas of improvement for each principle. Additionally, the students were asked for general input on future improvements of the prototype and the underlying design principles.

Like the focus group sessions, the participants of the student evaluations were third and fourth year students from the HBO-ICT program of Zuyd University of Applied Science. None of the students that participated in the evaluations were part of any previous sessions done for this research. All the students had some prior knowledge of code quality from earlier courses, but found it somewhat difficult to apply this knowledge in practice. All students had experience with learning new concepts within the context of programming. Most students had at least some experience with using different programming tools within their workflow.

A senior teacher of the same HBO-ICT program participated in the teacher evaluation of the prototype. The teacher evaluation involved an exploration of the prototype. After exploring the prototype, the design principles and their implementation in the prototype were discussed. Like the student that participated in the evaluations, the teacher was not involved in any earlier parts of this study.

4.4.4. Main research question: How can software metrics be used to facilitate formative assessment on code quality in project-based, open-ended programming assignments?

We will use the insights gained from RQ1, RQ2, and RQ3 to answer the main research question. The software metrics gathered during RQ1 and the design principles formulated for RQ2 can be used to determine how software metrics can be used to facilitate code quality feedback. RQ3 attempts to validate the approach formulated by RQ1 and RQ2 and aims to generate input for further refinement and iteration on the design principles and the prototype based on those principles.

4.5. Ethical Considerations

Since this research involves student and teacher participation, it is important that these participants are not negatively affected by participating in this study. This research can raise two possible ethical concerns.

The first concern is the privacy of the participants. Both the law (GDPR), the guidelines provided by cETO[3], and the guidelines of Zuyd University of Applied Science (the context for this research) place an importance on privacy. This concern applies to both the students and the teachers. The privacy of the participants could be could be violated during the focus group sessions (RQ2) and the evaluations (RQ3).

The second concern, which applies only to the students, is that the research should affect the grading process as little as possible (excluding the effect the research has on the understanding of code quality). The feedback a student receives might affect how teachers perceive the work of that student and it might therefore affect the grade. For example, a student who receives a lot of feedback might bias a teachers' grading, either negatively (e.g. the student receives a lot of feedback because the code is of low quality) or positively (e.g.

[3]Since this study involved student and teacher participation, it had to pass a review by cETO. For detailed documentation of this review, see Appendix B.

the student is actively asking for feedback and therefore engaged with the material).

4.5.1. Participation and informed consent

Participation in any part of the research is completely voluntary. Additionally, before participating, potential participants have been informed about the fact that participation is voluntary and on what data is gathered, how this data is processed, and how this data can be deleted. This will be done in both verbal and written form. Consent will be given by filling in an online form based on the example form provided by Research Ethics Committee (cETO)[4].

4.5.2. Data gathered

At several points during this study, we have gathered, stored, and processed data. First, the two focus group sessions have been recorded and transcribed (RQ2). In addition to the focus group sessions, the evaluations (RQ3) have also been recorded. These evaluations also involve observing a programming session for each participant. For each programming session the screen of the participant has been recorded.

4.5.3. Privacy Measures

Several measures were taken in this study to address the privacy and grading concerns raised. These measures aim to ensure the privacy of the participants and fair grading for the students.

The recording of the focus group sessions are deleted after the transcripts have been completed. In the transcripts, the participant names (and any other names mentioned) will be manually replaced be generic identifiers (e.g. StudentA, StudentB, TeacherA, etc.). This ensures that the data gathered during this part of the study is anonymous.

We have reviewed the recordings made during the evaluations for personally identifiable information shown on screen during the programming session. This information will be covered by black bars in the recording. In addition, the interviews done for the evaluation have been processed in the same manner as the recordings of the focus group sessions.

One area that might also contain personally identifiable information is the code students upload to the prototype. To reduce the likelihood of this happening, only relevant files are uploaded to the prototype back-end. This is implemented using a whitelist approach (i.e. only file types explicitly listed in the whitelist are uploaded). This filters out data from the development environment (e.g. git folders, build folders, etc.), which often contain personal data. If personal data is found in the assignments uploaded by students, it has been removed manually.

Table 4.2 gives an overview of the data gathered and the measures taken to preserve privacy and to ensure fair grading.

4.6. Summary

This research aims to generate a set of initial design principles for providing automated formative feedback on code quality using software metrics. This is done using a design research approach. In the next three chapters (Chapter 5, Chapter 6, and Chapter 7), the

[4]https://www.ou.nl/documents/40554/361215/11_Example_online_Informed_Consent.pdf

Data	*Measures*
Focus group recordings	• Recordings have been deleted after the transcriptions have been completed.
Focus group transcriptions	• Participant names are replaced by generic identifiers (i.e. StudentA, StudentB, TeacherA).
Evaluation recordings	• Personal information have been covered by black bars. • Participant names are replaced by generic identifiers (i.e. StudentA, TeacherA).
Prototype data (code uploaded by students)	• Only code files supported by the prototype are uploaded to the prototype. • Basic personal information (e.g. email addresses, student numbers, etc.) have been removed from the uploaded files where possible.

Table 4.2: Data gathered and measures to ensure privacy and fair grading

sub-questions have been answered. Finally, in Chapter 9 we have answered the main research question.

5
SELECTING SOFTWARE METRICS FOR CODE QUALITY FEEDBACK FROM THE LITERATURE

To answer RQ1, "Which software metrics can be used to generate code quality feedback?", a literature review has been done to find suitable metrics for automatically generating feedback on code quality.

The search query *("code measure" OR "code metric" OR "software measure" OR "software metric" OR "code measurement" OR "software measurement") AND ("code quality" OR "software quality" OR "maintainability" OR "maintenance")* resulted in a total of 2679 results (ACM: 682; IEEE: 1997). From these results, a list of code quality metrics has been extracted. This resulted in 42 relevant sources from which a list of 90 distinct software metrics have been extracted. See section 4.4.1 for details of the research method for this question.

5.1. SELECTION CRITERIA FOR SUITABLE METRICS

To select the correct metrics, a number of criteria need to be formulated. To do this, we have used the literature from both the software engineering and education perspectives. For the software engineering perspective, the requirements for software metrics given by Heitlager et al. [30] have been used. For the education perspective, we have used the design principles for good formative feedback defined by Nicol and Macfarlane-Dick (see Section 2.4.3) and the results found by Hattie and Timperley [28].

The selected metrics should allow for root cause analysis (CRC, or Criteria Root Cause). Heitlager et al. note that metrics that enable root cause analysis help programmers understand which areas of a code base need attention [30]. According to the design principles defined by Nicol and Macfarlane-Dick, feedback should enable students to assess their own work [51]. To do this, it is not enough to give students a general measure of the quality of their code. Instead, the feedback should point to specific parts of the code that need improvement. Additionally, if a metric allows for more granular analysis, it can also identify good parts of the students' code, which might have a motivating effect.

According to Heitlager et. al, software metrics should be easy to understand and to explain [30]. This aspect is also important for feedback. The goal of feedback is not only to point out mistakes or misunderstandings, but also provide guidance on how to improve [28]. The design principles FB2 (feedback should facilitate self-assessment) and FB6 (feedback should provide opportunities to close the gap between current and desired performance) also highlight this aspect of feedback [51]. When this idea is applied to code quality, the goal of the software metric based feedback is not only to inform students about the quality of their code, but also to help students understand how to improve the quality of their code. To ensure that students understand the feedback provided (including how it is generated), and get guidance on how to improve their code, the metrics should not contain any magic numbers that have no direct relation to some aspect of the source code (CMN, or Criteria Magic Numbers), since these magic numbers would hide the relationship between the metric result and the aspect of the code that is measured. Additionally, the metrics should not be a combination of two or more metrics (CCM, or Criteria Combination Metric). In this case, the component metrics used to construct the combination will be used.

The metrics should be as technology independent as possible [30]. To apply this idea to an educational setting, the metrics should support programming paradigms that are commonly taught (CMP, or Criteria Multi Paradigm). In this case the metrics should at least support the object-oriented and procedural programming paradigms. Additionally, it should be easy to implement the metrics. For this reason, the metrics should be calculable using only static analysis (CST, or Criteria STatic) and the calculations should be easy to calculate or should be provided by existing tools (CIM, or Criteria IMplementable).

An overview of the criteria is shown in Table 5.1. Each of these criteria is given a score from 1 to 3. For criteria that are either true or false, scores of 3 and 1 are used. How each score is determined for each criteria is described in the rubric shown in Table 5.2.

Code	*Criteria*	*Description*
CRC	Root cause	The software metric should allow for root cause analysis.
CMN	Magic numbers	The software metric should not contain magic numbers (i.e. numbers that do not have a direct relation to some aspect of the code).
CCM	Combination	The software metric should not be a combination of two or more other metrics.
CMP	Multi paradigm	The metric should be able to support multiple programming paradigms. At least the object-oriented and procedural paradigms.
CST	Static	The metrics should be calculable using only static analysis techniques.
CIM	Implementable	The metric should be easy to calculate and this calculation should be easy to implement. Alternatively, the metric should have existing tools for calculation.

Table 5.1: Software metric criteria

Criteria	*Score 1*	*Score 2*	*Score 3*
CRC	Does not allow for root cause analysis	Does allow for some root cause analysis, but not down to a specific unit (such as a method)	Allows for root cause analysis down to specific units
CMN	Does contain magic numbers	Not applicable	Does not contain magic numbers
CCM	Is a combination of one or more metrics	Not applicable	Is not a combination metric
CMP	Does not support the required paradigms (procedural and object oriented programming)	Supports one of the two required paradigms	Supports both required paradigms
CST	Requires dynamic analysis	Can be done using static analysis in some cases	Only requires static analysis
CIM	Has no existing implementation for the required languages (Python and C#) and requires parsing to calculate or the metric does not support the required languages	Has no existing implementation and requires some parsing to calculate	Has an existing implementation for the required languages or requires very minimal parsing

Table 5.2: Software metric criteria rubric

5.2. Answering RQ1: Selecting suitable metrics

For each of the 90 metrics, the criteria described in Section 5.1 have been given a score between 1 to 3. 1 in this case means that the metric does not pass the criteria and 3 means that the metric passes the criteria. A score of 2 means that the criteria only apply to a certain extent or only in certain conditions. The sum of the criteria scores is used as an overall score for a given metric. The top results (a score of 18) are shown in Table 5.3 (the complete list of metrics and scores is given in Appendix D).

#	*metric*	*CRC*	*CMN*	*CCM*	*CMP*	*CST*	*CIM*	*total*
1	Cyclomatic complexity	3	3	3	3	3	3	18
2	Extended cyclomatic complexity	3	3	3	3	3	3	18
8	Lines of code	3	3	3	3	3	3	18
12	Duplication	3	3	3	3	3	3	18
19	Lines of comments	3	3	3	3	3	3	18

Table 5.3: Metric score matrix

From the metrics with a score of 18, three metrics stand out. These are the (extended) cyclomatic complexity (CC) metric[1] [44], the lines of code metric, and the duplication metric. The lines of code and duplication metrics have been extracted from the SIG maintainability model paper by Heitlager et al. [30], however, no single original source for these metrics could be found. While comments are likely an important factor of high quality code, it is not clear whether lines of comments in itself is a useful metric for measuring this. Other approaches that actually look at the content of the comment might be more suitable for providing feedback on comments in students' code. For this reason the lines of comments metric is not used further in this research.

One interesting aspect is that issues related to all three of these metrics (CC, lines of code, and duplication) are also often found in students' code as stated by Keuning et al. [37]. This confirms that these three metrics might be good candidates for a first iteration of the prototype.

5.3. Summary

Three metrics have been found that could be good first candidates for generating formative feedback on code quality. These metrics are (extended) cyclomatic complexity, lines of code, and duplication. These three metrics will be used for the rest of this research. However, just the metric data is likely not sufficient for students to improve their code. In Chapter 6, discussions with students and teachers will be used as input to formulate a number of design principles to guide the use of the metrics found in this chapter to give effective formative feedback on code quality.

[1]Cyclomatic complexity and extended cyclomatic complexity differ only in which elements are included when calculating the value (e.g. extended cyclomatic complexity counts each logical 'or' or 'and' operation that is part of an if-statement as a seperate branchpoint, were the base definition of cyclomatic complexity only considers the if-statement itself as a branchpoint). Therfore we have used CC to refer to both variants of cyclomatic complexity.

6
DEFINING DESIGN PRINCIPLES FOR SOFTWARE QUALITY FEEDBACK

To translate the software metrics selected in chapter 5 (cyclomatic complexity, lines of code, and duplication) into feedback that is useful for students, we will define a number of design principles. These principles will use input from two focus group sessions, each with four students and one teacher. In both cases, the participants received a short explanation of code quality and the selected metrics. Additionally, the participants received a list of general meta design principles for feedback based on the work by Nicol and MacFarlane-Dick [51] (see section 2.4.3), by Few [25] (see section 2.5.1), and Charleen et al. [15] (see section 2.5.2). The design principles defined based on the input of these sessions have been given in Table 6.1. For details on how the focus group sessions were done, see Section 4.4.2.

6.1. CODE QUALITY FEEDBACK DESIGN PRINCIPLES

Based on the input of the participants of the focus group sessions, a number of design principles for feedback on code quality have been defined. The design principles are shown in Table 6.1. Each design principle has been coded by CQ (Code Quality) followed by a number. The participants of both sessions independently came up with design principles CQ1 to CQ6. CQ7 came from the first focus group session while CQ8 came from the second focus group session. The following sections will describe each design principle in detail.

6.1.1. CQ1: LINK THE FEEDBACK TO THE SOURCE CODE

A major complaint students had with feedback that they received is that often it is not specific enough. They understood the points made by, for example, a teacher, but it was not always clear which part of their assignment the feedback applied to. One student in the second session mentioned a specific case in the course the student was taking were it was not clear which part of a programming assignment the feedback was applicable to. When it comes to feedback on code quality, it is therefore important that students know which parts of the code the feedback is applicable to. As one student mentioned:

> "*... in a way that you can visually see where in the source code a certain cyclomatic complexity value applies to.* " (session 1, student 1)

Code	*Principle*	*related meta-principles*
CQ1	Link the feedback to the source code	FB2, DB1, LD2
CQ2	Provide examples of high quality code	FB1, FB3, FB5, DB3, LD3
CQ3	Provide the metric data with the feedback	FB2, FB3, DB4, LD1
CQ4	Provide comparisons of the metric results to both previous results and benchmark results	FB1, FB2, FB5, FB6, LD3, LD5
CQ5	Provide strategies to help improve the quality of the code	FB3, FB6, DB3, LD3
CQ6	Provide general information about code quality and software metrics in addition to the feedback	FB3, DB3, LD3
CQ7	Ensure that feedback is easy to access, but not intrusive	DB6, LD6
CQ8	Allow the feedback to be easily shared	

Table 6.1: Software metric feedback design principles

It is important to guide students to the parts of their code that could be improved, and to parts of their code that are high quality. This linking of feedback to code should be done both in the feedback itself and in the presentation of the feedback. Having the code and the feedback available at the same time helps students better understand which part of the code the feedback is relevant for.

In addition, linking feedback to specific parts of the code allows students to identify areas of their code that need attention. The code might be good quality overall, except for some parts. As the teacher in session two put it:

> "*Maybe 9 out of 10 methods are fine. However, one stands out as being bad. Pointing this out would help.* " (session 2, teacher)

6.1.2. CQ2: Provide examples of high quality code

According to the meta design principles (see section 2.4.3) two aspects that are important for feedback are showing students what they should be aiming for (FB1) and motivating students (FB5)[51]. By providing examples of high quality code, students can learn what to aim for and it could empower students to compare their own code to the examples provided. The main aspect of these examples was motivational effect it had on the students. One student said in relation to providing code examples:

> "*I think it is good to see what the student is working towards. One thing that motivated me with programming is seeing were I was going and what I could do if I progressed [my programming skills].* " (session 1, student 4)

In addition to the motivational aspects, examples of high quality code might also be used to provide strategies for solving problems in the students' own code. Explanations of why these samples are of high quality might help with this.

However, the examples should not be so advanced that students are not able to understand the code. Examples of this might be code that uses more advanced programming language concepts that make for higher quality code, but with which the student is not yet familiar. A mismatch between the students knowledge and the examples shown could have a demotivating effect. As the student from the previous quote mentioned:

> "*However, the examples should not be overwhelming, that you feel like you will never reach that level.* " (session 1, student 4)

6.1.3. CQ3: Provide the metric data with the feedback

One aspect that students found important is understanding how the feedback was generated. For this reason, it is important to provide students with the calculated metrics data. It allows students to get an understanding of how different code elements and different changes made to those elements affect the metric data (and thus the feedback). This understanding is something they can apply to future projects as well as to the current assignment. For example, when a student learns that including loops in a method increases the cyclomatic complexity of that unit, they can use this information in future assignments. As one student put it.

> "*I might have built a complex piece of code with too many loops in it, that is not very practical and here is the metrics data that shows it. This is information that you take with you to the next project you are working on.* " (session 2 - student 1)

While it was not directly mentioned in either focus group session, it might be useful to include explanations of how a metric is calculated. This might increase the likelihood that students gain an understanding of what affects the metric values calculated for their code and why they received certain feedback.

6.1.4. CQ4: Provide comparisons of the metric results to both previous results and benchmark results

An important aspect of feedback is that students should perceive it as being useful. It should provide students the opportunity to improve their performance over time as to get closer to the desired level (FB6, see section 2.4.3)[51]. By giving the students the opportunity to receive feedback on the same code multiple times, it enables comparison of the current quality of the code with a previous version for the same assignment. This might have an motivating effect, which is also an important factor of good feedback (FB5, see section 2.4.3). Additionally, this comparison facilitates students to learn how different changes affect the metric results. This fits well with the previous principle we have defined. On the topic of providing opportunity for comparison, one student said:

> "*If I have my old feedback and my new feedback, then I can compare the two, I can see if I have progressed.* " (session 2, student 1)

Besides providing comparisons between different versions of a students' code, it might be useful to provide comparisons with benchmarks. This helps clarify what the students should be aiming for. Especially for third and fourth year students it is relevant so see benchmark values of professional software. As one teacher said:

"*As a teacher it is interesting to see. You would hope that third and fourth year students at one point reach a level that might be expected of a junior professional programmer.* " (session 2, teacher)

During both sessions students mentioned that they would find it interesting to see the comparison to other students in their year in addition to comparisons to professional benchmarks. This might provide some sub-goals between the current level of the students' code quality and the professional benchmark values.

6.1.5. CQ5: Provide strategies to help improve the quality of the code

As mentioned for CQ1 (see section 6.1.6), one issue students experienced with feedback was that it was often not clear how to improve their work. By providing students with strategies on how to improve the quality of their code, they are enabled to close the gap between their current understanding and the expected performance. It should be noted however that these strategies should not only be applicable to the current situation, but also to similar situations in future assignments. These strategies also helps students who know which code needs to be improved, but who do not know how it can be improved. One student said:

"*... for example, if you have a very complex [piece of code]... with a high CC value, you could provide feedback on how complexity could be reduced.* " (session 2, student 3)

Providing strategies to common issues also might help students become aware of common code quality issues. For example, a common issue might be a method that has more than one task. By providing information on this issue and how to solve it (e.g. splitting a single method with multiple tasks into multiple methods with each its' own task), it might help students avoid this issue in the future. As one teacher said:

"*It might be helpful to provide information about the types of issues that often occur.* " (session 1, teacher)

6.1.6. CQ6: Provide general information about code quality and software metrics in addition to the feedback

The feedback should not only empower students to improve their code, but should also help them understand why it is important to write high quality code. To achieve this, it might be useful to provide general information on code quality and software quality in addition to the feedback. This also helps students understand how the bigger picture of software quality applies to their own work. One example mentioned during the first session is linking the feedback to the relevant aspects of the ISO 25010 standard for software quality. One student mentioned:

"*For example, a tooltip that links [the feedback] to an ISO [25010] explanation, another standard, or other general information.* " (session 1, student 1)

Another type of information that could be provided within the prototype is information about common issues found in code bases. This would provide students with more general patterns in addition to the feedback on their own code. As one teacher said:

> "*Maybe it also helps to provide general information about the types of issues that occur, so you have both general information and context specific information.* " (session 1, teacher)

6.1.7. CQ7: Ensure that feedback is easy to access, but not intrusive

One aspect that was very important for the students is that the feedback should be easily accessible, but at the same time should not hinder the programming workflow while making assignments. In short, the feedback should integrate into the usual programming workflow. This is also shown in one of the design principles for learning dashboard design (LD6, see section 2.5.2)[15].

One aspect that is important to ensure that the feedback does not distract is to ensure that not all information is shown at once. Only show the information that is directly relevant, but allow students to drill down for more details. One bad example of this that the students mentioned was many IDEs that show a lot of warnings. Because it is overwhemling, these are often ignored. As one teacher mentioned:

> "*I think it should be simple to use and not overwhelming. Be careful with the amount of information that is shown directly.* " (session 1, teacher)

6.1.8. CQ8: Allow the feedback to be easily shared

The feedback and metric data should empower students to ask for more focused feedback and help teachers identify code quality issues in their students' code. In addition it might also help students to collaboratively explore the quality of their code. As one student put it:

> "*For example, if we have an assignment, I could ask you [to another student], what is your CC value for a certain part.* " (session 2, student 4)

6.2. Relation between code quality design principles to general design principles

Based on the results of the focus group sessions, it is not possible to give a clear one-to-one translation for each meta design principle into a context specific design principle for automated code quality feedback based on software metrics. However, each code quality design principle described above does have a relation to one or more meta design principles. These relations are shown in Table 6.2.

CQ1 (Link the feedback to the source code) is related to developing students' ability to assess their own work (FB2). By showing specifically what parts of a code base the feedback applies to, the students might be able to identify patterns in the types of issues that occur.

CQ2 (Provide examples of high quality code) might help students to understand what high quality code looks like, thus making it clear what they should aim for (FB1). The examples might also give students a frame of reference to compare their own work to (FB3). Additionally, as one student mentioned, seeing what they are working towards might have

	CQ1	CQ2	CQ3	CQ4	CQ5	CQ6	CQ7	CQ8
FB1		✓		✓				
FB2	✓		✓	✓				
FB3		✓	✓		✓	✓		
FB4								✓
FB5		✓		✓				
FB6				✓	✓			
FB7								✓
DB1	✓							
DB2								
DB3		✓			✓	✓		
DB4			✓					
DB5								
DB6							✓	
DB7								
DB8								
LD1			✓					
LD2	✓							
LD3		✓			✓	✓		
LD4								
LD5				✓				
LD6							✓	
LD7								✓

Table 6.2: Relation between meta design principles and code quality feedback design principles

a motivating effect (FB5). Examples are also one of the pieces of extra information besides the feedback itself that are used to provide additional insights (DB4 and LD3).

CQ3 (Provide the metric data with the feedback) might empower students to judge the quality of their own code (FB2). For example, understanding a complexity metric such as cyclomatic complexity might help students judge the complexity of a method they have written. The metric values also give a quantitative view of their code (FB3). It also exposes the underlying mechanism for generating the feedback, which might be useful (DB4). It also abstracts the feedback to a value that can be compared to other values (LD1), for example, to see if one method is more complex than another.

CQ4 (Provide comparisons of the metric results to both previous results and benchmark results) gives student an understanding what metric values they should be aiming for by providing benchmarks (FB1). By allowing students to submit their code for feedback and to see the differences in the metric values between the versions, it allows students to close the gap between their current performance and the desired performance (FB6). By seeing how changes in their code affect the metric values from one version to another, students might identify how to determine the quality of their code themselves (FB2). The changing numbers might also give students a sense of progression by visualising improvements (LD5), which could have a motivating effect (FB5).

CQ5 (Provide strategies to help improve the quality of the code) makes sure that students do not only learn to identify code quality issues, but also learn how these issues could be solved, closing the gap between their current performance and the desired performance (FB6). Like the examples of high quality code, the strategies are also additional information beside the feedback that might be useful to students (DB3 and LD3).

CQ6 (Provide general information about code quality and software metrics in addition to the feedback) aims to provide context for the feedback and information the students receive (FB3). This is another example of additional information provided to students (DB3 and LD3).

CQ7 (Ensure that feedback is easy to access, but not intrusive) aims to ensure that the feedback integrates well into the students' programming workflow (LD6). Additionally, it aims to ensure a good balance between providing enough information while not being overwhelming (DB6).

Finally, CQ8 (Allow the feedback to be easily shared) helps students and teachers explore the feedback in a collaborative manner (FB4 and LD7). By also giving the teachers insight into this shared data, they might be able to use this information to see if there are categories of issues that students struggle with. This might help shape teaching (FB7).

6.3. Answering RQ2: How to provide formative feedback on code quality

Based on the focus group sessions, eight initial design principles for formative feedback on code quality based on software metrics have been defined. The principles defined in this chapter are linked to meta design principles. The design principles defined in this chapter are shown in Table 6.3.

Code	*Principle*
CQ1	Link the feedback to the source code
CQ2	Provide examples of high quality code
CQ3	Provide the metric data with the feedback
CQ4	Provide comparisons of the metric results to both previous results and benchmark results
CQ5	Provide strategies to help improve the quality of the code
CQ6	Provide general information about code quality and software metrics in addition to the feedback
CQ7	Ensure that feedback is easy to access, but not intrusive
CQ8	Allow the feedback to be easily shared

Table 6.3: Software metric feedback design principles

6.4. Summary

The principles defined in this chapter could be used to guide the implementation of automated systems for providing effective code quality feedback. In the next chapter, a first iteration of such an implementation and evaluation of these principles manifested by a prototype for formative feedback on the quality of students' code is done.

7
Instantiating and evaluating the design principles for code quality feedback

To evaluate the design principles defined in chapter 6, a prototype implementing these principles has been built. We have used this prototype to evaluate the feedback in a number of observations.

7.1. Prototype

This section gives an overview of the design principles of the previous chapter as implemented in the prototype. However, it is noteworthy that design principle CQ8 (Allow the feedback to be easily shared) has not been implemented into the prototype. This was an explicit choice. For the initially planned field experiment, the participants would be fully anonymous. Sharing data would negate the anonymity of the participants. The evaluations described in this chapter are done in a one-on-one setting. This makes it difficult to test CQ8 as well. For a detailed and more technical description of the design of the prototype, see Appendix E.

The prototype is a web-based application combined with either an IDE extension or a command-line (CLI) client. The IDE extension or CLI clients are used to interact with the API provided by the prototype. This API provides endpoints for uploading a project for feedback and checking the processing status of a project. When a project uploaded by a student has been processed, it is possible to view the feedback in a browser. This method of using IDE extensions in combination with a web application implement design principle CQ7 (Ensure that feedback is easy to access, but not intrusive). The integration of one of the IDE plugins (Visual Studio) is shown in Figure 7.1.

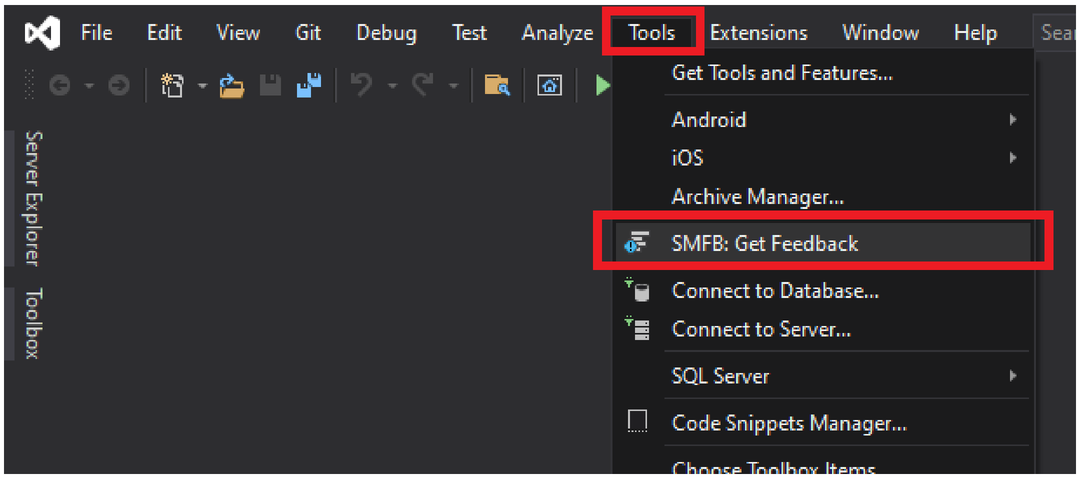
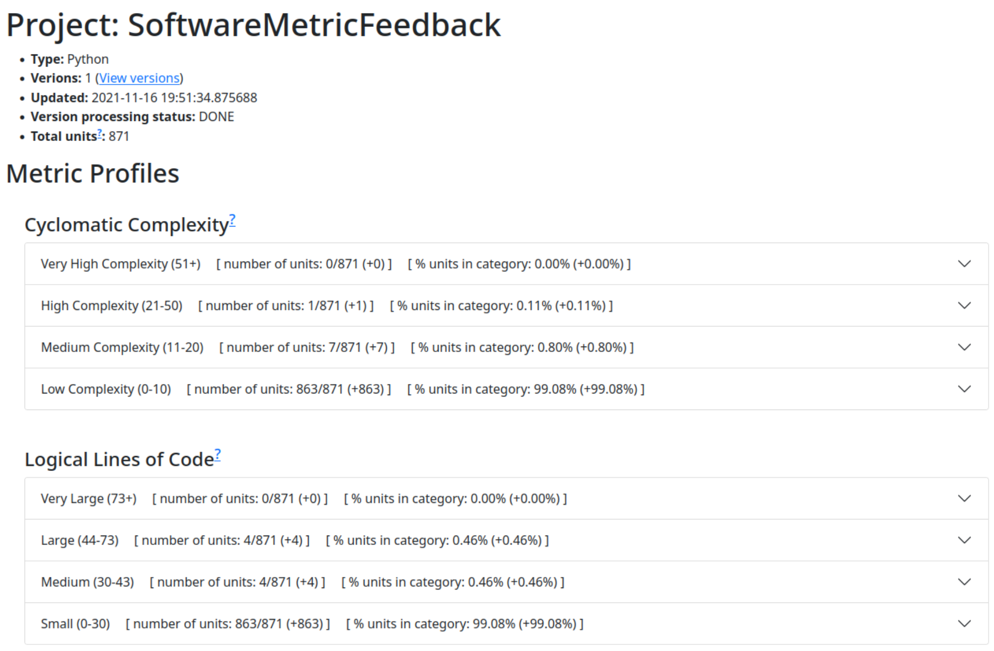


Figure 7.1: Visual Studio plugin

The feedback is shown on two levels. The first level is the project overview (see Figure 7.2). In this overview, the students get an overview of the units (i.e. the smallest measurable unit of code, such as a method) in their project. These units are categorized by metric and by a number of benchmark values for each metric. The benchmark values for the cyclomatic complexity have been derived from the SIG maintainability model by Heitlager et al. [30]. As the SIG maintainability model does not define benchmark values for unit size, the benchmark values for the logical lines of code have been derived from research by Alves et al. [3]. The calculated metric values have been tested for correctness by analyzing a number of open-source Python and C# projects with the prototype. For a number of functions or methods from these projects, the metric results have been manually calculated. These manually calculated results have been compared to the results given by the prototype.



Project: SoftwareMetricFeedback

- **Type:** Python
- **Verions:** 1 (View versions)
- **Updated:** 2021-11-16 19:51:34.875688
- **Version processing status:** DONE
- **Total units**?: 871

Metric Profiles

Cyclomatic Complexity?

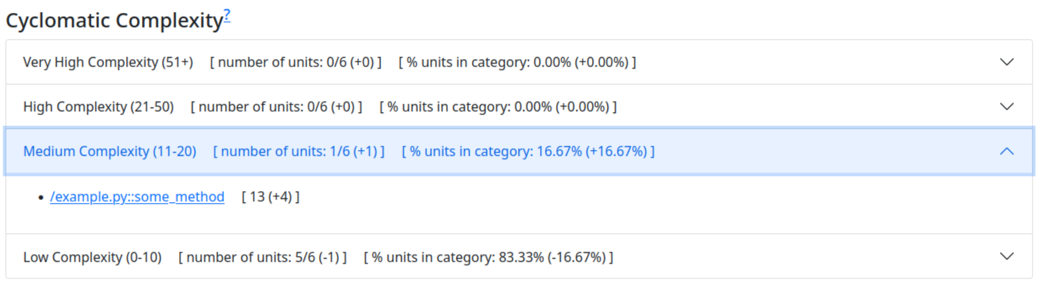
Very High Complexity (51+) [number of units: 0/871 (+0)] [% units in category: 0.00% (+0.00%)]
High Complexity (21-50) [number of units: 1/871 (+1)] [% units in category: 0.11% (+0.11%)]
Medium Complexity (11-20) [number of units: 7/871 (+7)] [% units in category: 0.80% (+0.80%)]
Low Complexity (0-10) [number of units: 863/871 (+863)] [% units in category: 99.08% (+99.08%)]

Logical Lines of Code?

Very Large (73+) [number of units: 0/871 (+0)] [% units in category: 0.00% (+0.00%)]
Large (44-73) [number of units: 4/871 (+4)] [% units in category: 0.46% (+0.46%)]
Medium (30-43) [number of units: 4/871 (+4)] [% units in category: 0.46% (+0.46%)]
Small (0-30) [number of units: 863/871 (+863)] [% units in category: 99.08% (+99.08%)]

Figure 7.2: Project overview

In addition to the benchmark values, the prototype also provides a comparison between the current version and the previous version. For example the `some_method` unit in Figure 7.3 shows that it has increased in cyclomatic complexity by 4. Additionally, it has moved from the low complexity category to the medium complexity category. This is shown both as an absolute value and as a percentage of the total units. The benchmark values and the comparisons together implement design principle CQ4 (Provide comparisons of the metric results to both previous results and benchmark results). To provide the comparisons, the metric data itself is also shown, implementing design principle CQ3 (Provide the metric data with the feedback).


Cyclomatic Complexity?

Very High Complexity (51+) [number of units: 0/6 (+0)] [% units in category: 0.00% (+0.00%)]

High Complexity (21-50) [number of units: 0/6 (+0)] [% units in category: 0.00% (+0.00%)]

Medium Complexity (11-20) [number of units: 1/6 (+1)] [% units in category: 16.67% (+16.67%)]

- /example.py::some_method [13 (+4)]

Low Complexity (0-10) [number of units: 5/6 (-1)] [% units in category: 83.33% (-16.67%)]

Figure 7.3: Comparison to previous version

In addition to the metric categories, it is also possible to browse to a source code file using a file tree overview of the project. This is shown in Figure 7.4.

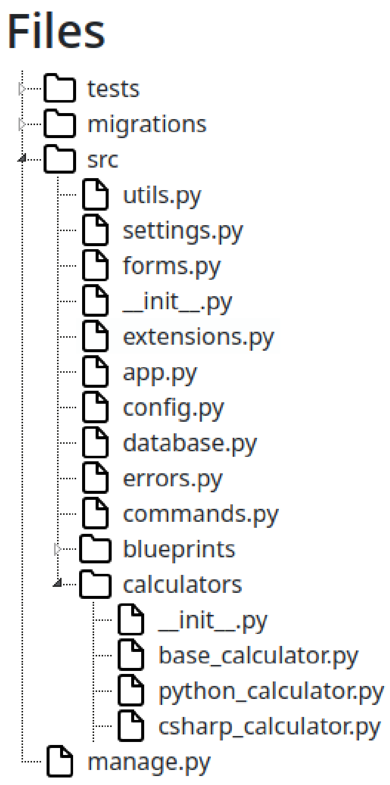
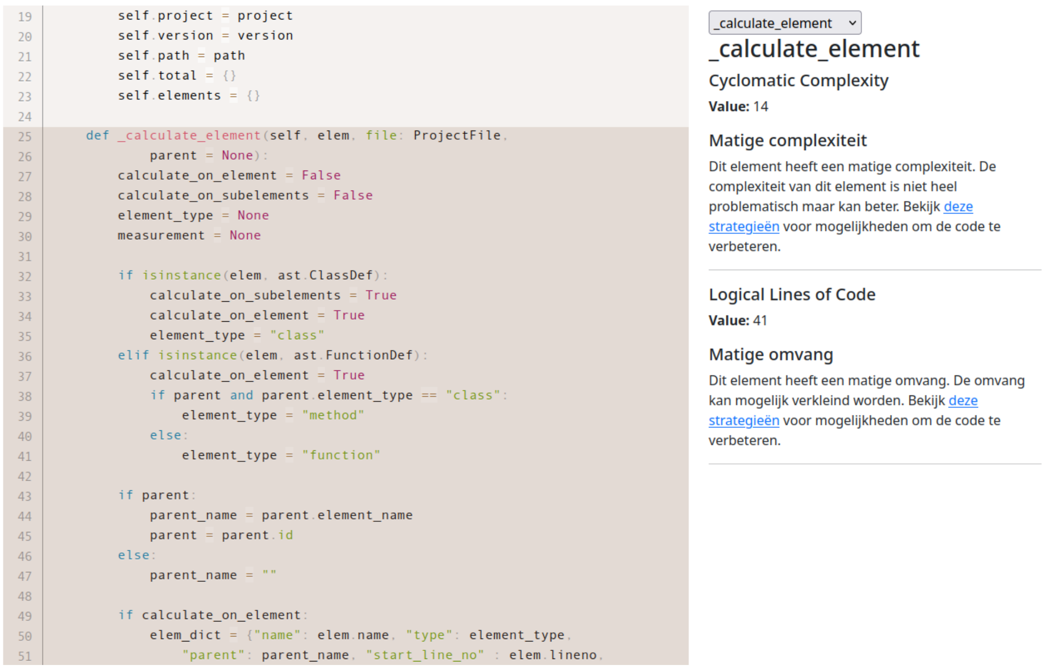


Figure 7.4: File tree overview

In various places in the prototype, question mark links are shown (e.g. next to the cyclomatic complexity and logical lines of code headers). Clicking on one of these question marks leads the user to an explanation of the related concept. For all information pages that show code, the code is given in all programming languages that are taught to students (in this case Python and C#). This information implements design principle CQ6 (Provide general information about code quality and software metrics in addition to the feedback).

By clicking on a unit in the metric categories or by clicking on a file in the file tree, it is possible to navigate to more detailed feedback for a specific file and unit. The feedback is always shown next to the code and the part of the code that is relevant for the given feedback is highlighted. This implements design principle CQ1 (Link the feedback to the source code). An example is shown in Figure 7.5. In this example, the source code is shown on the left, and the feedback for the selected unit is shown to the right. Additionally, the feedback provides a link with information about how to solve code quality issues and examples of good code. These strategies and code examples implement design principles CQ5 (Provide strategies to help improve the quality of the code) and CQ2 (Provide examples of high quality code). An example of one strategy, the single responsiblility principle, is shown in Figure 7.6.



```
        self.project = project
        self.version = version
        self.path = path
        self.total = {}
        self.elements = {}

    def _calculate_element(self, elem, file: ProjectFile,
            parent = None):
        calculate_on_element = False
        calculate_on_subelements = False
        element_type = None
        measurement = None

        if isinstance(elem, ast.ClassDef):
            calculate_on_subelements = True
            calculate_on_element = True
            element_type = "class"
        elif isinstance(elem, ast.FunctionDef):
            calculate_on_element = True
            if parent and parent.element_type == "class":
                element_type = "method"
            else:
                element_type = "function"

        if parent:
            parent_name = parent.element_name
            parent = parent.id
        else:
            parent_name = ""

        if calculate_on_element:
            elem_dict = {"name": elem.name, "type": element_type,
                "parent": parent_name, "start_line_no" : elem.lineno,
```

_calculate_element

Cyclomatic Complexity

Value: 14

Matige complexiteit

Dit element heeft een matige complexiteit. De complexiteit van dit element is niet heel problematisch maar kan beter. Bekijk deze strategieën voor mogelijkheden om de code te verbeteren.

Logical Lines of Code

Value: 41

Matige omvang

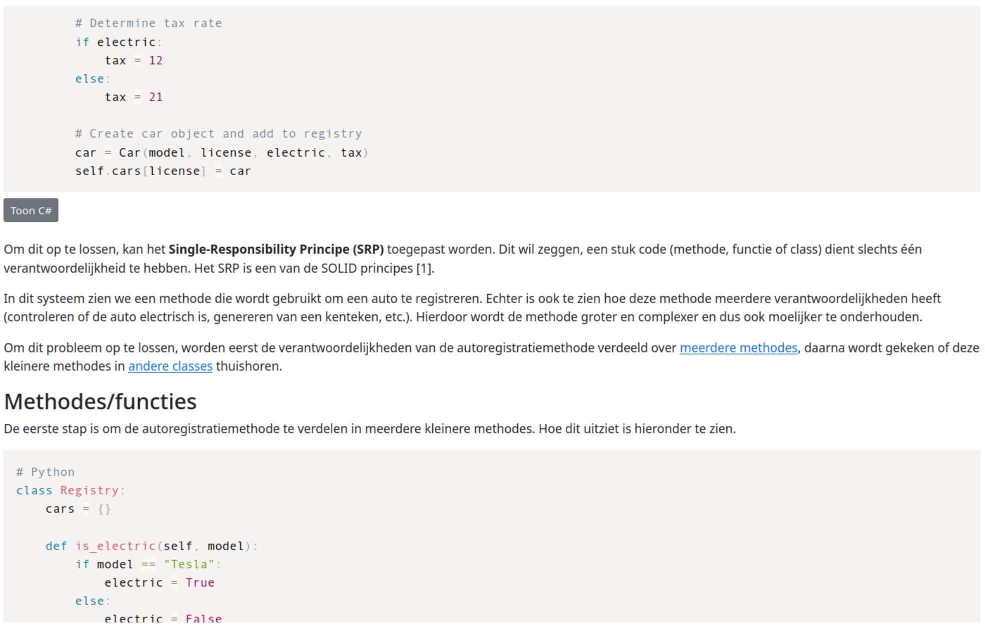
Dit element heeft een matige omvang. De omvang kan mogelijk verkleind worden. Bekijk deze strategieën voor mogelijkheden om de code te verbeteren.

Figure 7.5: File and unit level feedback

7.2. Evaluations

To evaluate the design principles and their implementation into the prototype, a number of evaluations have been done. Five evaluations with students have been done to asses the usefulness of the principles during the students' programming assignments. Additionally, an evaluations has been done with a teacher. For a detailed description of how the evaluations were done, see Section 4.4.3

Overall, the students found the feedback generated by the application useful. All of the students also noted that they would like to use subsequent versions of the tool used during the observation.



```
        # Determine tax rate
        if electric:
            tax = 12
        else:
            tax = 21

        # Create car object and add to registry
        car = Car(model, license, electric, tax)
        self.cars[license] = car
```

Om dit op te lossen, kan het **Single-Responsibility Principe (SRP)** toegepast worden. Dit wil zeggen, een stuk code (methode, functie of class) dient slechts één verantwoordelijkheid te hebben. Het SRP is een van de SOLID principes [1].

In dit systeem zien we een methode die wordt gebruikt om een auto te registreren. Echter is ook te zien hoe deze methode meerdere verantwoordelijkheden heeft (controleren of de auto electrisch is, genereren van een kenteken, etc.). Hierdoor wordt de methode groter en complexer en dus ook moelijker te onderhouden.

Om dit probleem op te lossen, worden eerst de verantwoordelijkheden van de autoregistratiemethode verdeeld over meerdere methodes, daarna wordt gekeken of deze kleinere methodes in andere classes thuishoren.

Methodes/functies

De eerste stap is om de autoregistratiemethode te verdelen in meerdere kleinere methodes. Hoe dit uitziet is hieronder te zien.

```
# Python
class Registry:
    cars = {}

    def is_electric(self, model):
        if model == "Tesla":
            electric = True
        else:
            electric = False
```

Figure 7.6: Single responsibility principle as a strategy for improving code

7.2.1. CQ1: Link the feedback to the source code

Overall, the students found the link between the feedback given by the prototype and the parts of their code that the feedback was applicable to clear. One student said:

> "*I found it very clear. When I clicked on the name of a function then I saw the highlighted code right away.* " (Student 1)

The fact that the prototype pointed students to issues that could be fixed was useful to students during the programming sessions. Four out of the five students spent their time during the programming session fixing issues pointed out by the prototype. This was also confirmed in the evaluation with the teacher, who said:

> "*It is I think a very important principle, so that you can see what [in your code] could be improved.* " (Teacher)

However, in some cases, the students had trouble finding specific parts that they were interested in (e.g. a class they had just modified). For this, students would like different overviews of their project. For example, as one student mentioned:

> "*Maybe it would be useful to show the size [as calculated by the logical lines of code metric] in the file tree. If I want to look at a certain part of the code, for example, to see how complex it is.* " (Student 3)

Sometimes the participants were interested in for example the largest or most complex modules, other times they were interested in viewing the changes per file or only in the files that had been changed. It would be useful to provide different views of the feedback to serve different goals. These changes would make it easier to find the areas of the code that need attention. However, all students mentioned that they found the current overview (by metric and category) useful as well. One student mentioned that he would find it useful to receive feedback on a more detailed level than a single unit (procedure or method).

7.2.2. CQ2: Provide examples of high quality code

The students found the code examples provided with the feedback to be helpful. As one student said:

> "*... it [code examples] is just the the easiest way to transfer information about coding.* " (Student 1)

It should be noted however that two students did not click through to the code examples during the programming session itself, but only after the session had ended. This suggests that the examples (or the link to the examples) should be more prominent. One student noted that he would have liked to see examples from applications more similar to the one he was working on, as this would have made it easier to apply the ideas from the examples to his own code.

Another suggestion put forward during the evaluation with the teacher is that it might be useful to show bad examples and an improved example side-by-side. This might be even further improved by showing visually how code has changed between the two versions of the code. For example, when splitting a large method into multiple smaller methods, arrows could be used to show which piece of code was moved to separate methods. Care should be taken however that the visual elements used to highlight the changes, highlight the correct aspects. The visual elements should not be to cluttered. For example, a Git diff style syntax might highlight many changes in the source code that are not directly relevant. This suggestion might make it useful to combine this principle and principle CQ5.

7.2.3. CQ3: Provide the metric data with the feedback

All students found the logical lines of code metric to be clear and useful, however, some students had trouble understanding the cyclomatic complexity (CC) metric. When given further explanation on the CC metric after the observation, the students who did not understand CC at first, did understand it then. This suggests that the explanation of the CC metric was not sufficient for all students. All students found that the metric data helped them identify areas of their code that could be improved. One student also mentioned that understanding how the metrics are calculated would be useful to know for future projects. The student said:

> "*I think it gives a good guideline when working on a project in the future.*
> " (Student 4)

The explanation of the software metrics could be improved, especially for the cyclomatic complexity. Additionally, one of the students suggested adding some kind of code duplication metric (this was one of the highest scoring metrics, next to lines of code and CC in Chapter 5). The metric categories (e.g. medium complexity, high complexity, small, large, etc.) could also be explained more clearly.

7.2.4. CQ4: Provide comparisons of the metric results to both previous results and benchmark results

In addition to the metric data itself, all students thought that seeing improvements in the metrics of their code was useful and had a motivating effect. Student 2 mentioned the motivating effect of seeing the improvements in the metrics of their code right after ending the programming session (before the after interview had started). Another student had a similar experience, saying:

> "*I find it very useful. It gives you satisfaction that it shows you that you have delivered good work... that you can see that it has improved.* " (Student 5)

However, it should be noted that one student did not notice the comparison between the current version of their code and the previous version of their code at first (partially due to an unusual project structure which resulted in a lot of library files being included in the feedback). This aspect could be improved by providing students a view that shows changed files. Some of the participants also noted that a more visual indication of comparisons between different versions of a project would better highlight changes. One student suggested providing a Git diff style syntax in the source code to highlight changes between versions. Another student suggested color coding the changes (e.g. make a decrease in complexity green and an increase in complexity red). The student also suggested combining the color coding of the changes with gamification elements. Another way it of making the changes more visible would be providing a seperate overview of units that have changed compared to the previous version.

Beside the comparisons on a function or method level, one student mentioned he would have liked to have scores for the entire project, a file, or a class, saying:

> "*Maybe you could show it [the metrics] per script [code files in Unity] so that you get some kind of score for the entire script.* " (Student 3)

Finally, students would have liked a more detailed explanation of how the benchmark values (e.g. lower complexity, medium complexity, etc.) had been derived. This opinion was shared by the teacher, who also thought further explanation of the benchmarks would be useful for students.

7.2.5. CQ5: Provide strategies to help improve the quality of the code

All students found the strategies for improving their code useful. Four out of five students used the programming session to apply the strategies for reducing code quality issues to improve their own code. For example, one student who applied the single responsibility principle to split a large Python function with multiple responsibilities into several smaller functions said:

> "*One thing that really stuck with me was that every function should have one responsibility instead of multiple.* " (Student 2)

However, sometimes the students found it hard to apply the strategies to their own code. One student mentioned that providing more examples of how to apply the different strategies might help with this aspect. Students also said that they would like more examples and strategies of different types of applications so that they could look for examples that applied the strategy applications of the type that they were working on. For example, one student working on GUI elements of a mobile application was not sure how to apply the strategies shown in the examples.

The teacher thought it would be interesting to also look at higher level elements of a code base that do not only apply to a single unit, but also to for example a number of components. Metrics that have been excluded in chapter 5 because they did not really allow for root cause analyis, for example metrics that measure coupling (e.g. the depth of inheritance) might be suitable for this type of feedback.

7.2.6. CQ6: Provide general information about code quality and software metrics in addition to the feedback

Three students found the general information about code quality provided in the prototype to be useful, while two students found the general information to be of limited use. One student that thought the information was useful said:

> "*It is definitely something that I find useful. Many concepts explained in it [the general information] are things that I would like to apply to my own projects as well.* " (Student 4)

However, one of the students that did not think the general information is useful said:

> "*Honestly, I think it only has limited usefulness. It is mostly a case of in one ear and out the other.* " (Student 1)

Additionally, not all students were able to find the general information about code quality during the programming session, which suggests that this information could be better integrated into the feedback. This might also have contributed to the perceived lack of usefulness by student 1.

The teacher thought the general information about code quality and software quality was very important. In his experience in the industry, most of the work he had done had been maintenance of existing systems. This is consistent with the observations made by Bennett [7] and Glass [27].

7.2.7. CQ7: Ensure that feedback is easy to access, but not intrusive

All students found the integration of the feedback process into their normal programming workflow to be good. This includes one student using PyCharm, for which no IDE extension was available. He used the CLI client to upload his code. Additionally, in one case, the Visual Studio extension did not work. Here the CLI client was also used instead. In the cases where the IDE extension worked, the students were satisfied with how easy it was to request feedback. One student described it as:

> "*Just press the button and you immediately get an overview, so that is great. You don't have to manually browse to a website and manually upload a zip file. Just press a single button and you are done.* " (Student 1)

The integration of the tool could be improved by supporting more IDEs. Additionally, the installation process could be made easier, for example by creating an installer. Finally, the feedback and information could be shown directly in the IDE. This would greatly improve the integration and also clarify the link between the code and the feedback.

7.2.8. General Areas of Improvement

In addition to the design principles, there are a number of improvements that could be made to the prototype overall.

A number of improvements could be made on the aspect of the navigation to and the availability of the existing information. For example, always showing the related links for navigation would make it easier to find additional information when needed. In addition, it is important to always show the relevant information on each view. For example, ensure that the feedback also shows the benchmark values (even if it receives less focus than on the overview page). In general the user experience as a whole could be improved.

During the evaluations, a number of technical issues occured. A number of issues had to do with the Visual Studio extension, which did not always work correctly. Additionally, the prototype had some small issues with some projects used during the evaluation. For example, the inclusion of library files in the case of Unity projects. This made some parts of the prototype confusing for the students.

Finally, students expressed the desire to fine-tune their feedback requests. This includes aspects such as requesting feedback on a single file or explicitly excluding files from feedback. One example of this is to give students the option to use the gitignore file to ignore certain files or folders.

7.3. Answering RQ3: Affects of formative code quality feedback

Overall, the students found the prototype and the design principles implemented therein helpful and motivating. It helped them fix issues in their code. The results described in this chapter also suggest that the feedback could increase code quality in future programming assignments. The teacher thought that the information and feedback was useful for students and important for students to learn.

However, it is also clear that further refinement of these principles and their implementation is possible. In addition, there might also be other design principles that could further guide the design and development of tools for formative feedback on code quality. Also note that CQ8 (Allow the feedback to be easily shared) has not been evaluated, since there was limited opportunity for sharing feedback in the the evaluation.

8 Discussion

8.1. Reflection on design principles

In this research, we have attempted to find a method for automatically generating formative feedback on code quality for open ended assignments. To do this, a number of design principles have been defined (see chapter 6) and evaluated (see chapter 7). In this section we reflect on these design principles and link them to the literature.

Feedback needs to be detailed enough to be useful to students [26][28]. Design principles CQ2 (Provide examples of high quality code), CQ3 (Provide the metric data with the feedback), CQ5 (Provide strategies to help improve the quality of the code), and CQ6 (Provide general information about code quality and software metrics in addition to the feedback) aim to provide this detail. Overall the students found the information provided to be useful. However, there is also room for further refinement of these principles. Especially CQ6 needs further research. For this principle, some students did not see the relevance to their work. Other students found some of the information difficult to understand or to apply.

The main goal of formative feedback is to identify gaps between the students' current level of performance and the desired level of performance [28][70]. Additionally, students should learn how to assess their own performance [46] [51]. By linking the feedback directly to the source code (QC1, Link the feedback to the source code), providing students with concrete metrics data (CQ3, Provide the metric data with the feedback), and finally comparing that data with benchmark values and previous results (CQ4, Provide comparisons of the metric results to both previous results and benchmark results), students can learn where they stand in comparison to the desired level of performance. They are also guided to what areas of their work could be improved. In addition, students mentioned feeling motivated by the guidance on how to improve, which is an important factor in learning outcome [42]. The motivational aspect could be further improved by adding gamification elements.

Any digital learning tools should be integrated into the workflow of the learner and should be easy to use [15][74]. Design principle CQ7 (Ensure that feedback is easy to access, but not intrusive) aims to give guidelines for this integration within the context of programming education. In this research, this is done by making use of the plugin systems provided by different IDEs. These IDE plugins provide an easy method for requesting and accessing the feedback without breaking the normal programming workflow. However, it is important to not show too much information at once and clutter the IDEs' interface. While the principle as implemented in the prototype is a good start, there is also room for further refinement and additions (e.g. making better use of the plugin features provided by IDEs).

8.2. Reflection on research method

In addition to reflecting on the results, we want to reflect on the research method as well. These insights could be useful when iterating further on the design principles defined in this research.

Participants in both focus group sessions had a tendency to slip into discussion about how certain things could be implemented. For example, while the students were clear about that they would like to be able to compare their code to benchmark values, they did not know how this could be implemented. In this case, the students were not aware of efforts to generate metric benchmark values, such as the research by Alves et al. [3]. The discussions on implementation did at times distract from the main topic of how the feedback should be generated and presented, regardless of how it could be implemented. It might also be the case that some participants did not mention things they would have liked to see in the feedback, but thought were impossible. In the future it might help to instruct participants to not consider how something could be implemented.

During two of the evaluations, there were technical issues with the prototype. In both cases this had to do with unusual project structures. In one case a student uploaded a Unity project (i.e. C# based). Normally when using C#, the libraries are kept outside of the project directory by NuGet (the C# package manager). However, with Unity, the library files are kept inside the project directory. This lead to a large number of library files being included in the feedback, which made the students' own files harder to find. In the second case, the student had multiple solution files (the files used by Visual Studio to keep track of the files and library of a project). These multiple solution files confused the C# metric extraction tool, resulting in incorrect values used to generate feedback. These issues might have been avoided by using more, and more varied, test cases while developing the prototype.

8.3. Threats to validity and limitations

This research is only done in the context of a single program, this might lead to a solution that is only viable for this specific context. One such example is the fact that within the HBO-ICT program of Zuyd University of Applied Science, students learn Python and C#. In 3rd and 4th year courses, students are allowed to choose their own preferred programming language in most cases. This means that at least Python and C# both had to be supported by the prototype. While C# is a purely object oriented (OO) language, Python is a multi-paradigm language. As a result, some metrics and feedback which would likely be useful to students programming using only the OO paradigm could not be well supported in this case.

Due to the Covid19 pandemic, the HBO-ICT program of Zuyd University of Applied Science has not been operating in the usual manner. Courses that have been developed with in-person education in mind, have been taught online. It is at the time of writing not clear what the affects of the pandemic are on the education of the population used within this research. This same issue applies to the research itself as well. For example, the focus group sessions could be done in person. However, when the evaluations started, a lockdown was in place. This meant that the evaluations had to be done online.

Only a small number of participants have been involved in the formulation of the initial principles (10 participants) and the evaluation of these principles (6 participants). Initially, a larger evaluation had been planned (and the prototype was designed with this in mind), but not enough participants could be found. Additionally, all these participants are either enrolled at the same institution and in the same program or are employees of the same institution.

In addition to the small number of participants, participation in the research was entirely voluntary. This might have resulted in response bias. The students that participate in the research might not be representative of the typical student. In addition, since the researcher is an active teacher at the program the research was done at, many participants had prior experiences with the researcher and with each other. This might have also influenced the results. To mitigate this, the participants have been reminded to answer as honestly as possible and have been informed that all personally identifiable data is removed from the results.

The design based research approach used in this research is usually done over a number of iterations. However, in this research only a single iteration is done. Usually, further iterations are used to refine both the design principles and the solution that implements these principles. Since only a single iteration is done in this research, this further refinement has not been done.

Finally, the design principles formulated in chapter 6 have been evaluated with a prototype based on these principles. However, some of the results gathered during the evaluations indicate issues with or improvements that could be made to the implementation of the principles, not necessarily the principles themselves. This might partially be due to the fact that only a single iteration of the design research cycle has been done. These issues could be resolved in further iterations on the prototype. Some of these issues with the implementation of the principles could lead to further refinement of the principles or to new principles entirely.

9 CONCLUSION

The aim of this research is to help students with their understanding of code quality and to empower them to write higher quality code. However, with the time constraints that teachers often face, it is difficult to provide enough feedback with sufficient detail. This is why we looked at automated methods for providing code quality feedback using software metrics. To do this we defined the following research question: **"How can software metrics be used to facilitate formative assessment on code quality in project-based, open-ended programming assignments?"**. To answer the main research question, we first answered three sub-questions.

For the first research question (RQ1), *"Which software metrics can be used to generate code quality feedback?"*, we have done a literature review to get a suite of suitable software metrics. This resulted in three suitable metrics for automatically generating code quality feedback. These metrics are (extended) cyclomatic complexity, lines of code, and duplication. These three metrics have been used in this research as examples of how software metrics can be used to generate feedback.

For the second research question (RQ2), *"How can software metrics be used for provide formative feedback on code quality?"*, two focus group sessions have been done. During these sessions, students and teachers dicussed what aspects are important for the generated feedback, i.e. what do students need to gain a better understanding of code quality and improve their code. This resulted in a set of initial design principles. These design principles are shown in Table 9.1.

Code	*Principle*
CQ1	Link the feedback to the source code
CQ2	Provide examples of high quality code
CQ3	Provide the metric data with the feedback
CQ4	Provide comparisons of the metric results to both previous results and benchmark results
CQ5	Provide strategies to help improve the quality of the code
CQ6	Provide general information about code quality and software metrics in addition to the feedback
CQ7	Ensure that feedback is easy to access, but not intrusive
CQ8	Allow the feedback to be easily shared

Table 9.1: Software metric feedback design principles

The aim of the third research question (RQ3), *"How does software metric based formative feedback affect code quality knowledge among students?"*, is to evaluate the design principles formulated by the second research question. In addition, it aims to provide input for further changes or refinement of the design principles. For the evaluation, the principles have been implemented in a prototype. Several students were then asked to use the prototype while working on their programming assignment. This programming session was observed and the students were interviewed before and after the session. In addition, a teacher was interviewed as well. From the evaluations it is clear that students thought the feedback provided by the prototype was useful. However, it is also clear that further refinements of the principles and their implementation is needed.

By answering the three sub-questions, the main research question can also be answered. By combining the software metrics found in RQ1 with the design principles formulated in RQ2, we have provide an automated method for formative feedback on code quality. These design principles and the approach for automated formative code quality feedback defined in this study have been instantiated in a prototype. The evaluations of RQ3 show that while further refinement is needed, the approach outlined by the intial set of design principles defined in this research is promising.

9.1. Future work

While this research gives an initial set of design principles for formative feedback on code quality based on software metrics, further iterations could improve on the design principles and their implementation. For example, design principle CQ6 (Provide general information about code quality and software metrics in addition to the feedback) needs further refinement in order to determine how the general information should be intergrated into the feedback. In addition to improvements, further research could yield additional design principles.

In this research, only a small scale evaluation has been done. Further research is needed to verify the design principles at a larger scale and in different contexts than the one used in this research Additionally, design principle [CQ8] (Allow the feedback to be easily shared) has not yet been evaluated.

Another interesting approach is seeing which other metrics (such as the metrics with a score of 17, listed in Appendix D) could be be used to generate feedback. For example, one area that is lacking from the metrics currently implemented in the prototype is a coupling metric. For further research, it might be interesting to look at which metrics are suitable for specific types of programming assignments, such as assignments using only object-oriented languages or assignments within a specific programming context (e.g. mobile application development).

During the both the focus group sessions in chapter 6 and the evaluations in chapter 7, participants mentioned that they would find it interesting to compare the metrics of their own code to that of peers. A future line of research could be to extend the approach described in this study with comparisons to peer results.

Finally, it might be interesting to look at these design principles from the perspective of teachers. As mentioned in the related work (see chapter 2), how the feedback can be useful to teachers should be considered. Initially a fourth sub-question for the impact of software metric based code quality feedback on teachers had been considered. However, to keep the focus on the student perspective, and due to ethical concerns with fair grading (see Section 4.5), this aspect has explicitly not been included in this research.

BIBLIOGRAPHY

[1] AHREN, T. Using online annotation software to provide timely feedback in an introductory programming course. In *Proceedings Frontiers in Education 35th Annual Conference* (2005), IEEE, pp. T2H–1. 12

[2] ALA-MUTKA, K., UIMONEN, T., AND JARVINEN, H.-M. Supporting students in c++ programming courses with automatic program style assessment. *Journal of Information Technology Education: Research 3*, 1 (2004), 245–262. 12

[3] ALVES, T. L., YPMA, C., AND VISSER, J. Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance* (Sep. 2010), pp. 1–10. 39, 48

[4] BENNEDSEN, J., AND CASPERSEN, M. E. Failure rates in introductory programming. *AcM SIGcSE Bulletin 39*, 2 (2007), 32–36. 7

[5] BENNEDSEN, J., AND CASPERSEN, M. E. Failure rates in introductory programming: 12 years later. *ACM Inroads 10*, 2 (2019), 30–36. 7

[6] BENNETT, K., AND RAJLICH, V. The staged model of the software lifecycle: A new perspective on software evolution. Tech. rep., Citeseer, 2002. iv, 4

[7] BENNETT, K. H., AND RAJLICH, V. T. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (2000), pp. 73–87. 1, 3, 4, 15, 45

[8] BIEHL, J. T., CZERWINSKI, M., SMITH, G., AND ROBERTSON, G. G. Fastdash: a visual dashboard for fostering awareness in software teams. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (2007), pp. 1313–1322. 13

[9] BIJLSMA, D., FERREIRA, M. A., LUIJTEN, B., AND VISSER, J. Faster issue resolution with higher technical quality of software. *Software quality journal 20*, 2 (2012), 265–285. 3, 17

[10] BODILY, R., IKAHIHIFO, T. K., MACKLEY, B., AND GRAHAM, C. R. The design, development, and implementation of student-facing learning analytics dashboards. *Journal of Computing in Higher Education 30*, 3 (2018), 572–598. 13

[11] BÖRSTLER, J., STÖRRLE, H., TOLL, D., VAN ASSEMA, J., DURAN, R., HOOSHANGI, S., JEURING, J., KEUNING, H., KLEINER, C., AND MACKELLAR, B. " i know it when i see it" perceptions of code quality: Iticse'17 working group report. In *Proceedings of the 2017 ITiCSE Conference on Working Group Reports* (2018), pp. 70–85. 8, 17

[12] BREUKER, D. M., DERRIKS, J., AND BRUNEKREEF, J. Measuring static quality of student code. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education* (2011), pp. 13–17. 8, 17

[13] Brown, C., Pastel, R., Siever, B., and Earnest, J. Jug: a junit generation, time complexity analysis and reporting tool to streamline grading. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education* (2012), pp. 99–104. 12

[14] Brown, E., and Glover, C. Evaluating written feedback. In *Innovative assessment in higher education*. Routledge, 2006, pp. 101–111. 10, 11

[15] Charleer, S., Klerkx, J., Duval, E., De Laet, T., and Verbert, K. Creating effective learning analytics dashboards: Lessons learnt. In *European Conference on Technology Enhanced Learning* (2016), Springer, pp. 42–56. 14, 15, 16, 31, 35, 47, ix

[16] Coleman, D., Ash, D., Lowther, B., and Oman, P. Using metrics to evaluate software system maintainability. *Computer 27*, 8 (1994), 44–49. 6

[17] Cummins, S., Burd, L., and Hatch, A. Tag based feedback for programming courses. *ACM SIGCSE Bulletin 41*, 4 (2010), 62–65. 12

[18] De Raadt, M., Lai, D., and Watson, R. An evaluation of electronic individual peer assessment in an introductory programming course. In *Proceedings of the 7th Baltic Sea Conference on Computing Education Research (Koli Calling 2007)* (2008), vol. 88, Australian Computer Society Inc., pp. 53–64. 12

[19] Denton, P., Madden, J., Roberts, M., and Rowe, P. Students' response to traditional and computer-assisted formative feedback: A comparative case study. *British Journal of Educational Technology 39*, 3 (2008), 486–500. 9

[20] Dixson, D. D., and Worrell, F. C. Formative and summative assessment in the classroom. *Theory into practice 55*, 2 (2016), 153–159. 9

[21] Dotson, R. Goal setting to increase student academic performance. *Journal of School Administration Research and Development 1*, 1 (2016), 45–46. 10

[22] Douce, C., Livingstone, D., and Orwell, J. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC) 5*, 3 (2005), 4–es. 12

[23] Faber, J. M., Luyten, H., and Visscher, A. J. The effects of a digital formative assessment tool on mathematics achievement and student motivation: Results of a randomized experiment. *Computers & education 106* (2017), 83–96. 9

[24] Fenton, N., and Bieman, J. *Software metrics: a rigorous and practical approach*. CRC press, 2014. 5

[25] Few, S. *Information dashboard design: The effective visual communication of data*. O'Reilly Media, Inc., 2006. 13, 14, 16, 31, ix

[26] Gibbs, G., and Simpson, C. Conditions under which assessment supports students' learning. *Learning and teaching in higher education*, 1 (2005), 3–31. 11, 47

[27] Glass, R. L. Frequently forgotten fundamental facts about software engineering. *IEEE software 18*, 3 (2001), 112. 3, 15, 45

[28] Hattie, J., and Timperley, H. The power of feedback. *Review of educational research 77*, 1 (2007), 81–112. iv, 8, 9, 10, 11, 15, 17, 27, 47

[29] Heinemann, L., Hummel, B., and Steidl, D. Teamscale: Software quality control in real-time. In *Companion Proceedings of the 36th International Conference on Software Engineering* (2014), pp. 592–595. 13

[30] Heitlager, I., Kuipers, T., and Visser, J. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)* (2007), IEEE, pp. 30–39. iv, 6, 7, 27, 28, 30, 39

[31] Hepplestone, S., Holden, G., Irwin, B., Parkin, H. J., and Thorpe, L. Using technology to encourage student engagement with feedback: a literature review. *Research in Learning Technology 19*, 2 (2011). 9, 10

[32] Honglei, T., Wei, S., and Yanan, Z. The research on software metrics and software complexity metrics. In *2009 International Forum on Computer Science-Technology and Applications* (2009), vol. 1, IEEE, pp. 131–136. 5

[33] Hwang, G.-J., Liang, Z.-Y., and Wang, H.-Y. An online peer assessment-based programming approach to improving students' programming knowledge and skills. In *2016 International Conference on Educational Innovation through Technology (EITT)* (2016), IEEE, pp. 81–85. 12

[34] ISO/IEC. International standard-iso/iec 14764 ieee std 14764-2006 software engineering; software life cycle processes &; maintenance. 4

[35] Jackson, D., and Usher, M. Grading student programs using assyst. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education* (1997), pp. 335–339. 12

[36] Kali, Y. Collaborative knowledge building using the design principles database. *International Journal of Computer-Supported Collaborative Learning 1*, 2 (2006), 187–201. 19

[37] Keuning, H., Heeren, B., and Jeuring, J. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (2017), pp. 110–115. vi, 1, 7, 8, 15, 17, 30

[38] Kirk, D., Crow, T., Luxton-Reilly, A., and Tempero, E. On assuring learning about code quality. In *Proceedings of the Twenty-Second Australasian Computing Education Conference* (2020), pp. 86–94. 8, 17

[39] Koyya, P., Lee, Y., and Yang, J. Feedback for programming assignments using software-metrics and reference code. *International Scholarly Research Notices 2013* (2013). 12

[40] LAHTINEN, E., ALA-MUTKA, K., AND JÄRVINEN, H.-M. A study of the difficulties of novice programmers. *Acm sigcse bulletin 37*, 3 (2005), 14–18. 7

[41] LIENTZ, B. P., AND SWANSON, E. B. Problems in application software maintenance. *Communications of the ACM 24*, 11 (1981), 763–769. 4

[42] LIU, O. L., BRIDGEMAN, B., AND ADLER, R. M. Measuring learning outcomes in higher education: Motivation matters. *Educational Researcher 41*, 9 (2012), 352–362. 11, 47

[43] LOCKE, E. A., AND LATHAM, G. P. Goal setting: A motivational technique that works! *Organizational Dynamics* (1984). 10

[44] MCCABE, T. J. A complexity measure. *IEEE Transactions on software Engineering*, 4 (1976), 308–320. vi, 5, 30

[45] MCMILLAN, J. H. *Classroom assessment: Principles and practice that enhance student learning and motivation.* Pearson, 2017. iv, 9, 10

[46] MEGA, C., RONCONI, L., AND DE BENI, R. What makes a good student? how emotions, self-regulated learning, and motivation contribute to academic achievement. *Journal of educational psychology 106*, 1 (2014), 121. 10, 47

[47] MENGEL, S. A., AND YERRAMILLI, V. A case study of the static analysis of the quality of novice student programs. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education* (1999), pp. 78–82. 12

[48] MITTAL, M., AND SUREKA, A. Process mining software repositories from student projects in an undergraduate software engineering course. In *Companion Proceedings of the 36th International Conference on Software Engineering* (2014), pp. 344–353. 12

[49] MOELLER, A. J., THEILER, J. M., AND WU, C. Goal setting and student achievement: A longitudinal study. *The Modern Language Journal 96*, 2 (2012), 153–169. 10

[50] MOLENAAR, I., AND KNOOP-VAN CAMPEN, C. Teacher dashboards in practice: Usage and impact. In *European Conference on Technology Enhanced Learning* (2017), Springer, pp. 125–138. 13

[51] NICOL, D. J., AND MACFARLANE-DICK, D. Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. *Studies in higher education 31*, 2 (2006), 199–218. 10, 11, 15, 27, 28, 31, 32, 33, 47, ix

[52] OMAN, P., AND HAGEMEISTER, J. Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software 24*, 3 (1994), 251–266. 6

[53] PANADERO, E., JONSSON, A., AND STRIJBOS, J.-W. Scaffolding self-regulated learning through self-assessment and peer assessment: Guidelines for classroom implementation. In *Assessment for learning: Meeting the challenge of implementation* (2016), Springer, pp. 311–326. 10

[54] PANDAZO, K., SHOLLO, A., STARON, M., AND MEDING, W. Presenting software metrics indicators: a case study. In *Proceedings of the 20th international conference on Software Product and Process Measurement (MENSURA)* (2010), vol. 20. 13

[55] PETTIT, R., HOMER, J., GEE, R., MENGEL, S., AND STARBUCK, A. An empirical study of iterative improvement in programming assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (2015), pp. 410–415. 12

[56] PITEIRA, M., AND COSTA, C. Learning computer programming: study of difficulties in learning programming. In *Proceedings of the 2013 International Conference on Information Systems and Design of Communication* (2013), pp. 75–80. 7

[57] POULOS, A., AND MAHONY, M. J. Effectiveness of feedback: The students' perspective. *Assessment & Evaluation in Higher Education 33*, 2 (2008), 143–154. 10

[58] RAJLICH, V. Software evolution and maintenance. In *Future of Software Engineering Proceedings*. 2014, pp. 133–144. 1, 5, 17

[59] REEVES, T. C. Design research from a technology perspective. *Educational design research 1*, 3 (2006), 52–66. iv, 19, 20, 21

[60] ROBINS, A., ROUNTREE, J., AND ROUNTREE, N. Learning and teaching programming: A review and discussion. *Computer science education 13*, 2 (2003), 137–172. 7, 12

[61] SANDEE, J. J., AND AIVALOGLOU, E. Gitcanary: A tool for analyzing student contributions in group programming assignments. In *Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (2020), pp. 1–2. 12

[62] SEDRAKYAN, G., LEONY, D., MUÑOZ-MERINO, P. J., KLOOS, C. D., AND VERBERT, K. Evaluating student-facing learning dashboards of affective states. In *European Conference on Technology Enhanced Learning* (2017), Springer, pp. 224–237. 13

[63] SHARP, C., VAN ASSEMA, J., YU, B., ZIDANE, K., AND MALAN, D. J. An open-source, api-based framework for assessing the correctness of code in cs50. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (2020), pp. 487–492. 12

[64] SHULMAN, L. Knowledge and teaching: Foundations of the new reform. *Harvard educational review 57*, 1 (1987), 1–23. 11

[65] SHULMAN, L. S. Those who understand: Knowledge growth in teaching. *Educational researcher 15*, 2 (1986), 4–14. 11

[66] SITTHIWORACHART, J., AND JOY, M. Web-based peer assessment in learning computer programming. In *Proceedings 3rd IEEE International Conference on Advanced Technologies* (2003), IEEE, pp. 180–184. 12

[67] SNEED, H. M., AND BROSSLER, P. Critical success factors in software maintenance: a case study. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* (2003), IEEE, pp. 190–198. 1, 3, 5, 17

[68] Stegeman, M., Barendsen, E., and Smetsers, S. Designing a rubric for feedback on code quality in programming courses. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (2016), pp. 160–164. 7

[69] Suzuki, R. Poster: Interactive and collaborative source code annotation. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), vol. 2, IEEE, pp. 799–800. 12

[70] Taras, M. Assessment–summative and formative–some theoretical reflections. *British journal of educational studies 53*, 4 (2005), 466–478. 8, 9, 47

[71] van den Akker, J., Branch, R. M., Gustafson, K., Nieveen, N., and Plomp, T. *Design approaches and tools in education and training*. Springer Science & Business Media, 1999. 18, 19, 20

[72] Van der Schaaf, M., Baartman, L., Prins, F., Oosterbaan, A., and Schaap, H. Feedback dialogues that stimulate students' reflective thinking. *Scandinavian Journal of Educational Research 57*, 3 (2013), 227–245. 11

[73] Venables, A., and Haywood, L. Programming students need instant feedback! In *Proceedings of the fifth Australasian conference on Computing education-Volume 20* (2003), Citeseer, pp. 267–272. 12

[74] Vesin, B., Mangaroska, K., and Giannakos, M. Learning in smart environments: user-centered design and analytics of an adaptive learning system. *Smart Learning Environments 5*, 1 (2018), 1–21. 15, 47

[75] Vihavainen, A., Vikberg, T., Luukkainen, M., and Pärtel, M. Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education* (2013), pp. 117–122. 12

[76] Wang, X.-M., Hwang, G.-J., Liang, Z.-Y., and Wang, H.-Y. Enhancing students' computer programming performances, critical thinking awareness and attitudes towards programming: An online peer-assessment attempt. *Journal of Educational Technology & Society 20*, 4 (2017), 58–68. 12

[77] Watson, C., and Li, F. W. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education* (2014), pp. 39–44. 7

[78] Williams, J., and Kane, D. Assessment and feedback: Institutional experiences of student feedback, 1996 to 2007. *Higher Education Quarterly 63*, 3 (2009), 264–286. 11

[79] Wisniewski, B., Zierer, K., and Hattie, J. The power of feedback revisited: A meta-analysis of educational feedback research. *Frontiers in Psychology 10* (2020), 3087. 8, 10

[80] Xhakaj, F., Aleven, V., and McLaren, B. Effects of a teacher dashboard for an intelligent tutoring system on teacher knowledge, lesson planning, lessons and student learning. In *Data Driven Approaches in Digital Education* (2017), Springer International Publishing, pp. 315–329. 13

Appendices

A. SUMMARY OF (META-)DESIGN PRINCIPLES

This Appendix gives a summary of the relevant (meta-)design principles from the literature as described in the related work section (Section 2).

Table 2 gives an overview of the design principles for good formative feedback defined by Nicol and MacFarlane-Dick [51]. These design principles are described in detail in section 2.4.3.

Code	*Design principle*
FB1	Good feedback helps clarify what good performance is.
FB2	Good feedback facilitates the development of self-assessment (reflection) in learning.
FB3	Good feedback delivers high quality information to students about their learning.
FB4	Good feedback encourages teachers and peer dialog around learning.
FB5	Good feedback encourages positive motivational beliefs and self-esteem.
FB6	Good feedback provides opportunities to close the gap between current and desired performance.
FB7	Good feedback provides information to teachers that can be used to help shape teaching.

Table 2: Summary of design principles for formative feedback

Table 3 gives an overview of the design principles for good dashboard design defined by Few [25]. These design principles are described in detail in section 2.5.1.

Code	*Design principle*
DB1	Organize information to support its meaning and use.
DB2	Maintain consistency to enable quick and accurate interpretation.
DB3	Put supplementary information within reach.
DB4	Expose lower-level conditions.
DB5	Make the experience aesthetically pleasing.
DB6	Prevent excessive alerting.
DB7	Keep viewers in the loop.
DB8	Accommodate real-time monitoring.

Table 3: Summary of design principles for dashboards

Table 4 gives an overview of the design principles for good learning dashboard design defined by Charleer et al. [15]. These principles are described in detail in section 2.5.2.

Code	*Design principle*
LD1	Abstract the learning analytics data.
LD2	Provide access to the learner artifacts.
LD3	Augment the abstracted data.
LD4	Provide access to teacher and peer feedback.
LD5	Visualise the learner path.
LD6	Integrate the dashboard into the work-flow.
LD7	Facilitate collaborative exploration of (learning analytics) data.

Table 4: Summary of design principles for learning dashboards

B. cETO Documetation

B.1. cETO fast track application[1]

Workflow Finalized

Intended start date Thursday, July 1, 2021

Intended end date Monday, February 28, 2022

Principal investigator (supervisor in case of thesis) Rahimi E.

Researcher(s) performing the research (first name, surname, email) Eddy van den Aker, eddy.v.d.aker@gmail.com/eddy.vandenaker@zuyd.nl

Name nad email address of student who is going to conduct the research (if applicable) Eddy van den Aker, eddy.v.d.aker@gmail.com/eddy.vandenaker@zuyd.nl

In what context will the research be performed Master thesis

Does the WMO law apply No (no WMO check necessary)[2]

Provide a brief description of the study (max 250 words) (research questions, study design, procedure and main variables) The aim of this research is to create a set of design principles for automated formative feedback on code quality using software metrics in the context of advanced undergraduate programming courses. The main question is "How can software metrics be used to facilitate formative assessment on code quality in project-based, open-ended programming assignments?". (RQ1) Which software metrics can be used to generate code quality feedback? (RQ2) How can software metrics be used for formative feedback on code quality? (RQ3) How does software metric based formative feedback effect code quality knowledge among students? RQ1 will be answered by reviewing the literature. RQ2 will be answered using two focus group sessions. Session 1 will translate a number of general design principles for feedback/dashboards and the metrics selected in RQ1 into high level requirements for a programming education context. Session 2 will define a number of context specific design principles based on the general design principles, the metrics selected in RQ1, and the high level requirements from the first session. Based on RQ1 and RQ2, a prototype will be built using the design principles. RQ3 will be answered using a field-test with a quasi-experimental design with pre- and post-knowledge tests and without control group. The prototype gathers a number of anonymous usage statistics (times participant uses the prototype, submissions to the prototype, reviewing previous feedback). The goal of RQ3 is to validate the design principles formulated in RQ2.

[1] cETO has two different application processes, the fast track process and the normal process. The fast track is only available in certain cases. See https://www.ou.nl/documents/40554/4197127/III_04_cETO_flowchart_F.pdf/ef873406-4ebc-b648-1056-f5eb9c305d33?t=1649768928709 for an overview.

[2] The WMO law applies to medical studies or studies that requires participants to follow procedures or rules that may cause an infringement of the (physical or psychological) integrity of the potential participant. See https://ceto.ou.nl/WMO for details.

Participants: RQ2: 6-10 students, 2 teachers RQ3: students second-year web-development course and third-year AI course.

What research popluation falls under the study? (You may check multiple answers) General without disorder

Among which age category are you going to perform your research and which corresponding consent forms are you going to use? (You may check multiple answers) >17 years and compos mentis - Informed Concent, model 1[3]

Subjects or their legal representative (for minors, those that are non compos mentis) provide written or online permission Yes

Are subjects or their legal representatives (for minors, those that are non compos mentis) informed in advance in written Yes

If it is not possible to provide full disclosure prior to the study taking place, then full discolure will be provided. [answer selected from a number of options in a list]

Is the physical and/or psychological integrity of the participant affected by this No

Is personal data rigistered in this study No

Is there any additional information relevant to the cETO's assessment I did not see a form to upload the measurement tools and consent form for external institution (Zuyd Hogeschool), so I have included these with the information letters/informed consent forms.

Since the sign-ups for the two courses used in the field-experiment are not yet closed, it is not possible to give an exact number of potential (students in the two courses can participate voluntarily without any consequences for the course itself) participants. An estimate is 2 classes (40 students) for the web-development course and 2 classes (40 students) for the AI course.

The informed consent form for the field-test will be supplied digitally (and anonymous). This will form will be included before participants fill in the pre-knoweldge test.

Here you find the data storage procedure Yes I have carefully read and understood the data storage procedure and confirm my adherence to the protocol (must be conducted by the principal researcher or student, depending on who is the applicant) PLEASE CONFIRM [checkbox]

[3]For an overview of the different consent models, see https://www.ou.nl/documents/40554/4197127/V_01_Informed_consent_models.pdf/5520628b-d1db-d12f-9bbd-d77824ec9392?t=1649063360175

Check here whether the following forms were uploaded Participants information letter (or online form) / if applicable the protocol of verban information, including any written additional information about the study. [checkbox]

List of the measurement tools used (questionnaires/interview questions, etc.) [checkbox]

Participant (informed) consent form [checkbox]

External institution declaration of consent [checkbox]

Please confirm that you have filled in all applicable fields in the form Yes [checkbox]

Check below that option that fits your situation best It concerns a student research of which the research plan is approved by the supervisory team or is only for educational purposes only (e.g. master thesis graduation). [answer selected from a number of options in a list]

B.2. cETO REPLY

Commented on Wednesday June 30, 2021.

Dear Eddy,

The cETO has reviewed your application and has the following remarks:

Concerning the ethical review:

General:

- Since the focus sessions will be recorded, the cETO would like to point out the audio policy of the OU. You will find this hereafter:

The use of your own video/audio recorders is allowed (please note that smartphones are not allowed). These recorders should contain a memory card to safely store the data. It is under no circumstances allowed to transfer the data via Wifi. Microsoft Teams can be used for online recordings. The audio or video data can be stored in a password secured folder in Research drive. Student should delete the original data on the memory card as soon as possible. The data in Research drive can be used to transcribe the audio or video data in such a manner that persons are no longer traceable to a specific person. The supervisor can transfer the audio/video data and the transcripts from Research Drive to the secured T-drive. It is not allowed to store audio/video data or anonymous transcripts on a personal computer and they must be deleted. Advice is to start the recording after collecting the personal data. The OU offers several tools for analysing qualitative data, which can be found on the cETO portal.

- Can you clarify what tool will be used for the pre- and post knowledge tests?

- In addition to audio recordings, is other personal data collected (this seems to be necessary in order to organize among other things the focus group sessions)?

Concerning the recruitment:

- It is unclear to the cETO who is distributing the various information letters within the organisation of the Hogeschool Zuyd. The cETO advises you to have the spread of the information letters done by the Hogeschool Zuyd itself.
- It is also unclear to the cETO whether different strategies of recruitment will be used for the field-test and for the focus group sessions.
- Will contact information be collected during the recruitment?
- How do the participants receive the different information letters and how do they provide consent (online or on paper)?
- It is unclear to the cETO how many participants are required at least for each phase of the research.
- In your research, contact details are collected. During the research these can be stored on the Research drive or the T-drive (storing on a laptop is not allowed). The cETO advises you to store the codification file on the Research drive or the T-drive.

Concerning the information letters:

- Please check the information letters for language errors.
- How exactly will the participants be informed about the results of the research?
- The cETO advises you to remove the blocks about insurance and unexpected results.
- The cETO advises you to make clear that participants have the right to ask questions prior, during and after the research.
- The information letter about the experiment (RQ3) contains the following passage: '6. Einde van het onderzoek Uw deelname aan het onderzoek stopt als de focusgroep sessie voorbij is.'. The cETO advises you to adjust this.
- It is unclear to the cETO whether participants can participate in both the focus group sessions or is it the case that participants can only participate in one of the focus group sessions?
- Please mention the name of your supervisor.
- Please only use OU contact information. Your supervisor can request for a OU research e-mail at the Servicedesk of the OU.

The cETO would like to receive a response to these remarks point by point. The cETO advises to consult the supervisor when needed.

You can respond to this via 'reply' under the reaction of the cETO and react to the questions and comments. If and where necessary, it is possible to make adjustments via the edit in the application itself and / or upload new documents since it is in draft mode again.

The request is then to ’Submit’ the application to the supervisor again, after giving the response. The supervisor then receives a message and can approve the response (supervisor approved).

If there are any questions, please let me know: ceto@ou.nl(link sends e-mail)

Kind regards

B.3. Author reply

Replied on Sunday, Juli 11, 2021.

General:

- Since the focus sessions will be recorded, the cETO would like to point out the audio policy of the OU. You will find this hereafter:

The use of your own video/audio recorders is allowed (please note that smartphones are not allowed). These recorders should contain a memory card to safely store the data. It is under no circumstances allowed to transfer the data via Wifi. Microsoft Teams can be used for online recordings. The audio or video data can be stored in a password secured folder in Research drive. Student should delete the original data on the memory card as soon as possible. The data in Research drive can be used to transcribe the audio or video data in such a manner that persons are no longer traceable to a specific person. The supervisor can transfer the audio/video data and the transcripts from Research Drive to the secured T-drive. It is not allowed to store audio/video data or anonymous transcripts on a personal computer and they must be deleted. Advice is to start the recording after collecting the personal data. The OU offers several tools for analysing qualitative data, which can be found on the cETO portal.

Answer: If the situation surrounding Covid allows it, which seems to be the case at this moment, the plan is to do the focus group sessions in person. The audio recorders will be provided by Hogeschool Zuyd. These audio recorders will contain a memory card for data storage. The audio will be transferred to the T-drive and will be removed from the recording devices. The anonymous transcriptions will be made using the data on the T-drive.

- Can you clarify what tool will be used for the pre- and post knowledge tests?

Answer: The pre- and post-knowledge tests will be integrated into the prototype that generates the feedback. The pre-knowledge test will be provided to participants on first use of the prototype. The integration of the pre- and post-knowledge test into the prototype allows for coupling of the results of the tests and the data of the prototype while the participants can stay anonymous (since the prototype will not collect any personally identifiable information).

- In addition to audio recordings, is other personal data collected (this seems to be necessary in order to organize among other things the focus group sessions)?

Answer: E-mail addresses will be collected to organize the focus group sessions. These e-mail addresses will be stored on the T-drive.

Concerning the recruitment:

- It is unclear to the cETO who is distributing the various information letters within the organisation of the Hogeschool Zuyd. The cETO advises you to have the spread of the information letters done by the Hogeschool Zuyd itself.

Answer: My plan was to distribute the information letters myself (in my role as employee of Hogeschool Zuyd) in collaboration with the relevant coordinators within Hogeschool Zuyd. Additionally, I have added the recruitment letters to the application.

- It is also unclear to the cETO whether different strategies of recruitment will be used for the field-test and for the focus group sessions.

Answer: For the focus groups sessions, recruitment will be done by e-mail to second-, third- and fourth-year students. The recruitment for the field-experiment will be done through e-mail to classes of the relevant courses and by visiting the classes in the first week of the course. For both parts, the e-mail communication will be done through mailing lists available within Hogeschool Zuyd.

- Will contact information be collected during the recruitment

Answer: For the focus group sessions, e-mail addresses will be collected for organizing the sessions. For the field-experiment, no contact information is needed.

- How do the participants receive the different information letters and how do they provide consent (online or on paper)?

Answer: For both the focus group sessions and the field-experiment, participants will receive the information letter by e-mail. Participants in the focus group sessions will provide consent on paper ("12a_Model_1_toestemmingsverklaring_cETO" as found in the Help section of the cETO portal will be used for this purpose). Participants in the field-experiment will provide consent through an online form ("Example of online informed consent" as found in the Help section of the cETO portal will be used for this purpose).

- It is unclear to the cETO how many participants are required at least for each phase of the research.

Answer: For the focus group sessions, each session will at least require five participants (four students and one teacher). For the two focus group sessions, ten participants total are at least required. For the field-experiment, at least 35 participants are required.

- In your research, contact details are collected. During the research these can be stored on the Research drive or the T-drive (storing on a laptop is not allowed).

Answer: The contact details will be stored on the T-drive.

The cETO advises you to store the codification file on the Research drive or the T-drive.

Answer: The codification files will be stored on the T-drive.

Concerning the information letters:

- Please check the information letters for language errors.

Answer: An additional check for language errors has been done for all documents.

- How exactly will the participants be informed about the results of the research?

Answer: Participants of the focus group session will be informed via direct e-mail. This will be done by the researcher. The participants of the field-experiment will be informed via e-mail through class mailing lists. These e-mails will be distributed in the same manner as the recruitment information. This has also been made more clear in the information letters.

- The cETO advises you to remove the blocks about insurance and unexpected results.

Answer: These blocks have been removed from the information letters.

- The cETO advises you to make clear that participants have the right to ask questions prior, during and after the research.

Answer: Extra emphasis on this fact has been added to the introduction and section 8 of each information letter.

- The information letter about the experiment (RQ3) contains the following passage: '6. Einde van het onderzoek Uw deelname aan het onderzoek stopt als de focusgroep sessie voorbij is.'. The cETO advises you to adjust this.

Answer: This has been corrected in the information letter of the field-experiment.

- It is unclear to the cETO whether participants can participate in both the focus group sessions or is it the case that participants can only participate in one of the focus group sessions?

Answer: Participants can only participate in one of the focus group sessions.

- Please mention the name of your supervisor.

Answer: My supervisor has been added to the information letters.

- Please only use OU contact information. Your supervisor can request for a OU research e-mail at the Servicedesk of the OU.

Answer: My supervisor has requested an OU research e-mail account and I have received access to this account. The contact details in the information letters have been replaced with OU contact information.

C. MATRIX OF LITERATURE REVIEW SOURCES

This appendix gives an overview of the sources found during the literature review in Chapter 5. The matrix of the sources in shown in Table 5. The metric numbers refere to the numbers in Table 6 in appendix D.

#	*Title*	*Authors*	*Year*	*metrics*
1	A complexity measure	McCabe	1976	1
2	A metrics suite for object oriented design	Chidamber, Kemerer	1994	21, 6, 4, 5, 7, 5, 3
3	A practical model for measuring maintainability	Heitlager, Kuipers, Visser	2007	1, 8, 12, 10, 11, 9, 13
4	A statistical comparison of Java and Python software metric properties	Destefanis, Ortu, Pporru, Swift, Marchesi	2016	8, 19, 20, 21, 15, 16, 17, 18, 4, 14
5	A survey on software metrics	Kafura	1985	1, 10, 11, 23, 24, 25, 26, 22
6	A systematic review of software maintainability prediction and metrics	Rias, Mendes, Tempero	2009	
7	An emperical study of three common softwar complexity measures	O'Neal	1993	1, 8, 19, 21, 23, 24, 25, 26, 31, 6, 18, 4, 5, 27, 28, 29, 30, 32, 33, 34, 35, 36, 3
8	An evolution of software metrics: A review	Chhillar, Gahlot	2017	1, 8
9	An evaluation of the MOOD set of object-oriented software metrics	Harrison, Counsell, Nithi	1998	1, 8, 21, 23, 24, 25, 26, 37, 6, 4, 5, 27, 28, 29, 30, 7, 38, 39, 40, 41, 42, 43, 5, 3
10	An investigation into coupling measures for C++	Briand Devanbu, Melo	1997	38, 39, 40, 41, 42, 43
11	Clustering software metric values extracted from C# code for maintainability assessment	Arshad, Tjortjis	2016	45, 46, 44
12	Construction and testing of polynomials predicting software maintainability	Oman, Hagemeister	1994	21, 6, 4, 5, 7, 47, 5, 3
13	Decoupling level: a new metric for architectural maintenance complexity	Mo, Cai, Kazman, Xiao, Feng	2016	1, 2, 8, 19, 23, 24, 25, 26, 52, 53, 48, 49, 54, 27, 28, 29, 30, 51
14	Effect of software evolution of software metrics: An open source case study	Johari, Kaur	2011	55

Continued on next page.

#	*Title*	*Authors*	*Year*	*metrics*
15	Evaluating software complexity measures	Weyuker	1988	1, 8, 21, 6, 4, 5, 7, 57, 58, 5, 56, 3
16	A comparison of measures of control flow complexity	Baker, Zweben	1980	1, 2, 20, 23, 24, 25, 26, 59, 27, 28, 29, 30
17	A measure of control flow complexity in program text	Woodward, Hennel, Hedley	1979	1, 23, 24, 25, 26, 59, 27, 28, 29, 30
18	Evaluating usefulness of software metrics: an industrial experience report	Bouwers, Deursen, Visser	2013	59
19	A model for program complexity analysis	McClure	1978	1, 8, 21, 6, 4, 5, 7, 61, 5, 3, 60
20	Dependency profiles for software architecture evaluations	Bouwers, Deursen, Visser	2011	21
21	Analysis of metrics for object-oriented program complexity	Kim, Kusumoto, Kikunon, Chang	1994	61
22	How can software metrics help novice programmers?	Rcardell-Oliver	2011	8, 21, 6, 18, 4, 5, 7, 32, 35, 58, 62, 5, 36, 3
23	How much information doe software metrics contain	Gil, Goldstein, Moshkovich	2011	1, 8, 19, 18
24	Maintenance metrics for the object-oriented paradigm	Li, Henry	1993	21, 6, 4, 3
25	Measuring dependency freshness in software systems	Cox, Bouwers, Eekelen, Visser	2015	8, 21, 6, 4, 5, 7, 32, 5, 36, 3
26	Measuring the difficulty of code comprehension tasks using software metrics	Kasto, Whalley	2013	63
27	A congitive complexity metric based on category learning	Klemola Rilling	2003	1, 8, 19, 20, 21, 23, 24, 25, 26, 64, 66, 6, 18, 67, 68, 4, 5, 27, 28, 29, 30, 7, 65, 5, 3
28	Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics	Curtis, Sheppard, Milliman, Borst, Love	1979	68
29	MetricHaven - More than 23000 metrics for measuring quality attributes of software product lines	El-Sharkawy, Krafczyk, Schmid	2019	1, 23, 24, 25, 26, 27, 28, 29, 30
30	Metrics for measuring the quality of modularization of large-scale object-oriented software	Sarkar, Kak, Meskeri Rama	2008	1, 8, 10, 11, 66
31	NPATH: A measure of execution path complexity and it's applications	Nejmeh	1988	75, 76, 69, 70, 71, 72, 73, 74, 77, 78

Continued on next page.

#	*Title*	*Authors*	*Year*	*metrics*
32	On automatically collectable metrics for software maintainability evaluation	Ostberg, Wagner	2014	79
33	Quantifying the analyzability of software architectures	Bouwers, Correia, Deursen, Visser	2011	1, 8, 19, 21, 23, 24, 25, 26, 66, 6, 4, 5, 27, 28, 29, 30, 7, 5, 3
34	Software metrics: roadmap	Fenton, Neil	2000	81, 80, 60
35	Software quality metrics and their impact on ebedded software	Oliveira, Redin, Carro, Lamb, Wagner	2008	8
36	Software structure metrics based on information flow	Henry, Kafura	1981	1, 8, 21, 64, 66, 18, 85, 4, 7, 33, 58, 89, 90, 56, 86, 87, 88, 3, 82, 83, 84
37	Some stability measures for software maintenance	Yau, Collofello	1979	10, 11
38	Source code quality classification based on software metrics	Vytovtov, Markov	2017	1, 23, 24, 25, 26, 27, 28, 29, 30
39	Student portfolios and software quality metrics in computer science education	Patton, McGill	2006	1, 23, 24, 25, 26, 27, 28, 29, 30
40	Survey of object-oriented metrics: focusing on validation and formal specification	Sharma, Gill, Sikka	2012	1, 23, 24, 25, 26, 27, 28, 29, 30
41	The research on software metrics and software complexity metrics	Honglei, Wei, Yanan	2009	21, 6, 18, 4, 5, 45, 46, 7, 32, 38, 39, 40, 41, 42, 43, 44, 5, 36, 3
42	Using code quality features to predict bugs in procedural software systems	Araujo, Zapalowski, Nunes	2018	1, 8, 21, 6, 4, 5, 7, 38, 39, 40, 41, 42, 43, 5, 3

Table 5: Literature review sources matrix

D. Full List of Metric Scores

Table 6 shows a full list of all metrics extracted from the literature in Chapter 5 and their scores. The table has been ordered by total score (from highest to lowest). The source numbers refere to the numbers in Table 5 in appendix C.

#	*metric*	*CRC*	*CMN*	*CCM*	*CMP*	*CST*	*CIM*	*Total*	*Sources*
1	Cyclomatic complexity	3	3	3	3	3	3	18	1, 3, 5, 7, 8, 9, 13, 15, 16, 17, 19, 23, 27, 29, 30, 33, 36, 38, 39, 40, 42, 43
2	Extended cyclomatic complexity	3	3	3	3	3	3	18	13, 16, 43
8	Lines of code	3	3	3	3	3	3	18	3, 4, 7, 8, 9, 13, 15, 19, 22, 23, 25, 27, 30, 33, 35, 36, 42, 43
12	Duplication	3	3	3	3	3	3	18	3, 43
19	Lines of comments	3	3	3	3	3	3	18	4, 7, 13, 23, 27, 33
10	Fan-in	3	3	3	3	3	2	17	3, 5, 30, 37
11	Fan-out	3	3	3	3	3	2	17	3, 5, 30, 37
20	Number of statements	3	3	3	3	3	2	17	4, 16, 27
21	Invocation complexity/Control variable complexity	3	3	3	3	3	2	17	2, 4, 7, 9, 12, 15, 19, 20, 22, 24, 25, 27, 33, 36, 41, 42
23	Number of distinct operators	3	3	3	3	3	2	17	5, 7, 9, 13, 16, 17, 27, 29, 33, 38, 39, 40, 43
24	Number of distinct operants	3	3	3	3	3	2	17	5, 7, 9, 13, 16, 17, 27, 29, 33, 38, 39, 40, 43
25	Total number of operators	3	3	3	3	3	2	17	5, 7, 9, 13, 16, 17, 27, 29, 33, 38, 39, 40, 43
26	Total number of operants	3	3	3	3	3	2	17	5, 7, 9, 13, 16, 17, 27, 29, 33, 38, 39, 40, 43
31	Tokens per method	3	3	3	3	3	2	17	7

Continued on next page.

#	*metric*	*CRC*	*CMN*	*CCM*	*CMP*	*CST*	*CIM*	*Total*	*Sources*
37	Number of tokens	3	3	3	3	3	2	17	9
52	Number of tokens	3	3	3	3	3	2	17	13
53	Lines of data declarations	3	3	3	3	3	2	17	13
64	Number of parameters	3	3	3	3	3	2	17	27, 36
66	Nested block depth	3	3	3	3	3	2	17	27, 30, 33, 36
6	Response for a class	3	3	3	2	3	2	16	2, 7, 9, 12, 15, 19, 22, 24, 25, 27, 33, 41, 42
9	Number of files	1	3	3	3	3	3	16	3
13	Test coverage	3	3	3	3	2	2	16	3
15	Number of declared instance methods	2	3	3	2	3	3	16	4
16	Number of declared instance variables	2	3	3	2	3	3	16	4
17	Number of local methods	2	3	3	2	3	3	16	4
18	Total number of methods	2	3	3	2	3	3	16	4, 7, 22, 23, 27, 36, 41
48	Average variables span	3	3	3	3	3	1	16	13
49	Number of executable simecolons	3	3	3	3	3	1	16	13
54	maximum level of control structure nesting	3	3	3	3	3	1	16	13
67	Number of import statements	1	3	3	3	3	3	16	27
68	Kind of line of code identifier density	3	3	3	3	3	1	16	27, 28
79	NPATH	3	3	3	3	3	1	16	32
59	Program knots	3	3	3	3	3	1	16	16, 17, 18
85	Normalized distance from main sequence	3	3	3	3	3	1	16	36
4	Depth of inheritance tree	2	3	3	1	3	3	15	2, 4, 7, 9, 12, 15, 19, 22, 24, 25, 27, 33, 36, 41, 42

Continued on next page.

#	*metric*	*CRC*	*CMN*	*CCM*	*CMP*	*CST*	*CIM*	*Total*	*Sources*
5	Number of children	2	3	3	1	3	3	15	2, 7, 9, 12, 15, 19, 22, 25, 27, 33, 41, 42
27	Halstead's Length	3	3	1	3	3	2	15	7, 9, 13, 16, 17, 27, 29, 33, 38, 39, 40, 43
28	Halstead's Difficulty	3	3	1	3	3	2	15	7, 9, 13, 16, 17, 27, 29, 33, 38, 39, 40, 43
29	Halstead's Volume	3	3	1	3	3	2	15	7, 9, 13, 16, 17, 27, 29, 33, 38, 39, 40, 43
30	Halstead's Effort	3	3	1	3	3	2	15	7, 9, 13, 16, 17, 27, 29, 33, 38, 39, 40, 43
45	Number of class method interactions	3	3	3	1	3	2	15	11, 41
46	Number of method method interactions	3	3	3	1	3	2	15	11, 41
7	Lack of cohesion in methods	2	3	3	1	3	2	14	2, 9, 12, 15, 19, 22, 25, 27, 33, 36, 41, 42
14	Number of base classes	1	3	3	1	3	3	14	4
22	Stability	3	3	1	3	3	1	14	5
32	Data abstraction coupling	2	3	3	1	3	2	14	7, 22, 25, 41
33	Number of overridden methods	2	3	3	1	3	2	14	7, 36
34	Percentage of privade members	2	3	3	1	3	2	14	7
35	Number of properties	2	3	3	1	3	2	14	7, 22
38	Method hiding factor	2	3	3	1	3	2	14	9, 10, 41, 42
39	Attribute hiding factor	2	3	3	1	3	2	14	9, 10, 41, 42

Continued on next page.

#	*metric*	*CRC*	*CMN*	*CCM*	*CMP*	*CST*	*CIM*	*Total*	*Sources*
40	Method inheritance factor	2	3	3	1	3	2	14	9, 10, 41, 42
41	Attribute inheritance factor	2	3	3	1	3	2	14	9, 10, 41, 42
42	Polymorphism factor	2	3	3	1	3	2	14	9, 10, 41, 42
43	Couling factor	2	3	3	1	3	2	14	9, 10, 41, 42
44	Number of class attribute interactions	2	3	3	1	3	2	14	11, 41
47	Number of public methods	2	3	3	1	3	2	14	12
55	Decoupling level	1	3	3	3	3	1	14	14
57	Number of methods in a class	2	3	3	1	3	2	14	15
58	Number of attributes in a class	2	3	3	1	3	2	14	15, 22, 36
62	Weighted attributes per class	2	3	3	1	3	2	14	22
63	Dependency freshness	1	3	3	3	1	3	14	26
65	Number of constructors	2	3	3	1	3	2	14	27
75	Module uniformity	1	3	3	1	3	3	14	31
76	Class uniformity	1	3	3	1	3	3	14	31
89	Number of static attributes	2	3	3	1	3	2	14	36
90	Number of static methods	2	3	3	1	3	2	14	36
51	Average number of intermodule arguments	1	3	3	3	3	1	14	13
61	Dependency profiles	1	3	3	3	3	1	14	19, 21
69	Module interaction index	1	3	3	3	3	1	14	31
70	Non-API method closedness index	1	3	3	3	3	1	14	31
71	Intermodule inheritance	1	3	3	3	3	1	14	31
72	Not programming to interfaces index	1	3	3	3	3	1	14	31

Continued on next page.

#	*metric*	*CRC*	*CMN*	*CCM*	*CMP*	*CST*	*CIM*	*Total*	*Sources*
73	Association coupling index	1	3	3	3	3	1	14	31
74	State access violation index	1	3	3	3	3	1	14	31
77	API usage index	1	3	3	3	3	1	14	31
5	Coupling between object classes	2	3	3	1	3	1	13	2, 9, 12, 15, 19, 22, 25, 27, 33, 41, 42
56	Number of packages	1	3	3	1	3	2	13	15, 36
86	Abstractness	1	3	3	1	3	2	13	36
87	Number of classes	1	3	3	1	3	2	13	36
88	Number of interfaces	1	3	3	1	3	2	13	36
36	Message-Passing Couple	1	3	3	1	3	2	13	7, 22, 25, 41
3	Weighted methods per class	2	3	1	1	3	2	12	2, 7, 9, 12, 15, 19, 22, 24, 25, 27, 33, 36, 41, 42
78	Common use of module classes	1	3	3	1	3	1	12	31
81	Component size uniformity	1	3	3	1	3	1	12	34
82	Afferent coupling	1	3	3	1	3	1	12	36
83	Efferent coupling	1	3	3	1	3	1	12	36
80	System breakdown	1	1	3	1	3	1	10	34
84	Instability	1	3	1	1	3	1	10	36
60	Component balance	1	1	1	1	3	1	8	19, 34

Table 6: Metric score matrix

E. Prototype Design Details

This appendix gives a more technical overview of the prototype created for the evaluations in chapter 7. However, it should be noted that the system was initially designed with a larger scale field-experiment in mind. For example, extracting the code metrics from a project using a task queue and background workers is not needed when only a single user is interacting with the system at a time, but it would have been useful in cases where multiple projects are uploaded simultaniously.

E.1. Architecture

The base architecture for the prototype uses a client-server approach. In addition the server, or back-end, component of the prototype uses a layered approach. An overview of the architecture is shown in Figure 1. Each component is described in Table 7.

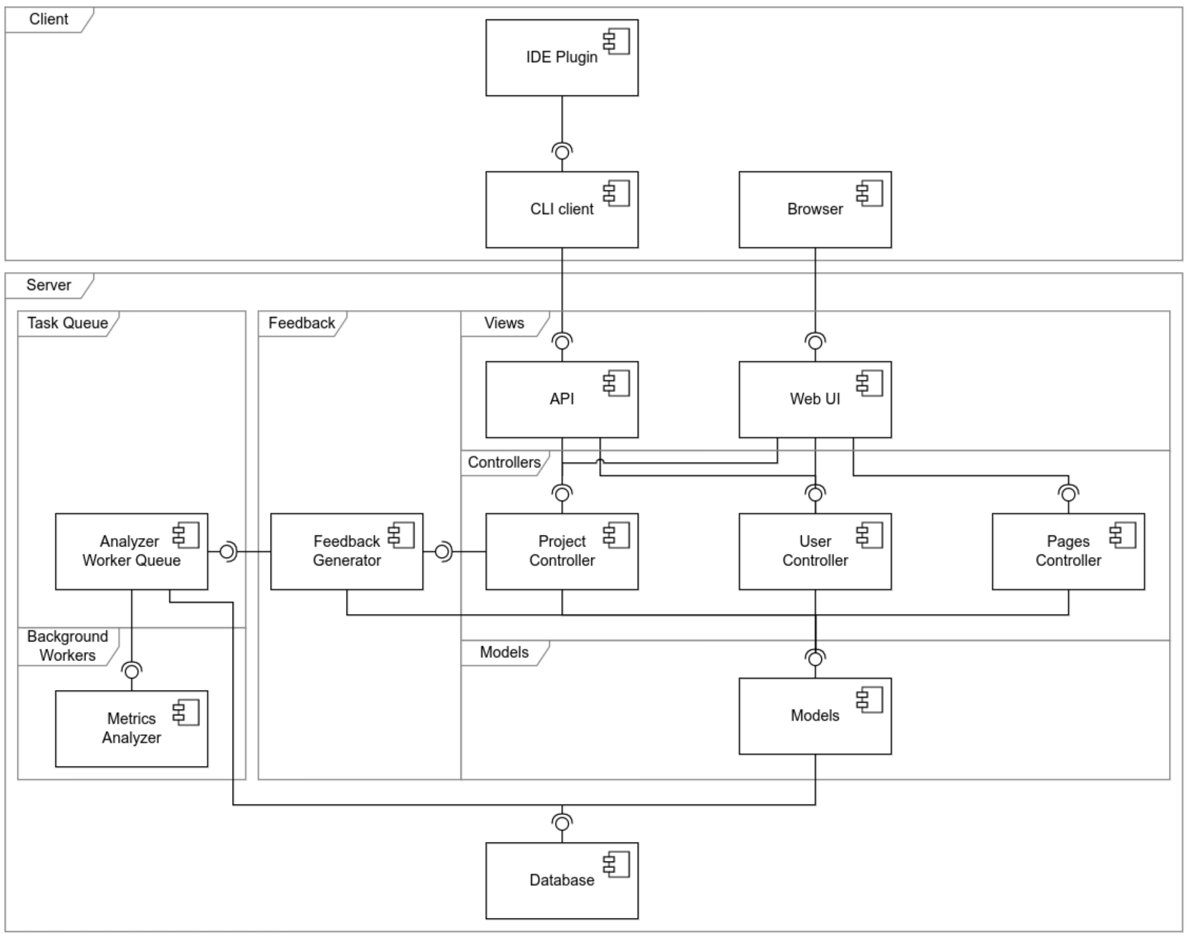


Figure 1: Component diagram

Component	*Description*
IDE Plugin	Plugin for the specific IDE (in this case, a Visual Studio plugin and a Visual Studio Code extension).
CLI client	A single executable that allows for communication with the API through a command line interface (CLI).
API	A REST-like API with endpoints for uploading code and polling the anlaysis status of a project.
Web UI	Web-based interface for viewing feedback and additional information.
Project Controller	Manages student projects, project versions, files, and the generated feedback.
User Controller	Manages users and authentication.
Pages Controller	Manages the additional information included in the prototype besides the feedback itself.
Feedback Generator	Manages the feedback generation process, including the handeling of the uploaded files, creating new tasks for the analyzers, and adding feedback to the metrics data.
Analyzer Worker Queue	Manages the task queue used by the Metrics Analyzers.
Metrics Analyzer	Runs the metrics extraction tools for the supported programming languages and transforms the data into a common format.
Models	The in-code representation of the tables in the database.

Table 7: Descriptions of components

One of the issues for the design of the prototype is that for the programming courses at Zuyd University of Applied Science do not require do not require students to use a prescribed IDE. This means that some students might use Visual Studio, some use Visual Studio Code, and others use Intellij based IDE's (e.g. PyCharm for Python or Rider for C#). To solve this, the functionality of the client has been implemented in a single executable with a command line interface (CLI). The plugins for the different IDEs make use of this CLI client. This is the same approach as used by the Git integration of many IDEs, which use the Git CLI for version management. Once a project has been uploaded, the rest of the interaction is done through a web-interface.

To support multiple languages and the different tools to extract metrics for each programming language, a common interface and data format have been defined. Each metric calculator is able to run the required tools to extract the metric data from a project and finally combine and convert the output of these tools into the common format. This data is then used to generate the feedback. The general interface for the metric calculators is implemented using an abstract class. The specific metric calculators for each supported programming language inherits from the base class. This is shown in Figure 2.

Frameworks, libraries, and tools

The prototype back-end has been build in Python using the Flask[4] web framework. The CLI client has been build using Go while the IDE plugins have been build using the sup-

[4]https://flask.palletsprojects.com

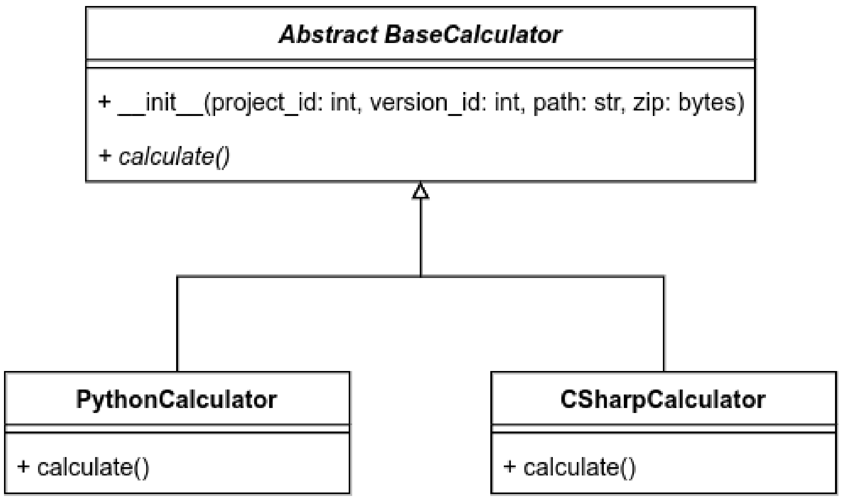


Figure 2: Class diagram - metric calculators

ported language for each IDE (C# for Visual Studio and Javascript/Typescript for Visual Studio Code). A combination of Celery[5] and Redis[6] have been used to create the task queue. The main database used for the prototype is Postgres[7]. For extracting the metrics, existing tools have been used. For extracting metrics from Python code, the Radon package has been used[8]. For C# we used the metrics component of the Roslyn Analyzers package[9].

E.2. Application Flow

Figure 3 shows the general flow of a student uploading a project and receiving feedback. This shows how a IDE plugin communicates with the back-end through the CLI client. Since the actual generation of the feedback is done asynchronously, the response to uploading a new version for a project is the project ID and the version ID. These IDs will be used by the IDE plugin and the CLI client to poll the API periodically on the status of the feedback generation. When the feedback generation is either done or has encountered an error, the IDE plugin will open the browser with the feedback overview page.

Figure 4 shows how a new version of a project is processed. When a new version of a project is uploaded, the Project Controller temporarily stores the code files of the project on the filesytem. The path to this temporary location is included as a parameters of the task, along with the project and version ID that the code belongs to. When a worker accepts the task, the path is used to point the correct metrics extraction tool to the code that needs to be analyzed. Since different metric extraction tool produce their own output with different formats, the worker also translates the output from the metric extraction tool into a common format. This data is returned as a result to the task queue store. The Analyzer Worker Queue takes the data from the task queue store and stores it in the database. At the same time, the Analyzer Worker Queue also updates the status of the project version to "DONE".

[5]https://docs.celeryq.dev/en/stable/
[6]https://redis.io/
[7]https://www.postgresql.org/
[8]https://github.com/rubik/radon
[9]https://github.com/dotnet/roslyn-analyzers

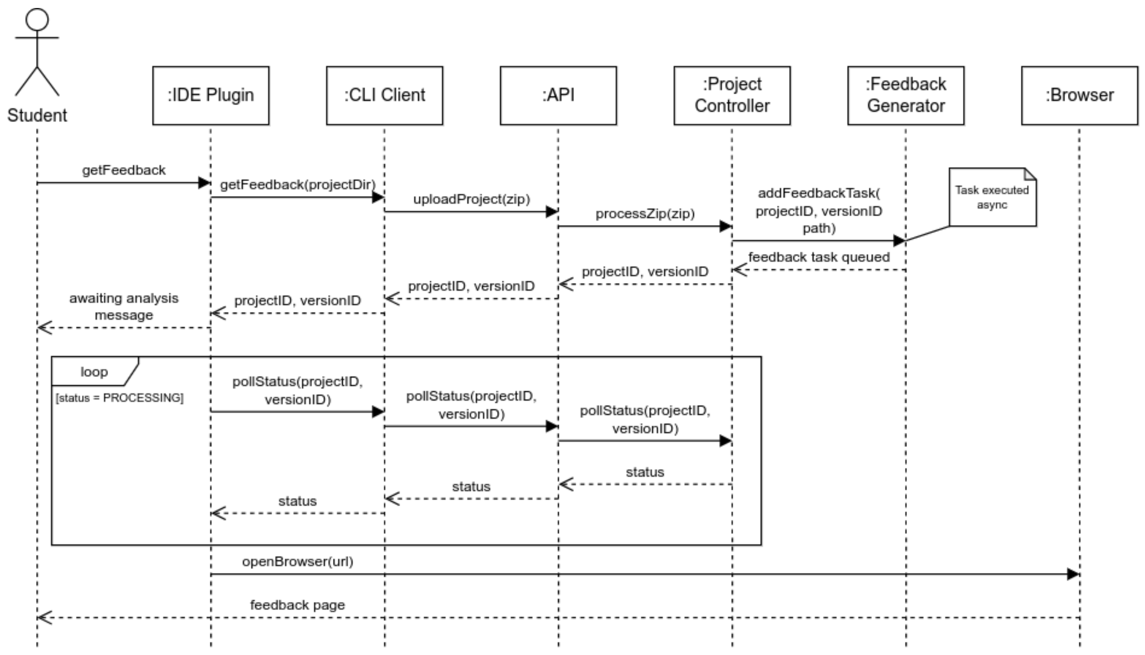


Figure 3: System sequence diagram - uploading feedback

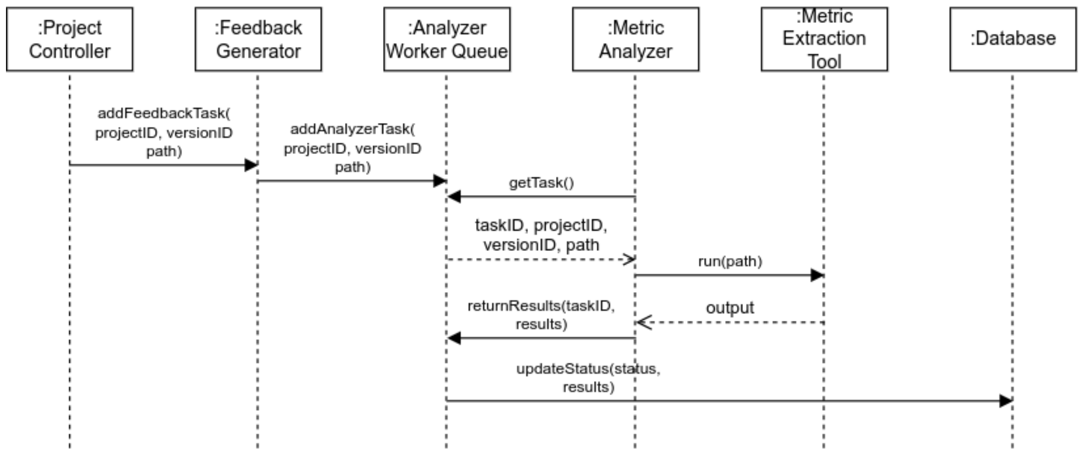

:Project Controller
:Feedback Generator
:Analyzer Worker Queue
:Metric Analyzer
:Metric Extraction Tool
:Database
addFeedbackTask(projectID, versionID path)
addAnalyzerTask(projectID, versionID path)
getTask()
taskID, projectID, versionID, path
run(path)
returnResults(taskID, results)
output
updateStatus(status, results)

Figure 4: Sequence diagram - project processing

E.3. Deployment

The prototype has been deployed on two servers provided by the Data Intelligence research group at Zuyd University of Applied Science. The main server is a Linux (Ubuntu 20.04 LTS) server which hosts the main back-end application, the Python analyzer, the task queue, and the database. The second server, a Windows 2019 server, is used to run a small service for the C# analyzer. The C# analyzer required a Windows server to support metrics extraction for .NET Framework applications. An overview of how the prototype is deployed is given in Figure 5.

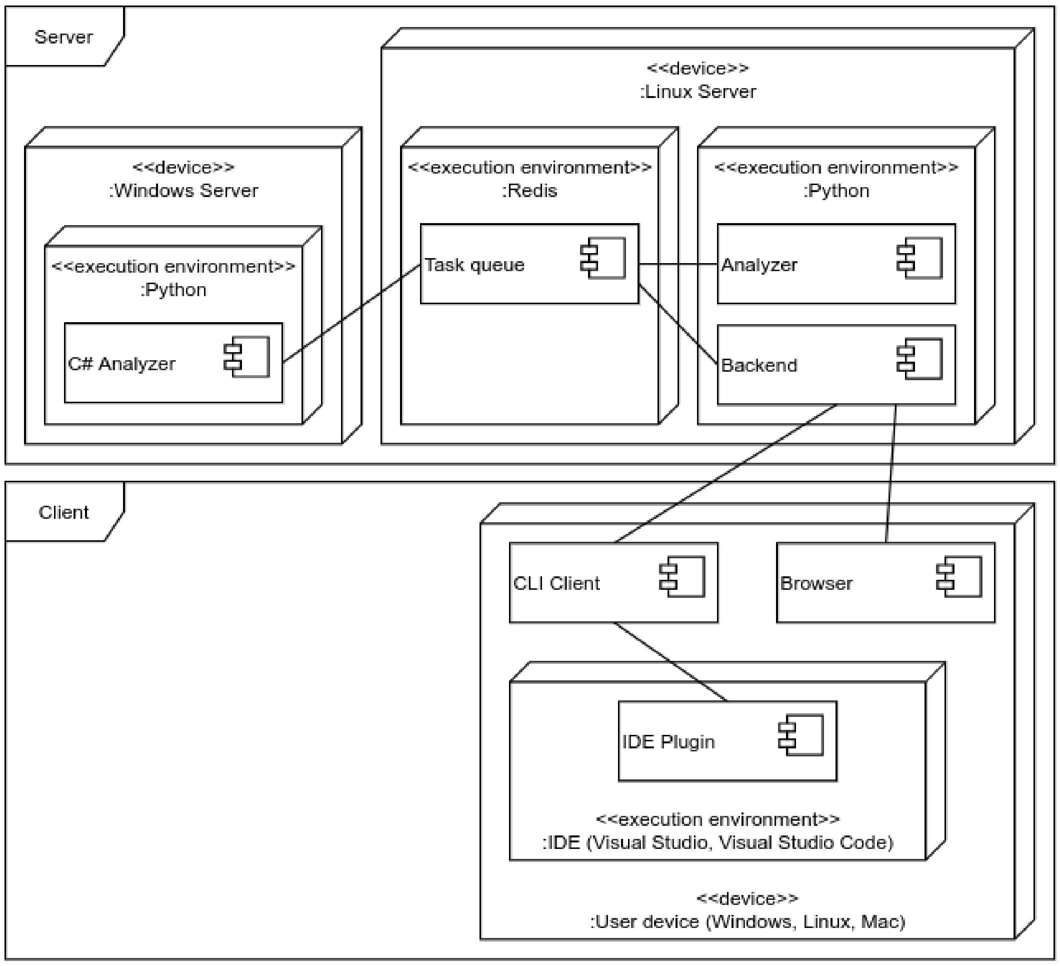

Server
<<device>>
:Linux Server
<<device>>
:Windows Server
<<execution environment>>
:Python
C# Analyzer
<<execution environment>>
:Redis
Task queue
<<execution environment>>
:Python
Analyzer
Backend
Client
CLI Client
Browser
IDE Plugin
<<execution environment>>
:IDE (Visual Studio, Visual Studio Code)
<<device>>
:User device (Windows, Linux, Mac)

Figure 5: Deployment diagram

F. Evaluation before and after interview questions

This appendix gives the before and after interview questions used during the evaluation in Chapter 7 (the methodology is further described in Section 4.4.3). Note that these questions were originally in Dutch and have been translated into English for this thesis.

F.1. Before Interview

Table 8 shows the question asked during the interview before the programming session.

Previous experience with code quality
What is your current experience with code quality?
How important do you think code quality is?
How difficult do you find it to write high quality code?

Self regulated learning
How difficult do you find it to learn new concepts, technologies, etc.?
How do you learn new concepts, technologies, etc.?

Use of tools
Which tools do you use during programming assignments?
If you use any tools, how do you use these?

Table 8: Evaluation before interview questions

F.2. After Interview

Table 9 shows the question asked during the interview after the programming session.

Principle 1: Link the feedback to the source code
To what extend is the link between the feedback and the code clear?
Which aspects make this link clear?
How could the link between the feedback and the code be improved?

Principle 2: Provide examples of high quality code
To what extend did the code examples help your understanding of code quality?
In what way do the code examples help?
How could the code examples be improved?

Principle 3: Provide the metric data with the feedback
To what extend are the metrics in the prototype clear?
To what extend do the metrics help you judge the quality of your code?
How could the metrics and their presentation be improved?

Continued on next page.

Principle 4: Provide comparisons of the metric results to both previous results and benchmark results
To what extend are the comparisons between different versions of your code useful?
To what extend are the comparisons between your code and the benchmarks useful?
How could these comparisons be improved?
Principle 5: Provide strategies to help improve the quality of the code
To what extend are the strategies for improving code provided in the prototype useful?
In what way did/would you use the strategies provided in the prototype?
How could the strategies provided in the prototype be improved?
Principle 6: Provide general information about code quality and software metrics in addition to the feedback
To what extend is the general information about code quality clear?
How does the general information about code quality affect your understanding?
How could the general information about code quality be improved?
Principle 7: Ensure that feedback is easy to access, but not intrusive
What is your opinion on the integration of the prototype with your programming workflow?
How could the intergration of the prototype be improved?
General
Do you think you would find a tool like this useful in future programming assignments?
How would a tool like this affect your learning about code quality?
Are there any other improvements or suggestions for the prototype?

Table 9: Evaluation after interview questions