Automated Feedback for Learning Code Refactoring

Hieke Keuning

Automated Feedback for Learning Code Refactoring

Druk: Ridderprint | www.ridderprint.nl
Ontwerp omslag: Robbert Menko
ISBN: 978-94-6416-127-4

Dit proefschrift komt voort uit een Promotiebeurs voor Leraren (projectnummer 023.005.063), gefinancierd door de Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO).

Automated Feedback for Learning Code Refactoring

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Open Universiteit
op gezag van de rector magnificus
prof. dr. Th.J. Bastiaens
ten overstaan van een door het
College voor promoties ingestelde commissie
in het openbaar te verdedigen

op vrijdag 9 oktober 2020 te Heerlen
om 13:30 uur precies

door

Hebeltje Wijtske Keuning

geboren op 14 augustus 1981 te Hardenberg

Contents

Chapter 1

Introduction

Learning to program is hard, or is it? Many papers in the field of novice programming begin with stating that it is indeed 'hard', 'challenging', and a 'struggle'. Guzdial contemplates that 'maybe the task of programming is innately one of the most complex cognitive tasks that humans have ever created'.[1] While there is plenty of evidence that students indeed struggle, there is also nuance: Luxton-Reilly believes our expectations and demands of novices are too high [172], reasoning that if children can learn how to program, it should not be that hard to learn at least the basics. He argues that demanding too much from beginners in too little time, which inevitably leads to unsatisfactory results, does not imply that programming itself is hard.

To write a good program, novices need knowledge of programming languages and tools, as well as the skills to adequately use these resources to solve actual problems [228]. Du Boulay identifies five main problem areas [71]: orientation (the goal of programming), the notional machine (an abstraction of how the computer executes a program), notation (language syntax and semantics), structures (such as plans or schemas to perform small tasks), and pragmatics (planning, developing, testing, debugging, etc.). Programming requires handling all of these aspects almost simultaneously, making things even harder. Consequently, it is not surprising that this high cognitive load being placed on novices, combined with often flawed mental models, leads to struggling students.

Research into programming education has been focussing mainly on student difficulties, and the mistakes that they make affecting functional correctness. Style, efficiency, and quality have played a minor role. This thesis aims to focus attention on these aspects by applying them to the context of novice

[1]Mark Guzdial, Computing Education Research Blog (2010). Is learning to program inherently hard?

programmers and the small programs that they write. One might think that bothering novice programmers with yet another topic would make it even harder for them. However, writing code that is more *readable* and *understandable*, and thinking about how code constructs work and how they can be used in the best way, could prepare students to become critical and quality-oriented programmers.

The central topic of this thesis revolves around students learning about code quality, and how tools in general and software technology in particular can be employed to support them. In this introductory chapter we first explore the context by looking briefly at the history of how programming is being taught and studied, and zoom in on the perceived difficulties. Next, we introduce the topic of (educational) tools to support students with programming. Then, the topic of programming style and code quality is discussed, establishing the terminology and its definitions used in this thesis, and we give some background of the topic's place in education. Finally, we list the research questions of this thesis, and describe how these questions are addressed in the subsequent chapters.

1.1 A short history of programming education research

With the emergence of modern, digital computing in the 1940s quickly came the realisation that programming was much more difficult than just performing some mechanical operations [77]. Simple workers would not suffice; people had to be trained properly so they could handle the intricacies of the job and take computer programming to a higher level. The profession of programmer quickly transformed into a well-paid and valued occupation.

Computer Science Education has been studied since the 1960s, when the demand for programmers first emerged. In that time there was a huge problem with finding enough, and qualified programmers. In the early days, programmers were mostly trained in-house at companies. Later, vocational schools offered educational programs. The ACM Special Interest Group on Computer Personnel Research (SIGCPR) was founded in 1962, together with the first standardised curriculum, marking the academic start of computer science as a discipline [227]. However, at that time institutions struggled to train competent programmers, because it was not generally known which traits a good programmer should have and how these could be taught [77].

During that time, research focused on the skill of programming and its psychology (e.g. ‘The psychology of computer programming’ by Weinberg [284]), inspired by the great demand for programmers, and methods for teaching programmers [103]. The main topics were the difficulties of novices, language design, and learning programming both for its own sake and as a means to learn other skills, such as mathematics. This was the era of Logo [79], BASIC, Pascal, and the emergence of the ‘notional machine’ [71].

Later, object orientation (through Smalltalk) and an increased focus on user interface and interaction through graphical artefacts and programmable devices gained popularity, laying the foundation for modern block-based languages such as Scratch. In the 1970s en 1980s the advent of cognitive science and learning sciences influenced the field. The first Intelligent Tutoring Systems for programming were built based on theories from cognitive science, such as the LISP Tutor [57].

In 1970 the ACM Special Interest Group on Computer Science Education (SIGCSE) was founded, followed by several other venues in which researchers and teachers share their work and experiences. Most of the aforementioned themes are still being investigated, and new themes have emerged as well. Recently, the research field has expanded because learning how to program is not just for computer science students, but also for kids, teenagers in high school, non-majors studying other topics (e.g. biology, physics), and an increasing number of employees needing to be trained in various computing skills. This motivated the need to change the name of the field to Computing Education Research (CEdR) [80]. Furthermore, the increasing availability of large amounts of educational data shows promises to learn more about how students approach computing, and to design better interventions to support them. However, this trend also requires improved validation methods and more replication studies [120].

Novice programming has been, and will continue to be, a major topic in this increasing research field. The 2018 systematic literature review of Luxton-Reilly et al. on introductory programming shows an increase in paper count from 2003 to 2017 with a factor three [173]. Categories with the most papers were ‘measuring student ability’, ‘student attitudes’, ‘tools’, ‘teaching techniques’ and ‘the curriculum in general’.

1.2 The struggles of novice programmers

'Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program.' [40]

In the early days of the field, it was believed that being able to program was innate. Programming was considered some kind of 'black art', shrouded in mystery. Aptitude testing was used for a long time to 'discover' who had the programming gene and who did not. However, after decades of research into which factor predicts programming skill, there is still no clear answer to that question [226]. It is a common belief that the grades of novice programming show a bimodal distribution in courses, implying there are the ones who get it (who have the 'geek gene'), and the ones who do not, and probably never will. This viewpoint is still very persistent: Lewis et al. [166] asked students and faculty to respond to the statement 'Nearly everyone is capable of succeeding in computer science if they work at it', and 77% of faculty rejected this statement, while the majority of students was positive towards it. Whether grades are truly bimodal is currently under debate [206].

Robins poses the 'Learning Edge Momentum (LEM)' hypothesis that states that once you have success in learning, you will more easily learn new concepts, because learning is most successful if you build on your current knowledge [226]. This is especially true in a domain in which concepts are tightly connected and often build upon each other. Programming gets easier to learn once you have learned something successfully, but at the other end becomes more difficult once you struggled at the beginning of your learning process. This hypothesis calls for a big emphasis on the early stages on learning programming, and a gradual build-up of knowledge and skills.

Failure rates of first CS courses have also had much attention in the last decades. Bennedsen and Caspersen [30] measure an average failure rate of 33%. Watson and Li [280] report a pass rate of 67%, and the latest results from Bennedsen and Caspersen [31] are an improved average failure rate of 28%, which is much lower compared to some other subjects such as college algebra. Reservations can be made because measuring failure rates is very difficult. A recent study tried to tackle this by *comparing* pass rates to those of introductory courses in other STEM (Science, Technology, Engineering and Mathematics) disciplines [240], finding an average of about 75% and some weak evidence that programming resides at the lower end.

The well-known McCracken study measured if students could really program *after* passing a novice course, and the authors were disappointed with their skills [189]. A comparable result was found by Lister et al. in a study with different assignments [167]. Apparently, there is still a lot to learn after the first course.

Misconceptions have been studied extensively, distinguishing between syntactical, conceptual and strategic misconceptions [219]. Students must learn the syntax of a programming language, learn how language constructs work and how a complete program is executed, and employ all of these aspects to create a program that solves a particular problem. Factors that contribute to these misconceptions are: complex tasks leading to high cognitive load, confusing formal and natural language, incorrectly applying previous math knowledge, incorrect mental models of program execution (the notional machine [71]), lack of problem-solving strategies, issues with tools and IDEs, and inadequate teaching.

What students mostly find hard is designing a program to solve a certain problem, subdividing functionality into methods, and solving bugs [156]. Garner et al. find comparable results: understanding the task, design and structure of a solution, and some basic typo/syntax issues [86].

There are still many open questions and debates on what is the most effective way to teach programming. For instance, the 'programming language wars' have also taken a prominent place in computing education research, disputing whether we should teach Python, Java, or C; start with object orientation, imperative programming or even a functional or logical paradigm; or maybe even begin with a visual language such as Scratch. To date, no clear answer has emerged to that question [173].

However, there are some things we know are effective [80]. We should teach a suitable language with a straight-forward syntax and helpful tool support, based on proper selection criteria [180]. Attention should be paid to building correct mental models of program execution. For example, Nelson et al. [199] propose a comprehension-first pedagogy, which first teaches how code is executed, before teaching how to write code. Teaching about this notional machine is often supported by tools simulating the computer, which is further discussed in the next section. This pedagogy also fits well with the best practice of offering different types of carefully designed exercises: reading, writing, expanding, and correcting code (e.g. [267]). Some of these exercise types reduce the cognitive load students particularly struggle with as beginners. Attention should also be paid to problem-solving, problem design, and programming strategies. A final effective method is collaboration in the form of pair programming or peer learning (e.g. [216]).

Some of these teaching methods can be supported by automated tools. In the next section we shift the focus to programming tools complementing human teachers.

1.3 Tools supporting the learning of programming

From the very beginning of the research field of programming education, tools have been developed to support students in their learning. Several categorizations can be made, for instance [44]:

- Algorithm and program visualization tools, which teach students how algorithms work and how programs are executed, and algorithm and program simulation tools, which go beyond visualization by providing interaction. Some examples are the Python Tutor [101] in which students can step through Python programs, UUhistle [247] that lets the student play the role of the computer through executing various commands, and TRAKLA2 [178], which creates visualisations of operations on data structures such as (balanced) binary trees and graphs.

- Automatic assessment tools, which provide grades and feedback on student submissions. Many of these systems run test cases on submitted programs and run additional tools to check style and performance. Ihantola et al. [119] provide a review, and a well-known system is Web-CAT [75].

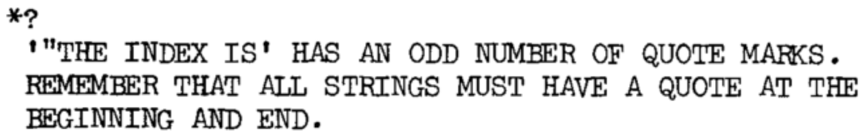
- Coding tools, in which student can practice and learn with programming. Nowadays, many of these tools are offered online, either free or commercial.[2]

- Problem-solving support tools, for instance Intelligent Tutoring Systems (ITSs). As an example, in Parsons' problems students have to put code fragments in the correct order so the program works [204]. These tools could also be more focussed on learning a specific skill.

For this thesis we are mostly interested in coding tools, problem-solving support tools (in particular ITSs), and aspects of automatic assessment tools.

[2]Examples are: `codeacademy.com`, `code.org`, `codingbat.com`

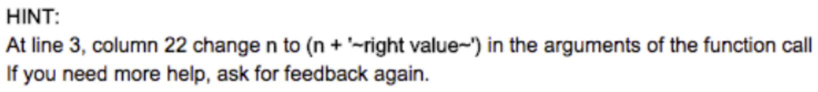
A central element in these tools is *feedback*, which has the potential to greatly affect learning if delivered properly [107], [239]. When the feedback is *automated*, it alleviates teachers from the effort of giving feedback for large groups of students. One of the earliest examples is shown in Figure 1.1, which is a feedback message on an incorrectly quoted string given by the BASIC Instructional Program (BIP) developed in the 1970s [19].



```
*?
'"THE INDEX IS' HAS AN ODD NUMBER OF QUOTE MARKS.
REMEMBER THAT ALL STRINGS MUST HAVE A QUOTE AT THE
BEGINNING AND END.
```

FIGURE 1.1: Feedback from the BASIC Instructional Program (BIP) in the 1970s [19].

A more recent example can be seen in Figure 1.2. This hint is from the Intelligent Teaching Assistant for Programming (ITAP) system that generates data-driven hints [225]. Data-driven solutions are increasingly being used for many applications, and have also taken a role in educational tools. ITAP searches for student paths from a similar starting point leading to a correct solution, and bases hints on the potentially most successful next step.



HINT:
At line 3, column 22 change n to (n + '~right value~') in the arguments of the function call
If you need more help, ask for feedback again.

FIGURE 1.2: Feedback from the Intelligent Teaching Assistant for Programming (ITAP) [224].

Feedback can be generated on various aspects of students' programming, such as programming mistakes, test results, task requirements, and, most relevant for this thesis, style and quality aspects. In ITSs, feedback is used for the *inner loop*, indicating whether steps are correct and giving feed forward in the form of next-step hints.

Several tutors have been developed in the domain of programming. For my Master thesis I worked on supporting students with building small programs step by step [139], [140]. I developed a prototype of a tutoring system that helps students with feedback and hints suggesting next steps to expand and refine their programs. This tutor is similar to the Ask-Elle tutor for functional programming [88], but adapted for the paradigm of imperative programming. Building this tutor introduced me to Ideas (Interactive Domain-specific Exercise Assistants), a framework for creating interactive learning environments [109]. This framework has been used for many different applications in various domains, such as programming [88], logic [168], mathematics [110], statistics [257], and communication skills [127]. Ideas is available as a software package[3] written in the functional programming language Haskell. Several components have to be built to make a tutor for a specific domain: a data structure for the artefacts to be manipulated (e.g. programs, expressions, or texts), rules that specify the steps to transform these artefacts, and strategies that combine, sequence, and prioritise these steps.

For the work in this PhD thesis I have used this framework and my earlier work on programming tutors to build a tutoring system for the domain of code refactoring. The next section will give some background on refactoring and code quality, and illustrates its place and importance in the field of programming education.

1.4 Teaching programming style and code quality

'Thus, programs must be written for people to read, and only incidentally for machines to execute.' [1]

In the context of this thesis, we define *code quality* as dealing with the directly observable properties of source code, such as control flow, expressions, choice of language constructs, decomposition, and modularization. The properties are derived from the rubric by Stegeman et al. [249], which has been developed to assess code quality in introductory programming courses. Other aspects such as naming, layout, and commenting are outside our scope. Coding *style* is often associated with quality. The topics of quality and style touch upon personal preferences and beliefs. Even though this might complicate deciding what and how to teach, we believe this should not just be left to the teacher. We should actively look for agreement and discuss what we disagree on.

[3] `hackage.haskell.org/package/ideas`

Code refactoring is defined as improving code step by step while preserving its external behaviour [84]. The starting point of refactoring is code that is already functionally correct, but has characteristics resembling *code smells.* The term 'refactoring' emerged from works in the early 1990s by Opdyke and Griswold, and some other software engineering communities, as investigated by Martin Fowler.[4] The effect of refactoring on software quality characteristics is not consistent [11], [194]; although improving quality attributes is the ultimate goal of refactoring, it can apparently do harm as well by negatively impacting these attributes.

In this thesis we focus on single methods and how to improve aspects such as flow, expressions and use of language constructs, thus refactoring at the data-, statement- and method-level [188]. These are not the types of higher-level refactorings most commonly known, such as 'Extract Method', 'Pull up Field' and 'Inline Class' as documented by Fowler [84]. However, Fowler also describes the 'Substitute Algorithm' refactoring as 'you want to replace an algorithm with one that is clearer'. We consider our focus to be on the micro-refactorings needed to perform this possibly complex task. Refactorings related to structure and modularity will be future work.

Attention for the stylistic aspects of code is not a new topic, and caught the attention of researchers in the past. In 1978 Schneider proposed ten principles for a novice programming course, among which number six: 'The presentation of a computer language must include concerns for programming style from the very beginning' [236]. However, programming style and quality have had much less attention than writing functioning programs and fixing mistakes such as compiler errors, runtime errors or incorrect output.

Recent increased attention might be attributed to changes in the field of software engineering. The growing need for technological solutions has led to software increasingly being made as products frequently updated with improvements and new features [3], [69]. Code is also more often shared as open source software to be expanded by others. These developments, and in general the increasing maturity of the field, call for understandable code that is easily maintained and extended.

Another interesting development is the ever expanding choice of tools dealing with quality and style available to developers, as well as the growing sophistication of IDEs. Developers can have their code analysed for bugs, flaws, and smells; metrics can be calculated for performance, test coverage,

[4]Martin Fowler, Blog – Etymology of refactoring (2003)

and complexity; and code can be automatically formatted and refactored. However, according to a study from 2012 the use of refactoring tools is not that widespread [195]. Another study investigated reasons why developers hesitate to use static analysis tools [130]. Developers mention the sheer overload of warnings, sometimes even false positives, and lack of explanation of why a warning is problematic and how to fix it. If even professional developers struggle with this, how should our students deal with these tools? Students learning programming can come into contact with these professional tools early, and should be taught how to use these tools wisely.

A 2017 ITiCSE working group intended to answer questions about how students, educators and professional developers perceive code quality, which quality attributes they consider important, and what the differences are between these groups. I was a participant of this working group. We interviewed individuals from various groups, questioning them on several aspects related to code quality. We found no coherent image of what defines code quality, although 'readability' was the most frequently mentioned indicator, and that all interviewed groups have learned very little on the subject in formal education, pleading for more attention to the subject [37].

More evidence for lack of attention to code quality comes from Kirk et al. [149], who investigated whether code quality is mentioned in learning outcomes of introductory programming courses in higher education. They found that in only 41 of 141 courses this was the case, and that it remained unclear what exactly students were supposed to learn if code quality *was* mentioned.

The Lewis study on teacher and student attitudes and believes towards Computer Science mentioned earlier [166] also contains the statement 'If a program works, it doesn't matter much how it is written'. While 92% of teachers rejected this, only 55% of students in CS1 rejected the statement. The rejection percentage went up, however, for CS2 and senior students.

This thesis supports the call for more attention to the subject of code quality in education.

1.5 Research questions and thesis structure

The central research question of this thesis is:

How can automated feedback support students learning code refactoring?

Each of the following chapters answers one of the subquestions, which will help finding an answer to the central research question:

RQ1 What are the characteristics of existing tools that give automated feedback on programming exercises?

RQ2 Which code quality issues occur in student code, and are these issues being fixed?

RQ3 How would teachers help students to improve their code?

RQ4 How do we design a tutoring system giving automated hints and feedback to learn code refactoring?

RQ5 What is the behaviour of students working in a such a tutoring system?

The thesis is composed of five main chapters, each being a published or submitted paper of which I am the first author. For those papers, I have developed the software, performed the analyses, and wrote the papers. For the literature review in Chapter 2, the second author also played an active role by participating in selecting papers and developing the coding, as is required to ensure quality for a systematic review. In general, the co-authors contributed to regular discussions on the research questions, and the research methods we would need to answer these questions. Some minor improvements have been made in the published chapters.

Chapter 2 "A Systematic Literature Review of Automated Feedback Generation for Programming Exercises". Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. In: *ACM Transactions on Computing Education (TOCE).* 2018. [147]

This thesis begins with a systematic literature review (SLR) of automated feedback generation for programming exercises. The first results of the literature study appeared as a conference paper [145], accompanied by a more detailed version as a Technical Report [146]. After reducing the scope, the final review was published as a journal paper [147]. The SLR has a broad focus looking at the earliest work from the 1960s up to papers from 2015. A total of 101 tools that provide automated feedback are included, describing the type of feedback they generate, the techniques used, the adaptability of the feedback, and the methods used for evaluation. To categorise the types of feedback, we have used an existing feedback content classification by Narciss [197] that we instantiated for the domain of programming. From this study we learn that feedback mostly focusses on correcting mistakes, and much less on helping students along the way of solving a programming problem. We observe an increasing diversity of techniques used for generating feedback, providing new opportunities, but also posing new challenges. Adaptability of tools and evaluating the use and effectiveness of feedback techniques also remains a concern.

Chapter 3 "Code Quality Issues in Student Programs". Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. In: *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education.* 2017. [141]

To find out to what extent code quality issues occur in student programs, we conducted a study analysing student code. We analysed over 2.5 million Java code snapshots for the presence of code smells using a professional static analysis tool. We selected a subset of rules from this tool that we categorised under a rubric for assessing student code quality [249]. We found several occurrences of issues and noticed they were barely resolved, in particular the modularization issues. We did not see the effect of an installed code quality tool extension on the number of issues found.

Chapter 4 "How Teachers Would Help Students to Improve Their Code". Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. In: *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education.* 2019. [142]

Teachers play an important role in raising awareness of code quality. Ideally, every student should receive personalised feedback from a teacher on their code, however, this is often an impossible task due to large class sizes. In this chapter we investigate teacher views on code quality through a survey. Thirty teachers gave their opinion on code quality, and were asked to make this concrete by assessing three imperfect student programs. We found quite a diversity in how they would rewrite the programs, but also extracted some similarities.

Chapter 5 "A Tutoring System to Learn Code Refactoring". Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. In submission. n.d. [143]

This chapter describes the tutoring system to learn code refactoring that was developed based on the findings from the previous studies. The tutoring system offers refactoring exercises, in which students have to rewrite imperfect solutions to given problems. The system offers feedback and hints at various levels, and is based on a ruleset derived from our preliminary research.

The imperative programming tutor developed for my Master thesis (see Section 1.3) has served as a basis for the refactoring tutor. Several components, such as the parser, the data types, normalisation rules, the code evaluator, have been reused and expanded. In this chapter the design of the system is described and its functionality is shown. The system is evaluated by comparing it with professional tools, conducting a technical evaluation, and showing to what extent the functionality matches with how teachers would help students.

Chapter 6 "Student Refactoring Behaviour in a Programming Tutor". Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. In submission. n.d. [144]

This chapter describes the findings of 133 students working with the tutoring system in the autumn of 2019. We elaborate on how they solved six refactoring exercises using the feedback and hints the system provides. Log data of all interactions were studied revealing their programming behaviour, difficulties and successes. We also analyse the results of the survey the students filled in on using the system and working on code quality. Several improvements for the tutoring system were derived from this study.

Chapter 7 The last chapter provides a final conclusion and reflection on the central topic of this thesis: automated feedback for students learning about code quality and refactoring. We derive general insights from the thesis and describe implications for future work. We also put the work of the published papers from Chapters 2 to 4 in the context of the latest work that has appeared since these papers were published.

1.5.1 Other work

The following papers are relevant work that I have done before and during my PhD, but are not a part of my thesis:

Code quality working group "'I know it when I see it' – Perceptions of Code Quality". Jürgen Börstler, Harald Störrle, Daniel Toll, Jelle van Assema, Rodrigo Duran, Sara Hooshangi, Johan Jeuring, Hieke Keuning, Carsten Kleiner, Bonnie MacKellar. In: *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education, Working Group Reports.* 2017. [37]

Section 1.4 elaborates on the study of the working group I participated in. My contribution consisted of conducting and transcribing interviews, and processing the interview data together with the other group members.

Imperative programming tutor "Strategy-based Feedback in a Programming Tutor". Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. In: *Proceedings of the Computer Science Education Research Conference.* 2014. [140]

As explained in Section 1.3, I built an imperative programming tutor of which some components have been reused for this thesis.

Predicting student performance "Automatically Classifying Students in Need of Support by Detecting Changes in Programming Behaviour". Anthony Estey, Hieke Keuning, and Yvonne Coady. In: *Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education.* 2017. [78]

This paper focuses on student behaviour in a programming tutor, investigating to what extent compile- and hint-seeking behaviour can predict failure or success in a programming course. We found that using a metric that incorporates behaviour change over time is more accurate at predicting outcome than a metric that calculates a score at a single point in time. My contribution consisted of taking part in discussions about the prediction metrics and writing down the findings.

Chapter 2

A Systematic Literature Review of Automated Feedback Generation for Programming Exercises

This chapter is a published paper [147].

Abstract Formative feedback, aimed at helping students to improve their work, is an important factor in learning. Many tools that offer programming exercises provide automated feedback on student solutions. We have performed a systematic literature review to find out what kind of feedback is provided, which techniques are used to generate the feedback, how adaptable the feedback is, and how these tools are evaluated. We have designed a labelling to classify the tools, and use Narciss' feedback content categories to classify feedback messages. We report on the results of coding 101 tools. We have found that feedback mostly focuses on identifying mistakes and less on fixing problems and taking a next step. Furthermore, teachers cannot easily adapt tools to their own needs. However, the diversity of feedback types has increased over the last decades and new techniques are being applied to generate feedback that is increasingly helpful for students.

2.1 Introduction

Tools that support students in learning programming have been developed since the 1960s [70]. Such tools provide a simplified development environment, use visualisation or animation to give better insight in running a program, guide students towards a correct program by means of hints and feedback messages, or automatically grade the solutions of students [138]. Two important reasons to develop tools that support learning programming are:

- learning programming is challenging [189], and students need help to make progress [48];
- programming courses are taken by thousands of students all over the world [30], and helping students individually with their problems requires a huge time investment of teachers [200].

Feedback is an important factor in learning [107], [239]. Boud and Molloy define feedback as 'the process whereby learners obtain information about their work in order to appreciate the similarities and differences between the appropriate standards for any given work, and the qualities of the work itself, in order to generate improved work' [38]. Thus defined, feedback is formative: it consists of 'information communicated to the learner with the intention to modify his or her thinking or behavior for the purpose of improving learning' [239]. Summative feedback in the form of grades or percentages for assessments also provides some information about the work of a learner. However, the information a grade without accompanying feedback gives about similarities and differences between the appropriate standards for any given work, and the qualities of the learner's work, is usually only superficial. In this article we focus on the formative kind of feedback as defined by Boud and Molloy. Formative feedback comes in many variants, and the kind of formative feedback together with student characteristics greatly influences the effect of feedback [191].

Focussing on the context of computer science education, Ott et al. [203] provide a roadmap for effective feedback practices for different levels and stages of feedback. The authors see a role for automated feedback at all three levels as defined by Hattie and Timperley [107]: 'task level', 'process level' and 'self-regulation level', discarding feedback at the 'self level' because of its limited effect on learning. In their roadmap, automated assessment of exams is placed at the task level, student support through adaptive feedback from automated tools at the process level, and tutoring systems and automated assessment as options for self-assessment of students at the self-regulation level.

Given the role of feedback in learning (programming), we want to find out what kind of feedback is provided by tools that support a student in learning programming. What is the nature of the feedback, how is it generated, can a teacher adapt the feedback, and what can we say about its quality and effect? An important learning objective for learning programming is the ability to develop a program that solves a particular problem. We narrow our scope by only considering tools that offer exercises (also referred to as tasks, assignments or problems, which we consider synonyms) that let students practice with developing programs.

To answer these questions, we have performed a systematic literature review of automated feedback generation for programming exercises. A systematic literature review (SLR) is 'a means of identifying, evaluating and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest' [150]. An SLR results in a thorough and fair examination of a particular topic. According to the literature, a research plan should be designed in advance, and the execution of this plan should be documented in detail, allowing insight into the rigorousness of the research.

This article expands on the results of the first iteration of our search for relevant tools, on which we have already reported in a conference paper [145] and a technical report [146]. This first iteration resulted in a set of 69 different tools, described in 102 papers. After slightly adjusting our criteria, the completed search resulted in a final collection of 101 tools described in 146 papers. We searched for papers in related reviews on tools for learning programming and executed multiple steps of 'backward snowballing' by selecting relevant references. We also searched two scientific databases and performed backward snowballing on those results as well.

We have classified the kind of feedback given by the tools we found by means of Narciss' [197] categories of feedback, such as 'knowledge about mistakes' and 'knowledge about how to proceed'. We have instantiated these feedback categories for programming exercises, and introduce several subcategories of feedback particular to programming. Narciss' categories largely overlap with the categories used to describe the actions of human tutors when they help students learning programming [266]. Next, we determine *how* these tools generate feedback by examining the underlying techniques. Besides looking at feedback categories (the output of a tool) and the technique (what happens inside a tool), we also look at the input. The input of a tool that supports learning programming may take the form of model solutions, test cases, feedback messages, etc., and determines to a large extent the adaptability of the tool, which is considered important [44], [170]. Finally, we collect information about the effectiveness of the generated feedback. The effectiveness of a tool depends on many factors and tools have been evaluated by a large variety of methods.

This review makes the following contributions:

- An extensive overview of tools that give automated feedback.
- A description of what kind of feedback is used in tools that support a student in learning programming. Although multiple other reviews analyse such tools, none of them specifically looks at the feedback provided by these tools.
- An analysis of the relation between feedback content and its technology, and the adaptability of the tool.

This article expands on our previous conference paper [145] in the following ways:

- We removed 23 tools from our initial set of 69, after adjusting our inclusion criteria based on the initial findings (described in Section 2.3.2). We completed our search by adding data for 55 new tools.
- We give elaborated examples and descriptions of several of these tools.
- We provide and discuss new tables and graphs summarising our final results. We look at the data more in-depth by identifying trends in time, and combinations of techniques and methods.
- We update, extend and fine-tune the discussion of the results, resulting in a more nuanced conclusion because of the characteristics of more recent tools that were included later.

The article is organised as follows. Section 2.2 discusses related reviews of tools for learning programming. Section 2.3 gives our research questions and research method, and Section 2.4 describes the labelling we developed for coding the tools. The results are described in Section 2.5 to 2.9, each describing the results for one of the research questions. Section 2.10 discusses the results and limitations of this review and Section 2.11 concludes the article.

2.2 Related work

We have found seventeen reviews of tools for learning programming, mostly on automated assessment (AA) tools [6], [46], [70], [119], [222], [229], [253] or learning environments for programming [66], [67], [93], [102], [138], [164], [165], [207], [213], [264]. Generating feedback is important for both kinds of tools. Most AA tools only grade student solutions, but some tools also provide elaborated feedback, and can be used to support learning [6]. We refer to the technical report on the first phase of our review [146] for a detailed discussion of these related reviews, in which we identified their main research questions, the scope of the selected tools and the method of data collection.

Most review papers describe the features and characteristics of a selection of tools, identify challenges, and direct future research. Except for the review by Ihantola et al. [119], authors select papers and tools based on unknown criteria. Some mention qualitative factors such as impact (counting citations) or the thoroughness of the evaluation of the tool. Most studies do not strive for completeness, and the scope of the tools that are described varies greatly. Tools are usually categorised, but there is no agreement on the naming of the different categories. Very few papers discuss technical aspects.

Our review distinguishes itself from the above reviews by focusing on the aspect of generating feedback in programming learning tools, closely examining the different types of feedback messages and identifying the techniques used to generate them. Furthermore, we employ a more systematic approach than all of the above reviews: we select tools in a systematic way following strict criteria, and code them using a predetermined labelling.

2.3 Method

Performing an SLR requires an in-depth description of the research method. Section 2.3.1 discusses our research questions. Section 2.3.2 describes the criteria that we have set to define the scope of our research. Section 2.3.3 describes the process for searching relevant papers. Finally, Section 2.3.4 explains the coding process.

2.3.1 Research questions

The following four research questions guide our review on automated feedback generation for programming exercises:

RQ 1 What is the nature of the feedback that is generated?

RQ 2 Which techniques are used to generate the feedback?

RQ 3 How can the tool be adapted by teachers, to create exercises and to influence the feedback?

RQ 4 What is known about the quality and effectiveness of the feedback or tool?

2.3.2 Criteria

There is a growing body of research on tools for learning programming for various audiences with different goals. These goals can be to learn programming for its own sake, or to use programming for another goal [138], such as creating a game. Our review focuses on students learning to program for its own sake. We have defined a set of inclusion and exclusion criteria (Table 2.1) that direct our research and target the characteristics of the papers and the tools described therein.

Although there are many online programming tools giving feedback, we do not include tools for which there are no publications, because we do not know how they are designed. The rationale of our functionality criteria is that the ability to develop a program to solve a particular problem is an important learning objective for learning programming [134]. Because we are interested in improving learning, we focus on formative feedback. We use the domain criteria to focus our review on programming languages used in the industry and/or taught at universities. Many universities teach an existing, textual programming language from the start, or directly after a visual language such as Scratch or Alice. We do not include visualisation tools for programming because they were surveyed extensively by Sorva et al. [246] in the recent past. However, we do include visualisation tools that also provide textual feedback.

Le and Pinkwart [164] have developed a classification of programming exercises that are supported in learning environments. The type of exercises that a learning tool supports, determines to a large extent how difficult it is to generate feedback. Le and Pinkwart base their classification on the degree of ill-definedness of a programming problem. Class 1 exercises have a single correct solution, and are often quiz-like questions with a single solution, or slots in a program that need to be filled in to complete some task. Class 2 exercises can be solved by different implementation variants. Usually a program skeleton or other information that suggests the solution strategy is provided, but

TABLE 2.1: Criteria for the inclusion/exclusion of papers.

	Include	**Exclude**
General	Scientific publications (journal papers and conference papers) in English. Master theses, PhD theses and technical reports only if a journal or conference paper is available on the same topic. The publication describes a tool of which at least a prototype has been constructed.	Posters and papers shorter than four pages.
Functionality	Tools in which students work on programming exercises of class 2 or higher from the classification of Le and Pinkwart [164] (see Section 2.3.2). Tools provide automated, textual feedback on (partial) solutions, targeted at the student.	Tools that only produce a grade, only show compiler output, or return instructor-written feedback.
Domain	Tools that support a high-level, general purpose, textual programming language, including pseudo-code.	Visual programming tools, e.g. programming with blocks and flowcharts. Tools that only teach a particular aspect of programming, such as recursion or multi-threading.
Technology	–	Tools that are solely based on automated testing and give feedback based on test results.

variations in the implementation are allowed. Finally, class 3 exercises can be solved by applying alternative solution strategies, which we interpret as allowing different algorithms as well as different steps to arrive at a solution.

We select papers and tools that satisfy all inclusion criteria and none of the exclusion criteria. We have included four PhD theses, one Master thesis and three technical reports, whose contributions have also been published in a journal or conference paper, because they contained relevant information. Since no review addressing our research questions has been conducted before, and we aim for a complete overview of the field, we consider all relevant papers up to and including the year 2015.

The criterion to exclude tools solely based on automated testing was added after publishing our preliminary results [145], because of the sheer volume of papers that we found. These papers all describe very similar tools, which would make the review too large. Moreover, we do not think that including these papers would provide an interesting contribution within the scope of this review.

2.3.3 Search process

The starting point of our search for papers was the collection of 17 review papers described in Section 2.2. Two authors of this SLR independently selected relevant references from these reviews. Then two authors independently looked at the full text of the papers in the union of these selections, to exclude papers not meeting the criteria. After discussing the differences, we assembled a final list of papers for this first iteration. Following a 'backwards snowballing' approach, one author searched for relevant references in the papers found in the first iteration. This process was repeated until no more new papers were found. We believe that one author is sufficient for this task because the scope had already been established.

Next, we searched two databases to identify more papers of interest, and to discover more recent work. We have selected a computer science database (ACM Digital Library) and a general scientific database (Scopus). We used the search string from Listing 2.1 on title, abstract and key words, slightly adjusted for each database.

Listing 2.1: Database search string

```
    (exercise OR assignment OR task OR (solv* AND problem))
AND programming
AND (   (tutor OR tutoring)
     OR ((learn OR teach) AND (tool OR environment))
     OR ((automat* OR intelligent OR generat*)
            AND (feedback OR hint))
    )
```

Although the query could have been adjusted so that it would have returned more papers that match our criteria, this adjustment would also have generated a much larger number of irrelevant results. We believe the final query has a good enough balance between accuracy and breadth, and because we also traced references we had an alternative way to find papers that we would have missed otherwise.

The results of the Scopus search were partly inspected by two authors, who separately selected papers by inspecting the title, abstract, key words and the name of the journal or conference. We combined the results and discussed all differences. In the event of disagreement, we included the paper for further inspection. Two authors then further refined the acquired list by examining the full text separately and again discussing differences. The second part of the Scopus search results, the ACM search results and the relevant references from both searches were inspected by one author, only consulting another author if there were doubts.

We had to exclude a small number of papers we could not find after an extensive search and, in some cases, contacting the authors. Some excluded papers point to a potentially interesting tool. We checked if these papers mention a better reference that we could add to our selection.

When we encountered papers we did not trust, we looked further into its contents, author, or the journal that published it. We excluded one paper from the review after all authors agreed that the paper was unreliable and would have a negative influence on the quality of our review (this particular paper seemed to be a copy of existing work).

Often multiple papers have been written on (versions of) a single tool. We searched for all publications on a tool by looking at references from and to papers already found, and searching for other relevant publications by the authors. We selected the most recent and complete papers about a tool. We prefer journal papers over conference papers, and conference papers over theses or technical reports. All papers from which we collected information appear in our reference list.

Table 2.2 shows the number of papers found by the searches. Many papers appeared multiple times in our search, both in references and in database searches. The table only counts a tool when it first appeared in the search, which was conducted in the order of the sources in the table.

2.3.4 Coding

To systematically encode the information in the papers, we developed a labelling (see Section 2.4) based on the answers to the research questions we expected to get, refined by coding a small set of randomly selected papers. One of the authors coded the complete set of papers. Whenever there were questions about the coding of a paper, another author checked. In total, 24.8% of the codings were (partly) checked by another author. Most of the checks were done in the earlier stages of the review. A third author joined the general discussions about the coding. When necessary, we adjusted the labelling.

TABLE 2.2: Results of database search and snowballing. Our previous work [145] included the 46 tools from the first iteration of review papers, and 23 additional tools that we excluded after adjusting the criteria for this final review.

		Snowballing iterations**				
Source	**Papers***	**1st**	**2nd**	**3rd**	**4th**	**Total**
Review papers	–	46 (76)	15 (17)	6 (9)	2 (2)	69 (104)
Scopus database	1830	25 (35)	5 (5)	–	–	30 (40)
ACM Digital library	798	2 (2)	–	–	–	2 (2)
						101 (146)

* excluding duplicates and invalid entries
** number of tools (number of papers)

2.4 Labelling

This section describes the labels used for our coding.

2.4.1 Feedback types (RQ1)

Narciss [197] describes a 'content-related classification of feedback components' for computer-based learning environments, in which the categories target different aspects of the instructional context, such as task rules, errors and procedural knowledge. We use these categories and extend them with representative subcategories identified in the selected papers. Narciss also considers the *function* (cognitive, meta-cognitive and motivational) and *presentation* (timing, number of tries, adaptability, modality) of feedback, which are related to the effectiveness of tutoring. We do not include these aspects in our review because it is often unclear how a tool or technique is used in practice (e.g. formative or summative).

Narciss first identifies three *simple* feedback components:

- Knowledge of performance for a set of tasks (KP): summative feedback on the achieved performance level after doing multiple tasks, such as '15 of 20 correct' and '85% correct'.
- Knowledge of result/response (KR): feedback that communicates whether a solution is correct or incorrect. We identify the following meanings of correctness of a programming solution: (1) it passes all tests, (2) it is

equal to a model program, (3) it satisfies one or more constraints, (4) a combination of the above.

- Knowledge of the correct results (KCR): a description or indication of a correct solution.

These types of feedback are not intended to 'generate improved work', a requirement in the feedback definition by Boud and Molloy. Moreover, Kyrilov and Noelle [155] have investigated the effect of instant binary feedback (messages that either contain 'correct' or 'incorrect') in automated assessment tools and found harmful effects on student behaviour. They found that students who received this kind of messages plagiarised more often and attempted fewer exercises. Because we focus on formative feedback on a single exercise, we do *not* identify these types in our coding.

The next five types are *elaborated* feedback components. Each type addresses an element of the instructional context. Below we describe these types and their subtypes in detail.

Knowledge about task constraints (KTC)

This type focusses on the task itself, and is subdivided into two subtypes:

- Hints on task requirements (TR). A task requirement for a programming exercise can be to use a particular language construct or to not use a particular library method.
- Hints on task-processing rules (TPR). These hints provide general information on how to approach the exercise and do not consider the student's current work.

Narciss gives a larger set of examples for this type of feedback, such as 'hints on type of task'. We do not identify this type because the range of exercises is limited by our scope. Also, we do not identify 'hints on subtasks' as a separate category, because the exercises we consider are relatively small. Instead, we label these hints with KTC-TPR.

Knowledge about concepts (KC)

We distinguish two subtypes:

- Explanations on subject matter (EXP), generated while a student is working on an exercise.
- Examples illustrating concepts (EXA).

Knowledge about mistakes (KM)

KM feedback messages have a type and a level of detail. The level of detail can be *basic*, which can be a numerical value (total number of mistakes, grade, percentage), a location (line number, code fragment), or a short type identifier such as 'compiler error'; or *detailed*, which is a description of the mistake, possibly combined with some basic elements. We use five different labels to identify the type of the mistake:

- Test failures (TF). A failed test indicates that a program does not produce the expected output.
- Compiler errors (CE). Compiler errors are syntactic errors (incorrect spelling, missing brackets) or semantic errors (type mismatches, unknown variables) that can be detected by a compiler and are not specific for an exercise.
- Solution errors (SE). Solution errors can be found in programs that do not show the behaviour that a particular exercise requires, and can be runtime errors (the program crashes because of an invalid operation) or logic errors (the program does not do what is required), or the program uses an alternative algorithm that is not accepted.
- Style issues (SI). In various papers we have found different definitions of programming style issues, ranging from formatting and documentation issues (e.g. untidy formatting, inconsistent naming, lack of comments) to structural issues and issues related to the implementation of a certain algorithm (use of control structures, elegance).
- Performance issues (PI). A student program takes too long to run or uses more resources than required.

Knowledge about how to proceed (KH)

We identify three labels in this type. Each of these types of feedback has a level of detail: a *hint* that may be in the form of a suggestion, a question, or an example; a *solution* that directly shows what needs to be done to correct an error or to execute the next step; or both hints and solutions.

- Bug-related hints for error correction (EC). Sometimes it is difficult to see the difference between KM feedback and EC. We identify feedback as EC if the feedback clearly focuses on what the student should do to correct a mistake.
- Task-processing steps (TPS). This type of hint contains information about the next step a student has to take to come closer to a solution.
- Improvements (IM). This type deals with hints on how to *improve* a solution, such as improving the structure, style or performance of a correct solution. However, if style- or performance-related feedback is presented in the form of an analysis instead of a suggestion for improvement, we label it as KM. The IM label has been added after we published the results of the first iteration of our search [145].

Knowledge about meta-cognition (KMC)

Meta-cognition deals with a student knowing which strategy to use to solve a problem, if the student is aware of their progress on a task, and if the student knows how well the task was executed. According to Narciss, this type of feedback could contain 'explanations on metacognitive strategies' or 'metacognitive guiding questions'.

2.4.2 Technique (RQ2)

We distinguish between general techniques for Intelligent Tutoring Systems (ITSs), and techniques specific for the programming domain. Each category has several subcategories.

General ITS techniques

- Tools that use model tracing (MT) trace and analyse the process that the student is following solving a problem. Student steps are compared to production rules and buggy rules [192].
- Constraint-based modelling (CBM) only considers the (partial) solution itself, and does not take into account how a student arrived at this (partial) solution. A constraint-based tool checks a student program against

predefined solution constraints, such as the presence of a for-loop or the calling of a method with certain parameters, and generates error messages for violated constraints [192].

- Tutors based on data analysis (DA) use large sets of student solutions from the past to generate hints. This type was also added after publishing our first results [145].

Domain-specific techniques for programming

- Dynamic code analysis using automated testing (AT). The most basic form of automated testing is running a program and comparing the output to the expected output. More advanced techniques are unit testing and property-based testing, often implemented using existing test frameworks, such as JUnit.

- Basic static analysis (BSA) analyses a program (source code or bytecode) without running it, and can be used to detect misunderstood concepts, the absence or presence of certain code structures, and to give hints on fixing these mistakes [253].

- Program transformations (PT) transform a program into another program in the same language or a different language. An example is normalisation: transformation into a sublanguage to decrease syntactical complexity. Another example is migration: transformation into another language at the same level of abstraction.

- Intention-based diagnosis (IBD) uses a knowledge base of programming goals, plans or (buggy) rules to match with a student program to find out which strategy the student uses to solve an exercise. IBD has some similarities to CBM and static analysis, and some solutions are borderline cases. Compared to CBM, IBD provides a more complete representation of a solution that captures the chosen algorithm.

- External tools (EX) other than testing tools, such as standard compilers or static code analysers. These tools are not the work of the authors themselves and papers do not usually elaborate on the inner workings of the external tools used. If a tool uses automated testing, for which compilation is a prerequisite, we do not use this label.

2.4.3 Adaptability (RQ3)

We identify several input types for tools that enable teachers to create exercises and influence the generated feedback.

- Solution templates (ST) (e.g. skeleton programs and projects) presented to students for didactic or practical purposes as opposed to technical reasons such as easily running the program.
- Model solutions (MS) are correct solutions to a programming exercise.
- Test data (TD), by specifying program output or defining test cases.
- Error data (ED) such as bug libraries, buggy solutions, buggy rules and correction rules. Error data usually specify common mistakes for an exercise.

Another aspect we consider is the adaptability of the feedback generation based on a student model (SM). A student model contains information on the capabilities and level of the student, and may be used to personalise the feedback.

2.4.4 Quality (RQ4)

As a starting point for collecting data on the quality of tools, we have identified and categorised how tools are evaluated. Tools have been evaluated using a large variety of methods. We use the three main types for the assessment of tools distinguished by Gross and Powers [96].

- Anecdotal (ANC). Anecdotal assessment is based on the experiences and observations of researchers or teachers using the tool. We will not attach this label if another type has been applied as well, because we consider anecdotal assessment to be inferior to the other types.
- Analytical (ANL). Analytical assessment compares the characteristics of a tool to a set of criteria related to usability or a learning theory.
- Empirical assessment. Empirical assessment analyses qualitative data or quantitative data. We distinguish three types of empirical assessment:

- Looking at the learning outcome (EMP-LO), such as mistakes, grades and pass rates, after students have used the tool, and observing tool use.
- Student and teacher surveys (EMP-SU) and interviews on experiences with the tool.
- Technical analysis (EMP-TA) to verify whether a tool can correctly recognise (in)correct solutions and generate appropriate hints. Tool output for a set of student submissions can be compared to an analysis by a human tutor.

2.5 General tool characteristics

In this section we discuss the general characteristics of the tools we investigated, such as their type, supported programming language and exercises. Table 2.3 shows an overview of these characteristics and the papers we consulted for each tool. The complete coding is available as an appendix to this article and as a searchable online table.[1]

In the remainder of this article we only cite papers on tools in specific cases. We refer to tools by their name in SMALL CAPS, or the first author and year of the most recent paper (AUTHOR00) on the tool we have used.

History

Figure 2.1 gives an impression of when the tools appeared in time. Because we do not know exactly in which time frame tools were active, we calculated the rounded median year of the publications related to a tool that we used for our review. Between the 60s and 80s a small number of tools appeared. Since the 90s we can see an increase in the number of tools, which slowly grows in the 2000s and 2010s.

Tool types

The tools that fall within our criteria are mostly either Automated Assessment (AA) systems or Intelligent Tutoring Systems (ITSs). AA systems focus on assessing a student's final solution to an exercise with a grade or a feedback report, to alleviate instructors from manually assessing a large number

[1]`www.hkeuning.nl/review`

Table 2.3: Tool publications, supported language paradigm and exercise class. Class 2 tools support exercises that can be solved by a predetermined strategy, allowing small variations. Exercises in class 3 tools can be solved by multiple strategies (see Section 2.3.2).

Tool	Publications	Language	Class
ACT Programming Tutor (APT)	[54]–[56]	Multi	2
ADAPT	[87]	Log	3
(Ala-mutka04)	[7]	Unknown	3
(Allemang91)	[10]	Imp/OO	3
AnalyseC	[289]	Imp/OO	3
APROPOS2	[169]	Log	3
Ask-Elle	[89], [90], [126]	Fun	3
ASSYST	[122], [124]	Imp/OO	3
At(x)	[25], [26]	Multi	3
AutoGrader	[111]	Imp/OO	3
AutoLEP	[279]	Imp/OO	3
AutoStyle	[193]	Imp/OO	3
AutoTeach	[15], [16]	Imp/OO	3
BIP	[19]–[21]	Imp/OO	3
Bridge	[35]	Imp/OO	2
C-tutor	[245]	Imp/OO	3
Camus	[270]	Imp/OO	3
Ceilidh	[27]–[29]	Multi	3
(Chang00)	[49]	Imp/OO	2
Checkpoint	[76]	Multi	3
CHIRON	[233]	Imp/OO	3
COALA	[135], [136]	Imp/OO	2
Code Hunt	[208], [209]	Imp/OO	3
CourseMarker/CourseMaster	[85], [112], [114]	Multi	3
CSTutor	[104], [105]	Imp/OO	3
Ctutor	[154]	Imp/OO	2
(Dadic11)	[59], [60]	Imp/OO	3
datlab	[175], [176]	Imp/OO	3
DISCOVER	[223]	Imp/OO	2
EASy	[177]	Imp/OO	3
ELM-PE/ELM-ART (II)	[45], [281]–[283]	Fun	3
ELP	[260], [261]	Imp/OO	2
(Fischer06)	[82]	Imp/OO	3
FIT Java Tutor	[97]–[99]	Imp/OO	3
FLIP	[137]	Imp/OO	3
GAME (2, 2+)	[33], [34], [184], [185]	Multi	3
(Ghosh02)	[91]	Imp/OO	3
Grace	[190], [221]	Imp/OO	2
(Gulwani14)	[100]	Imp/OO	3
HabiPro	[275], [276]	Imp/OO	2
(Hasan88)	[106]	Imp/OO	3
(He94)	[108]	Imp/OO	2
(Hong04)	[116]	Log	3
INCOM	[160], [162], [163]	Log	3
InSTEP	[202]	Imp/OO	2
INTELLITUTOR (II)	[263]	Imp/OO	2
IPTS	[291]	Imp/OO	2
ITAP	[225]	Imp/OO	3
ITEM/IP	[43]	Imp/OO	3
J-LATTE	[115]	Imp/OO	2
JACK	[92], [152], [153], [251]	Imp/OO	3
(Jackson00)	[123]	Imp/OO	3
Java Sensei	[22]	Imp/OO	2
(Jin12)	[128]	Imp/OO	3
(Jin14)	[129]	Imp/OO	2
JITS	[255], [256]	Imp/OO	2
(Keuning14)	[140]	Imp/OO	3
(Kim98)	[148]	Imp/OO	3
Koh	[151]	Imp/OO	2
LAURA	[4]	Imp/OO	2
(Lazar14)	[158]	Log	3
LISP tutor	[14], [57]	Fun	2
Ludwig	[237]	Imp/OO	3
(Mandal07)	[179]	Imp/OO	3
Marmoset	[248]	Multi	3
MENO-II	[244]	Imp/OO	3
(Naur64)	[198]	Imp/OO	3
Online Judge	[50]	Imp/OO	3
PASS	[258]	Imp/OO	2
PATTIE	[61]	Imp/OO	3
Pex4Fun	[259]	Multi	3
PHP ITS	[285], [286]	Imp/OO	3
PRAM	[113], [181]	Log	3
ProgramCritic	[232]	Imp/OO	3
ProgTest	[63]–[65]	Imp/OO	3
ProPAT_deBUG	[23]	Imp/OO	3
ProPL	[157]	Imp/OO	2
PROUST	[131], [233]	Imp/OO	3
(Rosenthal02)	[230]	Imp/OO	3
(Ruth76)	[231]	Imp/OO	3
(Sant09)	[235]	Imp/OO	3
SCENT	[186], [187]	Fun	3
Scheme-robo	[234]	Fun	3
(Shimic12)	[238]	Imp/OO	2
(Singh13)	[241]	Imp/OO	3
SIPLeS-II	[290]	Imp/OO	3
Smalltalker	[51]	Imp/OO	2
SOP	[242]	Imp/OO	3
(Striewe11)	[252]	Imp/OO	3
submit	[271]	Imp/OO	3
Submit!	[215]	Multi	3
Talus	[196]	Fun	3
Test My Code	[205], [274]	Imp/OO	3
Testovid	[217]	Multi	3
Ugo	[121]	Log	3
VC Prolog Tutor	[212]	Log	3
Virtual Programming Lab	[214]	Multi	3
(Vujosevic-Janicic13)	[277]	Imp/OO	3
Web-CAT	[74], [75]	Multi	3
WebToTeach	[18]	Imp/OO	3
WebWork-JAG	[94], [95]	Imp/OO	3

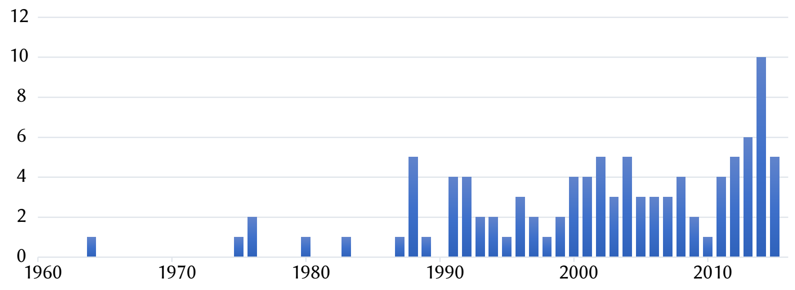


FIGURE 2.1: Number of tools per median year of their publications.

of students. ITSs help students to arrive at the solution by offering help at each step [268]. Other tools we found are programming environments for novices with a feedback component. A newer type of such a tool is the educational programming game, or serious game, in which learning programming is more implicit and considered a side effect of playing a fun game. Examples are PEX4FUN and its successor CODE HUNT that challenge students to iteratively discover the specification of a hidden program. CODE HUNT has an even stronger gaming vibe to it, created by the 'worlds' and 'levels' in which the player solves programming problems.

Some papers more generally describe a technique that can be used for generating feedback, which could be applied in an assessment or learning tool. We came across many papers on debugging and program understanding techniques, of which we only included the ones that were clearly used in an educational context.

Programming language

Tools offer either exercises for a specific programming language, a set of programming languages within a particular paradigm, or multiple languages within multiple paradigms. Table 2.4 shows the distribution of the different paradigms. The majority of the 101 tools supports programming in imperative languages, including object-oriented languages. Tools developed in the 21st century often support imperative languages such as Java, C and C++, whereas older tools provide exercises in ALGOL (NAUR64), FORTRAN (LAURA) and Ada (ASSYST). Some recent tools focus on (web) scripting languages such as PHP,

TABLE 2.4: Supported language paradigm of tools (n=101).

Paradigm	%
Imperative/object-oriented	73.3
Multiple paradigms	11.9
Logic	7.9
Functional	5.9
Unknown	1.0

JavaScript and Python. Tools for functional programming languages support Lisp (the LISP TUTOR), Scheme (SCHEME-ROBO) or Haskell (ASK-ELLE). All tools for logic programming offer exercises in Prolog. The remaining tools support multiple languages of different types and paradigms, and are often test-based AA systems.

Programming exercises often require a student to write a few lines of code or a single function, meaning that many tools only support a subset of the features of a programming language. For instance, a tool that requires programming in Java, an object-oriented language, may not support feedback generation on class declarations.

Exercise type

We have recorded the highest exercise class a tool supports and found that 23.8% of the tools support exercises of class 2 (can be solved by a predetermined strategy), and 76.2% exercises of class 3 (can be solved by multiple strategies). However, exercises that do not require a student to write code him or herself do not easily fit in this classification. For example, in PATTIE and PROPL the student engages in a conversation with an automated tutor to solve a programming problem. If the student makes a good suggestion, the tutor expands the solution-in-progress. CHIRON, the successor to the intelligent debugger PROUST also incorporates an interactive question/answer session with the student. Through natural language parsing CHIRON responds to questions on topics such as terminology, goal/plan implementation and data flow. We have included these systems because this approach closely mimics the behaviour of a human tutor.

Another uncommon type of exercise is offered by the programming game CODE HUNT that asks the student to write a program for which he or she does not know the specification yet. Instead, the student has to discover the specification by inspecting the results of a set of test cases and modifying their code to satisfy these tests.

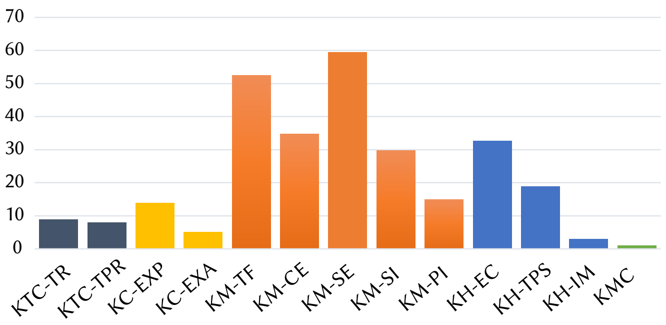
2.6 Feedback types (RQ1)

This section describes the results of the first research question: ‘what is the nature of the feedback that is generated?’ Figure 2.2 shows for each feedback type the percentage of tools that offer it, including the distinction between class 2 and class 3 tools. The percentages do not add up to 100%, because one tool can provide more than one feedback type. We have found KTC (knowledge about task constraints) feedback and KC (knowledge about concepts) feedback in only a few tools, with 14.9% and 16.8% respectively. KM (knowledge about mistakes) is by far the largest type: with 96.0% it is found in almost all tools. The subtype of KM we found most in the first part of our review, was TF (test failures). After omitting purely test-based tools in this final review, SE (solution errors) is the largest category with 59.4%. We have found KH feedback (knowledge about how to proceed) in 44.6% of the tools, of which EC feedback (error correction) is the largest subcategory with 32.7%.

Class 2 tools provide more solution error feedback (83.3% versus 51.9%), and more KH feedback (knowledge about how to proceed) with around twice as much for each subtype. Class 3 tools provide more test failure feedback (59.7% versus 29.2%) and more style issue feedback (36.4% versus 8.3%) and performance issue feedback (19.5% versus 0%). KTC (knowledge about task constraints) feedback is also seen in fewer class 2 tools. We also calculated that in 11.9% of the tools one type of feedback is generated, but 48.5% of the tools only generate one of the five main types (these percentages are not in the table). In the next subsections we expand on the different feedback types and provide examples of tools and the feedback messages they provide.

2.6.1 Knowledge about task constraints (KTC)

The first things a student should know when attempting an exercise are the requirements of the task and possibly some information on how to approach it.



	All (n=101)		**Class 2 (n=24)**		**Class 3 (n=77)**	
Feedback type	Count	%	Count	%	Count	%
KTC Knowledge about task constraints	*15*	*14.9*	*2*	*8.3*	*13*	*16.9*
TR Hints on task requirements	9	8.9	1	4.2	8	10.4
TPR Hints on task-processing rules	8	7.9	1	4.2	7	9.1
KC Knowledge about concepts	*17*	*16.8*	*5*	*20.8*	*12*	*15.6*
EXP Explanations on subject matter	14	13.9	3	12.5	11	14.3
EXA Examples illustrating concepts	5	5.0	2	8.3	3	3.9
KM Knowledge about mistakes	*97*	*96.0*	*24*	*100.0*	*73*	*94.8*
TF Test failures	53	52.5	7	29.2	46	59.7
CE Compiler errors	35	34.7	8	33.3	27	35.1
SE Solution errors	60	59.4	20	83.3	40	51.9
SI Style issues	30	29.7	2	8.3	28	36.4
PI Performance issues	15	14.9	0	0.0	15	19.5
KH Knowledge about how to proceed	*45*	*44.6*	*16*	*66.7*	*29*	*37.7*
EC Bug-related hints for error correction	33	32.7	13	54.2	20	26.0
TPS Task-processing steps	19	18.8	7	29.2	12	15.6
IM Improvement hints	3	3.0	1	4.2	2	2.6
KMC Knowledge about meta-cognition	*1*	*1.0*	*1*	*4.2*	*0*	*0.0*

FIGURE 2.2: Number of tools (count) and percentage of tools that offer a feedback type (by subtype and combined by main type), for all tools, and subdivided by exercise class. A tool can offer more than one feedback type.

Hints on task requirements (TR)

An example of this subtype can be found in the INCOM system. When a student makes a mistake with implementing the method header, feedback is given by ‘highlighting keywords in the task statement and advising the student to fully make use of the available information’ [161].

Another example can be found in the BASIC INSTRUCTIONAL PROGRAM (BIP), a system from the seventies. Some exercises in BIP require the use of a specific language construct. If this construct is missing from the student solution, the student will see the following message [19]:

```
Wait. Something is missing.

For this task, your program should also include the following basic state-
ment(s): FOR
```

Automated assessment tools have to check for task requirements as well. For example, if an exercise requires implementing a method that is also available in the standard library of the language, the assessment tool will have to check if this library method was not used. The AA tool from FISCHER06 provides the following feedback when a student uses a prohibited method [82]:

```
signature and hierarchy: failed
Invocation test checks whether prohibited classes or methods are used; call
of method reverse from the prohibited class java.lang.StringBuffer
```

Hints on task-processing rules (TPR)

For this category we consider the built-in feedback related to the tutoring strategy of the tool, as opposed to the dynamically generated feedback based on a (partial) student solution that we categorise under KM and KH. Tools that provide this kind of feedback may have a built-in stepwise approach to solving a programming problem. The ACT PROGRAMMING TUTOR shows a ‘skill meter’ that indicates the probability that a student knows a ‘rule’ (a rule corresponds to a step). The automated tutor in ADAPT gives some general information on how to solve a particular exercise [87]:

```
There are 2 major components to this template:

  • base case
  • recursive step

For the base case, the basic idea is to stop processing when the list becomes
empty and return 0 for the sum. For the recursive step, the basic idea is
to [...]
```

2.6.2 Knowledge about concepts (KC)

This feedback type deals with the subject matter of the exercise.

Explanations on subject matter (EXP)

This subtype can be found in the Ask-Elle tutor that refers to relevant internet sources when a student encounters certain language constructs.

Examples illustrating concepts (EXA)

The LISP tutor uses this type of feedback in its tutoring dialogue. After a student has made a mistake, the tutor might respond with [14]:

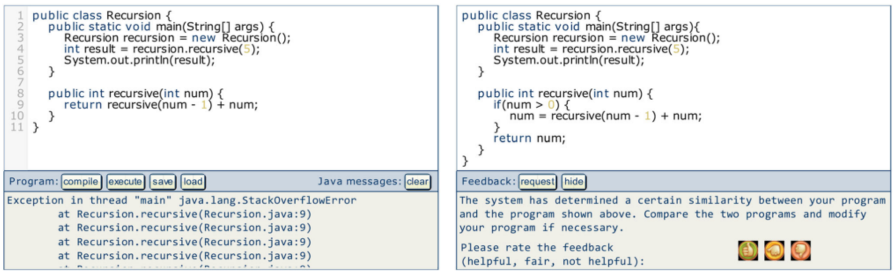
```
That is a reasonable way to think of doing factorials, but it is not a
plan for a recursive function.  Since you seem to be having trouble with
the recursive cases, let us work through some examples and figure out the
conditions and actions for each of these cases.
```

The ELM-PE/ART tutors support ‘example-based programming’, and provide a student with an example program that the student solved in the past, specifically selected to help the student solve a new problem.

The FIT Java Tutor uses machine learning techniques to generate example-based feedback, as shown in Figure 2.3. The program on the right is an example program (KC-EXA) for the student who programmed the erroneous program on the left. Differences with respect to the example program may be highlighted in the student program (KM-SE), and a feedback message tells the student that the program on the right is one step further (KH-TPS). The student can compare the two programs and identify mistakes and next steps. The data set consists of both student solutions and sample solutions by experts, from which a representative solution is chosen to compare with the student solution. In situations in which no representative solution can be selected, the student solution is compared to a similar program from the data set. However, it is possible that the similar program is not correct, in which case the feedback addresses the possible incorrectness and asks the student to identify mistakes in the similar program.

2.6.3 Knowledge about mistakes (KM)

This most common feedback type deals with reporting mistakes to students.



```
public class Recursion {
   public static void main(String[] args) {
      Recursion recursion = new Recursion();
      int result = recursion.recursive(5);
      System.out.println(result);
   }

   public int recursive(int num) {
      return recursive(num - 1) + num;
   }
}
```

```
Exception in thread "main" java.lang.StackOverflowError
        at Recursion.recursive(Recursion.java:9)
        at Recursion.recursive(Recursion.java:9)
        at Recursion.recursive(Recursion.java:9)
        at Recursion.recursive(Recursion.java:9)
```

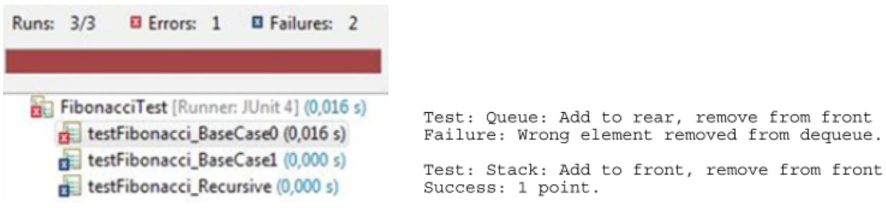
```
public class Recursion {
   public static void main(String[] args){
      Recursion recursion = new Recursion();
      int result = recursion.recursive(5);
      System.out.println(result);
   }

   public int recursive(int num) {
      if(num > 0) {
         num = recursive(num - 1) + num;
      }
      return num;
   }
}
```

The system has determined a certain similarity between your program and the program shown above. Compare the two programs and modify your program if necessary.

Please rate the feedback (helpful, fair, not helpful):

Figure 2.3: Feedback from the FIT Java Tutor (image from [97]).



```
Runs: 3/3    Errors: 1    Failures: 2

FibonacciTest [Runner: JUnit 4] (0,016 s)
   testFibonacci_BaseCase0 (0,016 s)
   testFibonacci_BaseCase1 (0,000 s)
   testFibonacci_Recursive (0,000 s)
```

```
Test: Queue: Add to rear, remove from front
Failure: Wrong element removed from dequeue.

Test: Stack: Add to front, remove from front
Success: 1 point.
```

Figure 2.4: Feedback showing test cases in COALA on the left (image fragment from [135]) and feedback on test execution in Testovid on the right (image fragment from [217]).

Test failures (TF)

A large number of tools give feedback based on the success and failure of executing tests on student programs. Online Judge is an automatic judge for programming contests that provides basic feedback on test failures. The system returns a short string such as '[..x]' as feedback, indicating that tests cases 1 and 2 are successful (indicated by a dot) and test case 3 is not successful (indicated by an 'x').

Figure 2.4 shows two examples of more detailed feedback on test execution. COALA shows the output of JUnit test cases integrated in a customised Eclipse environment. Testovid provides more informative feedback on the execution of test cases. We found this type of feedback, which resembles the output of professional testing tools, in many AA tools.

ProgTest requires a student to upload his or her own tests together with a solution. The tool shows the results of testing the student's code with the student's tests and the instructor's tests, but also the results of testing the model program of the instructor with the tests of the student. Additionally, the tool presents the results of a code coverage analysis. The authors of ProgTest have put effort into improving the understandability of the test coverage output. Based on this output, they calculate six metrics, such as 'testing completeness', 'program correctness' and 'tests adequacy', the results of which indicate how well the student performed on different aspects of testing. Instructor-written hints on failed tests can be shown as well.

Compiler errors (CE)

Feedback on compiler errors might be the output of a compiler that is passed on to the student, enabling the student to do exercises without directly using a compiler him or herself. The main reasons for working without a compiler are not having access to the necessary tools and avoiding the difficulty of the compilation process, as experienced by a novice programmer. Test-based AA systems often provide compiler output as feedback, because successful compilation is a prerequisite for executing tests.

Some tools have replaced a standard compiler or interpreter by a more student-friendly alternative. An example is the interpreter used in BIP, which generates extensive error messages in understandable language. Below we give an example of such a message [19]:

```
*20 PRINT "THE INDEX IS; I
          ↑
 SYNTAX ERROR: UNMATCHED QUOTE MARKS -- FOUND NEAR '"THE INDEX IS'
 LINE NOT ACCEPTED (TYPE ? FOR HELP)
```

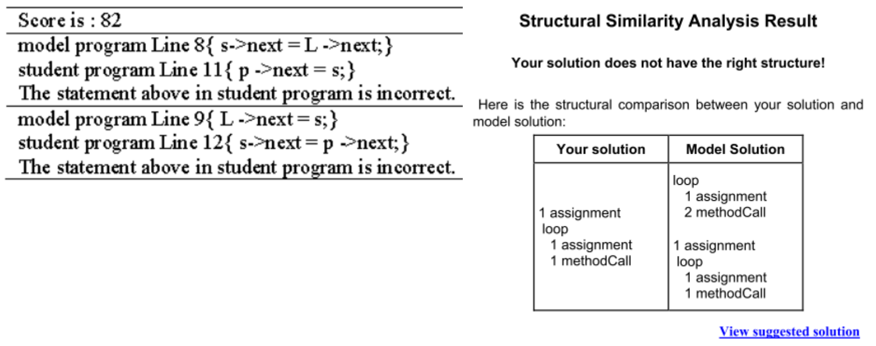
More feedback can be given if the student asks for more help [19]:

```
*?
 '"THE INDEX IS' HAS AN ODD NUMBER OF QUOTE MARKS.
 REMEMBER THAT ALL STRINGS MUST HAVE A QUOTE AT THE BEGINNING AND END.
```

Solution errors (SE)

We have seen many instances of feedback on solution errors. AutoLEP describes the results of matching the student program with several model programs, comparing aspects such as size, structure, and statements. The tool by Singh13 also produces this type [241]:

```
The program requires 1 change:
- In the function computeDeriv, add the base case at the top to return [0]
for len(poly)=1
```



Score is : 82

model program Line 8{ s->next = L ->next;}
student program Line 11{ p ->next = s;}
The statement above in student program is incorrect.

model program Line 9{ L ->next = s;}
student program Line 12{ s->next = p ->next;}
The statement above in student program is incorrect.

Structural Similarity Analysis Result

Your solution does not have the right structure!

Here is the structural comparison between your solution and model solution:

Your solution	Model Solution
1 assignment loop 1 assignment 1 methodCall	loop 1 assignment 2 methodCall 1 assignment loop 1 assignment 1 methodCall

View suggested solution

FIGURE 2.5: Solution error feedback in ANALYSEC on the left (image from [289]) and ELP on the right (image fragment from [261]).

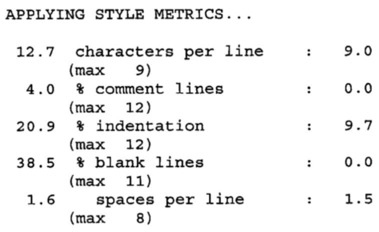
This feedback message provides a numerical value (the number of required changes) and the location of a solution error. The message does not give a description of what the student has done wrong. The suggestion on how to correct the mistake is labelled with KH, which is described next.

ANALYSEC provides detailed feedback on solution errors by identifying incorrect statements and showing the correct statement from a model solution, shown in Figure 2.5. ELP (also in Figure 2.5) matches a model solution with the student solution at a slightly higher level by comparing the structure (the language constructs used, such as loops and assignments) of a student solution to a model solution.

CODE HUNT takes test-based feedback a step further by combining feedback that shows the results of test cases with a hint that points to a line number on which the code needs to be changed (KM feedback). In addition, the student receives information on new features that are useful to solve the problem, and warns the student against features that might not be such a good idea (KH feedback). CODE HUNT uses an approach based on student data: the line number and feature recommendation hints are derived from a large set of previous solutions and unsuccessful attempts.

Style issues (SI)

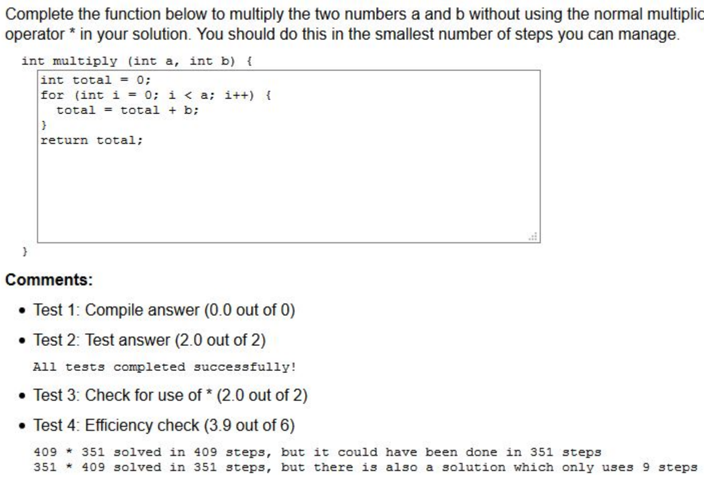
Many teachers consider learning a good programming style important for novice programmers. As an example of feedback on style issues, Figure 2.6



```
APPLYING STYLE METRICS...

 12.7  characters per line    :   9.0
      (max   9)
  4.0  % comment lines        :   0.0
      (max  12)
 20.9  % indentation          :   9.7
      (max  12)
 38.5  % blank lines          :   0.0
      (max  11)
  1.6    spaces per line      :   1.5
      (max   8)
```

FIGURE 2.6: Fragment of style feedback with a score for each issue in JACKSON00 (image from [123]).



Complete the function below to multiply the two numbers a and b without using the normal multiplication operator * in your solution. You should do this in the smallest number of steps you can manage.

```
int multiply (int a, int b) {
  int total = 0;
  for (int i = 0; i < a; i++) {
    total = total + b;
  }
  return total;
}
```

Comments:

- Test 1: Compile answer (0.0 out of 0)
- Test 2: Test answer (2.0 out of 2)

```
All tests completed successfully!
```

- Test 3: Check for use of * (2.0 out of 2)
- Test 4: Efficiency check (3.9 out of 6)

```
409 * 351 solved in 409 steps, but it could have been done in 351 steps
351 * 409 solved in 351 steps, but there is also a solution which only uses 9 steps
```

FIGURE 2.7: Feedback on performance in CHECKPOINT (image fragment from [76]).

shows the feedback generated by the tool of JACKSON00 aimed at formatting and commenting.

Performance issues (PI)

CHECKPOINT, a recent AA system, also provides feedback on performance issues, as can be seen in 'Test 4' in Figure 2.7. NAUR64, one of the earliest systems, checks one particular exercise that lets a student write an algorithm for finding the root of a given function. The system gives performance feedback for each test case, such as 'No convergence after 100 calls' [198].

2.6.4 Knowledge about how to proceed (KH)

Knowing what is wrong with a solution is meaningful information, but in order to learn how to proceed, students need feedback on fixing their mistakes and taking a next step.

Bug-related hints for error correction (EC)

JITS gives feedback on fixing typing errors, such as 'Would you like to replace smu with sum?' [256]. PROUST generates an elaborated error report containing hints on how to correct errors. The following fragment provides such hints [131]:

```
The maximum and the average are undefined if there is no valid input. But
lines 34 and 33 output them anyway. You should always check whether your
code will work when there is no input! This is a common cause of bugs.

You need a test to check that at least one valid data point has been input
before line 30 is executed. The average will bomb when there is no input.
```

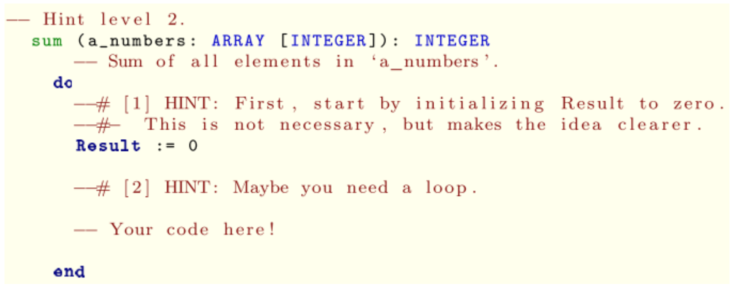
The examples from ELP and ANALYSEC (Figure 2.5) in the KM-SE category also contain the correct code of the solution, therefore we label these tools with KH-EC as well.

CSTUTOR first gives KM-SE feedback in the form of questions that get more specific after each request for help, such as '... Why do you think that the value in the variable 'fahrenheit' does not have this value?' [105]. The authors state that asking questions enables the students to reflect on the problem themselves for some time, as opposed to suggesting a solution instantly. If the student continues to ask for help, the system shows the solution code for the particular problem (KH-EC).

Hints on task-processing steps (TPS)

Hints on task-processing steps can help a student to solve a programming problem step by step. The Prolog tutor HONG04 provides a guided programming phase. If a student asks for help in this phase, the tutor will respond with a hint on how to proceed and generates a template for the student to fill in [116]:

```
You can use a programming technique that processes a list until it is empty
by splitting it into the head and the tail, making a recursive call with
the tail.
```



```
-- Hint level 2.
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do
      --# [1] HINT: First, start by initializing Result to zero.
      --#- This is not necessary, but makes the idea clearer.
      Result := 0

      --# [2] HINT: Maybe you need a loop.

      -- Your code here!

    end
```

FIGURE 2.8: Incremental code revealing feedback in AUTOTEACH (image from [15]).

```
reverse([ ], ⟨arguments⟩).
reverse([H | T], ⟨arguments⟩) :-
    ⟨pre-predicate⟩,
    reverse(T,⟨arguments⟩),
    ⟨post-predicate⟩.
```

Another example can be found in the ASK-ELLE tutor for functional programming. The tool provides a student with multiple strategies to tackle a programming problem [90]:

```
You can proceed in several ways:
- Implement range using the unfoldr function.
- Use the enumeration function from the prelude.
- Use the prelude functions take and iterate.
```

AUTOTEACH is an incremental hint system that gradually reveals larger parts of a model solution to the student, as shown in Figure 2.8. The hints are generated by taking a model solution as input and producing output files for multiple hint levels. An output file shows parts of the solution code and replaces hidden code by matching hint messages. However, these hints are pre-processed, meaning that they do not take into account the student's current work. Unit testing is used as a back-up mechanism to check alternative solutions.

The JIN14 programming tutor incorporates two aspects of programming: planning and coding. The 'guided-planning' component leads the student step-by-step through several predefined stages such as 'variable analysis', 'computation' and 'output', giving hints along the way. The 'assisted-coding' component helps the student with hints on writing code for the different stages.

The hint-generation is based on the technique from Jin12 that uses model programs as input.

Program improvements (IM)

Only a few tools give feedback aimed at program improvements. An example message from Gulwani14 to improve the performance of a program is [100]:

```
Instead of sorting input strings, compare the number of character occurrences
in each string.
```

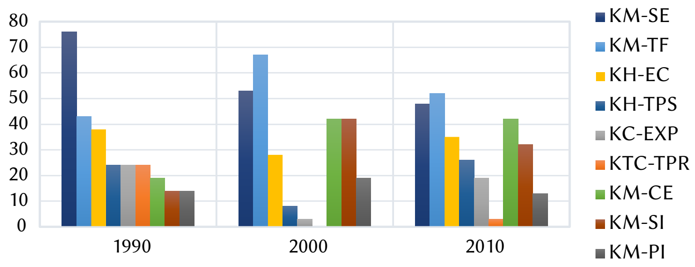
AutoStyle is a tool that gives feedback on programs to make them more concise and readable. It uses historical student data to find a path from a problematic, but functionally correct, solution to a stylistic better solution. As an example, AutoStyle can detect if the functionality of a library method is hand-coded by the student and may suggest using that library method instead.

2.6.5 Knowledge about meta-cognition (KMC)

We have only found one example of KMC. HabiPro provides a 'simulated student' that responds to a solution by checking if a student really knows why an answer is correct.

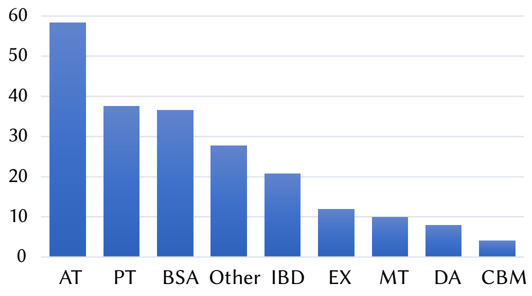
2.6.6 Trends

Figure 2.9 shows the changes of using a particular type of feedback in a tool over the last three decades. A tool is linked to the decade of its median year (as in Figure 2.1). We have omitted the decades before the 1990s because of the low number of tools and do not show the types of feedback that never occur in at least 15% of the tools (KTC-TR, KC-EXA, KH-IM and KMC). The figure shows that in the 1990s 76.2% of the tools provided solution error (KM-SE) feedback, which decreases in the 2000s and 2010s to 52.8% and 48.4%. Test failure feedback (KM-TF) increases in the 2000s from 42.9% to 66.7%, declining to 51.6% in the 2010s. It should be noted, however, that we exclude purely test-based tools, so the actual percentage would be much higher. Multiple types show an increase from the 2000s (KM-CE, KM-SI, KTC-TR), which makes the diversity of feedback types greater in the 21st century.



Feedback type		**1990** %	**2000** %	**2010** %
KM-SE	Solution errors	76.2	52.8	48.4
KM-TF	Test failures	42.9	66.7	51.6
KH-EC	Bug-related hints for error correction	38.1	27.8	35.5
KH-TPS	Task-processing steps	23.8	8.3	25.8
KC-EXP	Explanations on subject matter	23.8	2.8	19.4
KTC-TPR	Hints on task-processing rules	23.8	0.0	3.2
KM-CE	Compiler errors	19.0	41.7	41.9
KM-SI	Style issues	14.3	41.7	32.3
KM-PI	Performance issues	14.3	19.4	12.9

FIGURE 2.9: Percentage of tools with feedback type in the 1990s (n=21), 2000s (n=36) and 2010s (n=31), omitting types that never occur in at least 15% of the tools. The legend and table have the same order as the bars in the chart.

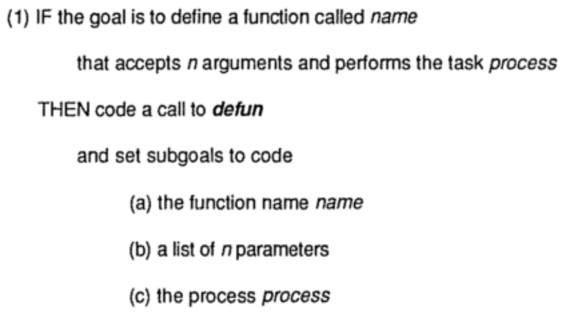


		All (n=101)		Class 2 (n=24)		Class 3 (n=77)	
Technique		Count	%	Count	%	Count	%
AT	Automated testing	59	58.4	7	29.2	52	67.5
PT	Program transformations	38	37.6	8	33.3	30	39.0
BSA	Basic static analysis	37	36.6	7	29.2	30	39.0
Other		28	27.7	6	25.0	22	28.6
IBD	Intention-based diagnosis	21	20.8	3	12.5	18	23.4
EX	External tools	12	11.9	0	0.0	12	15.6
MT	Model tracing	10	9.9	6	25.0	4	5.2
DA	Data analysis	8	7.9	1	4.2	7	9.1
CBM	Constraint-based modelling	4	4.0	2	8.3	2	2.6

FIGURE 2.10: Number of tools (count) and percentage of tools that employ a technique, for all tools, and subdivided by exercise class.

2.7 Technique (RQ2)

This section describes the results of the second research question: 'which techniques are used to generate the feedback?' Figure 2.10 shows for each technique the percentage of tools that use it. Even after omitting purely test-based tools, automated testing is still the technique used the most (58.4%) in tools that generate feedback. After that, 37.6% use program transformations and 36.6% of all tools use static analysis. Intention-based diagnosis is used in 20.8% and 11.9% of the tools use an external tool. We have found that 9.9% of the tools use model tracing, 7.9% use data analysis and only 4.0% use constraint-based modelling. In 27.7% of the tools other techniques were found. Figure 2.10 also shows the differences between tools of class 2 and class 3. Class 3 tools use more automated testing, external tools, intention-based diagnosis and data analysis than class 2 tools. Class 2 tools more often use model tracing and



```
(1) IF the goal is to define a function called *name*
        that accepts *n* arguments and performs the task *process*
    THEN code a call to **defun**
        and set subgoals to code
            (a) the function name *name*
            (b) a list of *n* parameters
            (c) the process *process*
```

FIGURE 2.11: A production rule from the LISP TUTOR (image from [57]).

constraint-based modelling. In the next subsections we expand on the different techniques and explain how they are used.

2.7.1 General ITS techniques

Model tracing (MT)

The LISP TUTOR is a classic example of a model tracing system. An example of a production rule (rephrased in English) used in this tutor is shown in Figure 2.11. Whereas classic tools use a production system for model tracing, some tools use a slightly different approach. As an example, ASK-ELLE and KEUNING14 generate *programming strategies* derived from model solutions that are used to check where the student is in his or her programming process, and give hints on what to do next. We found one instance of example-tracing used in the JAVA SENSEI tool that offers class 2 exercises. Example-tracing tutors follow the steps that a student takes based on 'generalised examples of problem-solving behaviour' [9], making it easier to create ITSs. Although example-tracing is a different tutoring paradigm, we categorise it under the MT label.

Constraint-based modelling (CBM)

INCOM uses Constraint-based modelling. The authors of INCOM argue that CBM has its limitations in the domain of programming because of the large solution space for programming problems [162]. They have designed a 'weighted constraint-based model', consisting of a semantic table, a set of constraints, constraint weights (to indicate the importance of a particular constraint), and transformation rules. The authors show that this model can recognise student intentions in a much larger number of solutions compared to the standard CBM approach.

J-LATTE, the Java Language Acquisition Tile Tutoring Environment, is a constraint-based ITS for learning Java. The tutor presents students with simple programming exercises that can be solved in two modes. In Concept mode, a student selects predefined programming artefacts defined at a higher level (e.g. declarations, return statements and loops) from the user interface and combines them to create the structure of the solution. After selecting a concept, the accompanying code can be entered in the Coding mode. Using constraints does not force the student to follow a predetermined path. J-LATTE uses constraints to represent domain knowledge describing the syntactic, semantic and style-related features of the solution. An example of a semantic constraint is:

```
(sum-of-function-over-a-range :range
    (:from (method-arg :name "startNum")
        :to (method-arg :name "endNum"))
    :function square)
```

This specification states that the 'sum-of-function-over-a-range' pattern should be used (which could be any kind of loop) with a specific lower and upper limit of the range and a call to a square-function. Feedback can be requested by the student at any time during the exercise. The system presents error messages related to constraints that are violated, such as 'You should be initialising the loop-variable to the beginning of the range you are looping over' [115].

Data analysis (DA)

Using large sets of historical student data to generate hints is a recent development that has already produced some promising results. ITAP (Intelligent Teaching Assistant for Programming) is a data-driven tutor for programming in Python. It creates a solution space graph with (intermediate) program states as nodes, in which directed edges represent next steps. ITAP matches a student solution with a state in the graph, constructs a path to a correct solution and generates hints based on the steps of the path.

The Jin12 tool also uses a set of student solutions and creates Markov Decision Processes to generate feedback to correct or finish an incorrect or incomplete program. Lazar14 inspects traces of students solving Prolog programming problems to collect common line edits. This information is then used to find a sequence of line edits that transform an incorrect program into a correct one, preferring the shorter sequences.

2.7.2 Domain-specific techniques for programming

Dynamic code analysis using automated testing (AT)

AT is often implemented by running a program and comparing its output to the expected output. Some tools integrate professional testing tools, such as JUnit[2] for unit testing and QuickCheck [53] for property-based testing. Testing is frequently used as a 'last resort' if the tool cannot recognise in any other way what the student is doing. Striewe11's technique offers a solution to the problem that students may have difficulty identifying the cause of a failed test case. This is particularly relevant for blackbox testing, which only shows a difference between expected and actual outcome. Striewe11's tool generates run-time traces for the execution of test cases using debugging technology. Students can inspect the traces to find out where unexpected behaviour occurred. For specific issues the tool automatically analyses the trace to provide even more help. For example, assertion checking is used to point to the location of an error, and automated comparison of the student trace to the trace from a model program can be used to detect abnormal behaviour.

Test My Code (TMC) counts bytecode instructions to assess the performance of algorithms. The authors illustrate this with an exercise that requires the student to write an implementation for the Fibonacci sequence that runs in linear time. The resulting calculations can be shown to the student. Counting bytecode instructions as an alternative to simply measuring running time prevents unreliable measurements caused by busy assessment servers.

Test cases may be predefined by the instructor, generated randomly, or have to be supplied by the student him or herself. We have also noticed the use of *reflection* in multiple tools, a technique that can be used to dynamically inspect and execute code. AutoGrader is a lightweight framework that uses Java reflection to execute tests for grading and creating feedback reports.

Feedback on performance issues can be done by profiling, a dynamic program analysis method. PRAM uses profiling to measure several aspects related to complexity in Prolog programs, such as the average number of calls and the

[2]`www.junit.org`

percentage of backtracking. The system compares the results to the results of the profiling of a model program and presents the issues found together with a mark to the student.

Finally, code coverage tools can be used to identify unnecessary statements, as seen in ProgTest.

Basic static analysis (BSA)

Some tools use static analysis for calculating metrics, such as cyclomatic complexity or the number of comments. The GAME-2 tool performs static analysis by examining comments in a solution. The analysis identifies code that is commented out as ‘artificial’ comments, and identifies ‘meaningful’ comments by looking at the ratio of nouns and conjunctions compared to the total word count. InSTEP looks for common errors in code, such as using ‘=’ instead of ‘==’ in a loop condition, or common mistakes in loop counters, and provides appropriate feedback accordingly.

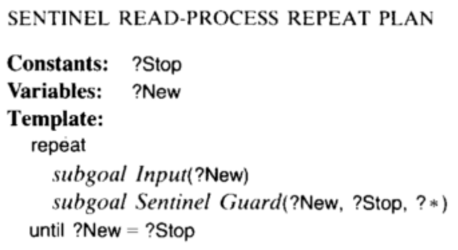
Program transformations (PT)

Transformations are often used together with static code analysis to match a student program with a model program. The normalisation technique used in SIPLeS-II is a notable contribution. The authors have identified 13 ‘semantics-preserving variations’ (SPVs) found in code. Some of these SPVs are handled using transformations that change the computational behaviours (operational semantics) of a program while preserving the computational results (computational semantics). As an example, the ‘different control structures’ SPV is handled by transformations that standardise control structures, and the ‘different redundant statements’ SPV by dead code removal. As a result, a larger number of student programs can be recognised.

Migration is applied in INTELLITUTOR, which uses the abstract language AL for internal representation. Pascal and C programs are translated into AL to eliminate language-specific details. After that, the system performs some normalisations on the AL-code.

Synthesis, transformation to a lower level such as bytecode, is another program transformation technique. We have not found this technique in tools other than compilers, and the external tool FindBugs.[3] FindBugs translates

[3]`findbugs.sourceforge.net`


SENTINEL READ-PROCESS REPEAT PLAN

```
Constants:  ?Stop
Variables:  ?New
Template:
  repeat
    subgoal Input(?New)
    subgoal Sentinel Guard(?New, ?Stop, ?*)
  until ?New = ?Stop
```

Figure 2.12: Simplified plan in PROUST (image from [131]).

Java code into bytecode, and performs static analysis on this bytecode to identify potential bugs.

Intention-based diagnosis (IBD)

The term intention-based diagnosis was introduced by Johnson and Soloway for their tutor PROUST [132]. PROUST has a knowledge base of programming plans, which are implementations of programming goals. One programming problem may have different goal decompositions. Figure 2.12 shows the simplified plan for the ‘Sentinel-Controlled Input Sequence goal’. PROUST tries to recognise these plans, including erroneous plans, in the submitted code and reports bugs. Later, Sack proposed an improved system (ProgramCritic) based on MicroPROUST (a simpler version of PROUST) by simplifying its design and fixing its weaknesses, including the elimination of the distinction between goals and plans.

MENO-II is a predecessor to PROUST, which ‘failed miserably’ [133] according to the authors. The tool uses a library of 18 common bug templates for simple programs containing a loop, derived from a set of student programs. The bug templates are linked to possible misconceptions, but are not problem-specific. The tool identifies plans in a student program, matches them against the bug templates (see Figure 2.13) and reports errors, misconceptions and possible solutions. This approach failed to recognise many mistakes in exercises because the bug templates were too general and lacked information on how program components worked together to solve a particular problem. A better solution was needed to support the great variety in student programs.

Some of the IBD systems only work for a limited set of predefined exercises, such as the classic Rainfall problem for Proust.

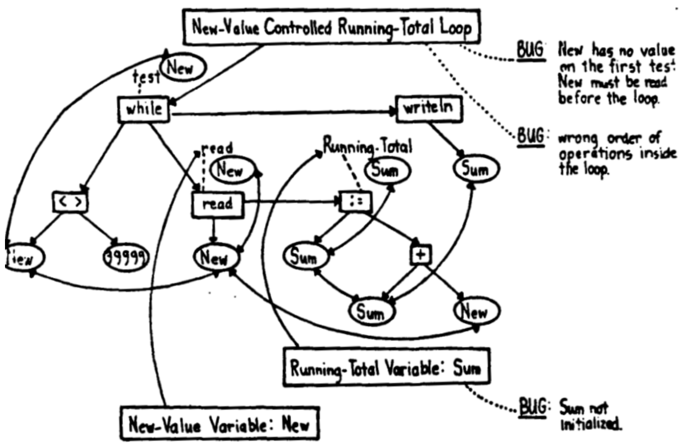

New-Value Controlled Running-Total Loop
BUG New has no value on the first test. New must be read before the loop.
test
New
while
writeln
BUG: wrong order of operations inside the loop.
read
New
Running-Total
Sum
Sum
< >
read
:=
New
99999
New
Sum
+
Sum
New
Running-Total Variable: Sum
BUG: Sum not initialized.
New-Value Variable: New

FIGURE 2.13: Bugs located in a parse tree in MENO-II (image from [244]).

External tools (EX)

We have found a number of static analysis tools, for example CheckStyle[4] for checking code conventions, FindBugs [117][5] for finding bugs, and PMD[6] for detecting bad coding practices. These tools do not specifically focus on novice programmers and may produce output that is difficult to understand for beginners. The tools are often configured to provide a limited set of output messages so as not to overwhelm and confuse the learner.

2.7.3 Other techniques

Of all tools, 27.7% use techniques that do not fit one of our labels, which are often AI techniques.

As an example, the ProPL tutor uses natural language processing to engage in a dialogue with the student to practice planning and program design. Human tutoring is a proven technique for effective learning. The tutor mimics the conversation that a human tutor would have with a student using natural language processing. ProPL uses a dialogue management system that requires a substantial amount of input to construct a tutor for a programming problem.

[4] `checkstyle.sourceforge.net`
[5] `findbugs.sourceforge.net`
[6] `pmd.github.io`

DATLAB employs machine learning techniques to classify student errors and generate corresponding feedback. The author uses a neural network to 'learn relationships corresponding to trained error categories, and apply these relationships to unseen data' [176].

The COALA system uses fuzzy logic to analyse similarity to a model solution. The system calculates the score for a set of metrics (such as cyclomatic complexity, lines of code, number of parameters) for both the student program and model program, fuzzifies the scores, and calculates feedback on style issues and how to improve them using fuzzy rules such as:

```
IF (control complexity is low AND cyclomatic complexity is high)
THEN show message ('The solution can be done easily by reviewing
                    the conditions and deleting some bifurcation')
```

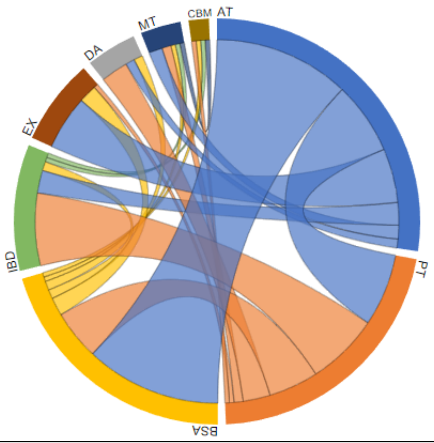
In addition, the system uses test cases to determine the correctness of the solution. The instructor has to provide a model program and a set of test cases, but also the region values for each metric that are used to generate the fuzzy sets.

The authors of the SINGH13 tool use program synthesis techniques to generate feedback on solutions for Python programming problems, such as [241]:

```
The program requires 3 changes:
- In the return statement return deriv in line 5, replace deriv by [0].
- In the comparison expression (poly[e] == 0) in line 7, change (poly[e] ==
0) to False.
- In the expression range(0, len(poly)) in line 6, increment 0 by 1.
```

The problem author writes a model solution and an error model. The error model consists of a set of correction rules that solve a mistake together with appropriate feedback messages. All possible programs based on these rules applied to the student's solution are then searched to find the one that most closely matches the model solution. This is done by translating the program with the correction rules into a Sketch program. Sketch is a software synthesis tool that can complete a partial code implementation to make it behave like a given specification. The Sketch synthesiser finds the solution and feedback is generated based on the applied correction rules. A limitation of the tool is its inability to deal with student programs that have large conceptual errors.

The technique employed in GULWANI14 focuses on performance issues in student solutions. They observe that the percentage of solutions that are algorithmically inefficient is around 60% and can go up to 90% for their example



Combination	**Count**
Automated testing + Basic static analysis	30
Automated testing + Program transformations	16
Intention-based diagnosis + Program transformations	16
Automated testing + External tools	12
Program transformations + Basic static analysis	11
Data analysis + Program transformations	6

FIGURE 2.14: Chord diagram showing the frequency of combinations of two techniques in the 101 tools through the width of a connection, omitting 'Other'. The table shows all combinations that occur more than 5 times, also omitting 'Other'.

problem of checking if two strings are anagrams of each other. In their solution, an instructor annotates a program by specifying key values that are computed during the execution of the program, thereby ignoring irrelevant implementation differences. A new language construct called 'observe' is introduced to specify these values. The execution trace of a student program is compared to the traces of annotated programs and for the matching program an appropriate feedback message is shown. Although the instructor has a substantial task to annotate programs with different algorithmic approaches, the authors show that this is worth the effort for use in large-scale systems.

2.7.4 Combining techniques

Figure 2.14 shows how often combinations of two techniques are used in one tool. Because our review excludes tools that only use automated testing (AT) and give feedback in the form of test case results, this technique is the one combined most with other techniques. AT is most often (in 30 tools) combined with basic static analysis (BSA). After that, combinations with program transformations (PT) are seen most often (in 16 tools), followed by combinations with external tools (EX) in 12 tools. Intention-based diagnosis (IBD) and PT are used in the same tool in 16 cases. BSA is also often combined with PT: we see this in 11 tools.

As a concrete example, the assessment technique of Vujošević-Janičić13 combines multiple techniques (testing, automated bug finding and control flow graph similarity) to overcome the shortcomings of only using a single technique. The authors state that software verification tools can pick up missing bugs not covered by a test case. Moreover, structural issues such as modularity cannot be intercepted by testing and verification only, so the similarity to a model program should also be incorporated in an assessment. In their solution, the C–code of students is translated into the LLVM intermediate language[7] and static verification is performed on the intermediate representation. The program is also translated into a control flow graph (CFG) and compared to the CFG of model programs, calculating the degree of similarity. The results are used to calculate a grade, but can also be used to give feedback on bugs and similarity to a model solution.

The ITS designed by Dadic11 uses different techniques for different types of students. The system uses model tracing to force ‘stoppers’, students that give up quickly, to write a program in a predefined order. ‘Movers’, students that will keep trying even when they are struggling, are allowed more freedom in their problem-solving process, for which the system uses a constraint-based technique to check their solution.

2.7.5 Trends

Figure 2.15 shows the techniques that have been used over the years. Intention based diagnosis emerged around the 1990s, but has been used less often in the 21st century. Automated testing is most seen in the 2000s, but remains popular

[7] `www.llvm.org/`

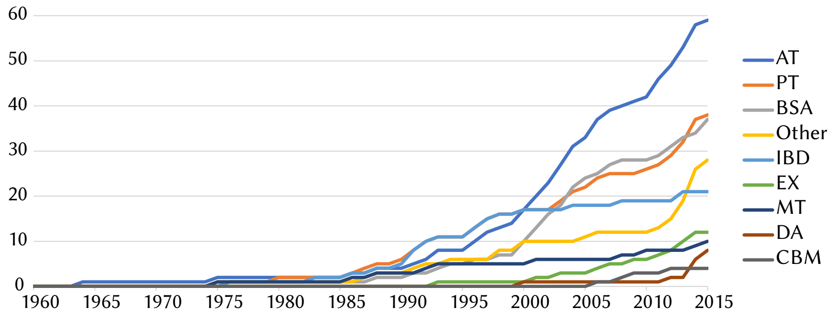


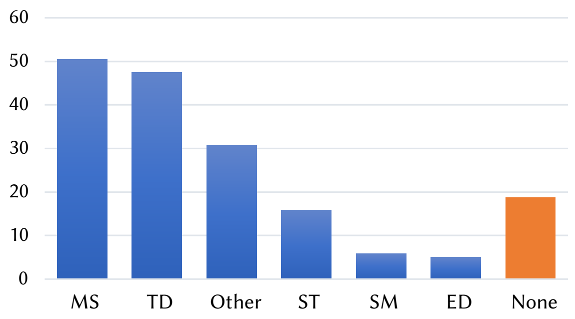
FIGURE 2.15: Cumulative number of tools with technique over the years, based on the earliest paper on a tool from our data set. The legend has the same order as the endings of the lines in the chart.

in the 2010s. The same applies to basic static analysis. In the 2010s various new techniques, such as data analysis, make their appearance.

2.8 Adaptability (RQ3)

In this section we describe the results of the third research question: 'how can the tool be adapted by teachers, to create exercises and to influence the feedback?' To answer this question, we have categorised the different types of input that teachers can provide. We require that the tool does not need to be recompiled, and not too much effort or specialised knowledge is needed. Some tools enable authors to write complex error models or rules and constraints to specify correct solutions. However, we consider these inputs to be too specialised and time-consuming for a teacher who quickly wants to add a new exercise.

Figure 2.16 shows for each input type the percentage of tools that use it. We have found that model solutions are used most, in 50.5% of the tools. After that, 47.5% of the tools use test data. Of all tools, 15.8% offer solution templates, and in 5.0% error data can be specified. We have found the use of a student model for generating feedback on solutions in 5.9% of the tools. In 30.7% of the tools we have found other types of input. A total of 18.8% of the tools do not use any input type. The feedback these tools generate is often based on hard-coded knowledge bases (e.g. HE94), or it is just too time-consuming to add an exercise (e.g. one person-week for BRIDGE).



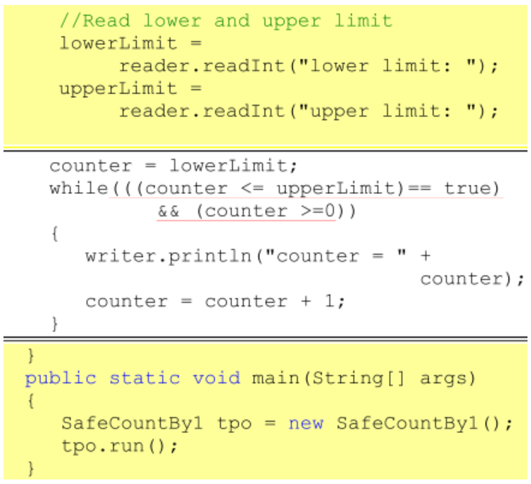
Input type		All (n=101) Count	%	Class 2 (n=24) Count	%	Class 3 (n=77) Count	%
MS	Model solutions	51	50.5	8	33.3	43	55.8
TD	Test data	48	47.5	4	16.7	44	57.1
Other		31	30.7	6	25.0	25	32.5
ST	Solution templates	16	15.8	7	29.2	9	11.7
SM	Student model	6	5.9	3	12.5	3	3.9
ED	Error data	5	5.0	0	0.0	5	6.5
None		19	18.8	9	37.5	10	13.0

FIGURE 2.16: Number of tools (count) and percentage of tools with input type, for all tools, and subdivided by exercise class.

Class 2 tools more often do not use any input. Class 3 tools use more test data and model solutions than class 2 tools. Class 2 tools use more solution templates and input from a student model. The next subsections expand on the different input types and explain how they are used.

2.8.1 Solution templates (ST)

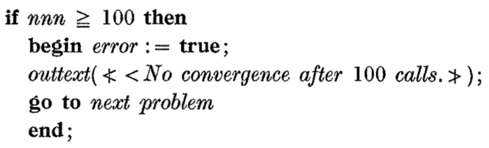
Solution templates are often used for class 2 exercises to restrict the student's freedom in solving a particular problem. An example is the ELP system shown in Figure 2.17, in which students fill in gaps (the white area in the middle) in a Java template with predefined code fragments (the yellow areas surrounding the white gap). Solution templates found in test-based assessment tools are often project skeletons, or an interface definition for a data structure that prescribes the names of functions, parameters and return values.



```
    //Read lower and upper limit
    lowerLimit =
        reader.readInt("lower limit: ");
    upperLimit =
        reader.readInt("upper limit: ");

   counter = lowerLimit;
   while(((counter <= upperLimit)== true)
            && (counter >=0))
   {
     writer.println("counter = " +
                                counter);
     counter = counter + 1;
   }
 }
 public static void main(String[] args)
 {
    SafeCountBy1 tpo = new SafeCountBy1();
    tpo.run();
 }
```

FIGURE 2.17: Fill-in-the-gap exercise in ELP (image fragment from [261]).



```
if nnn ≧ 100 then
  begin error := true;
  outtext(∗ <No convergence after 100 calls.∗);
  go to next problem
  end;
```

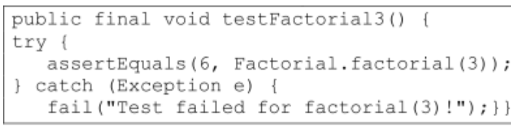
FIGURE 2.18: Test code in NAUR64 (image from [198]).

2.8.2 Model solutions (MS)

Correct solutions to a programming exercise are used as input in many tools. In dynamic analysis they are used for running test cases to generate the expected output. In this case a single correct solution will suffice. In static analysis the structure of a correct solution is compared to the structure of a student solution. To recognise more than one solution variant, some tools accept multiple solutions that each represent a different algorithm to solve the problem.

2.8.3 Test data (TD)

In older tools testing is done in scripts, for example in NAUR64. Figure 2.18 shows a small fragment of its test code. Newer systems accept unit tests as input, such as in Figure 2.19 that shows one of the unit tests for the factorial function for WEBWORK-JAG. The WEBWORK-JAG system also processes Java



```
public final void testFactorial3() {
try {
   assertEquals(6, Factorial.factorial(3));
} catch (Exception e) {
   fail("Test failed for factorial(3)!");}}
```

Figure 2.19: Unit test used in WebWork-JAG (image from [94]).

reflection code to test the signature of methods. Other tools require the teacher to simply provide input/output pairs.

2.8.4 Error data (ED)

A small number of tools accept input containing some form of error data. As an example, Talus, a debugger for Lisp programs, uses both model solutions and buggy solutions. If a student program is matched with a buggy solution, corresponding feedback addressing the misconception is presented to the student.

2.8.5 Other

In CourseMarker teachers can configure how much feedback should be given. Some tools let a teacher define custom feedback messages. In Ask-Elle model solutions can be annotated with feedback messages [90]:

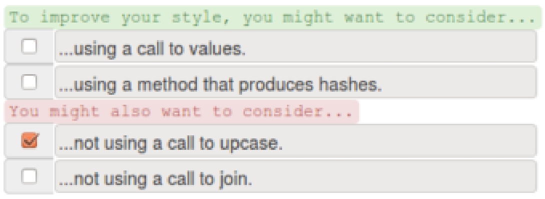
```
range x y = {-# FEEDBACK Note... #-}
   take (y-x+1) $ iterate (+1) x
```

These messages appear if the student asks for help at a specific stage during problem solving.

AutoStyle provides a user interface for an instructor that visualises paths consisting of subsequent submissions and their differences. The instructor can influence the hints by adjusting thresholds so the changes between submissions become larger or smaller, and can mark hints as helpful or not, as shown in Figure 2.20.

We found a system that *generates* exercises instead of having instructors author them (Shimic), although the resulting exercises seem rather contrived (‘Define private long integer method ‘n’ with integer argument named ‘n’, and which returns: ‘1022’ if ‘n’ is ‘8142’; ‘963’ if ‘n’ is ‘1261’; ’etc.).

Testovid is a platform and language-independent tool, for which teachers have to write Apache Ant scripts to build student solutions and run a set of


To improve your style, you might want to consider...

☐ ...using a call to values.

☐ ...using a method that produces hashes.

You might also want to consider...

✓ ...not using a call to upcase.

☐ ...not using a call to join.

FIGURE 2.20: Instructor UI for AUTOSTYLE (image fragment from [193]).

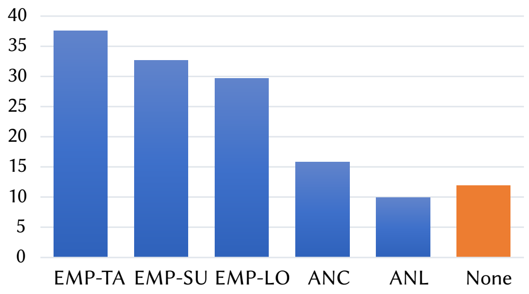
(testing, style checking) tools on the programs. Writing these scripts is a time investment, but they can be reused and adapted later for different exercises and languages.

In AUTOTEACH a model solution is gradually revealed (see Figure 2.8 in Section 2.6.4) The tool offers the teacher several options to customise the hints using 'meta-commands', which are special directives embedded in comments. Using these commands the teacher can specify custom hint messages and control which parts of the code should (not) be revealed. The system includes a default revealing mechanism that starts with showing only the basic structure of the code (classes, functions) and ending at the level that almost shows all code except for instructions inside conditional statements and loops.

2.9 Quality (RQ4)

This section describes the results of the fourth research question: 'what is known about the quality and effectiveness of the feedback or tool?' Figure 2.21 shows for each evaluation method the percentage of tools for which it has been used, and the number of evaluation methods used per tool. We have found that 15.8% of the tools we examined only provide anecdotal evidence on the success of a tool, and for 11.9% of the tools we have not found any evaluation at all. Of all tools, 9.9% have been assessed by an analytical method and 71.3% by some form of empirical assessment, of which technical analysis is the largest group with 37.6% of the tools. Not including anecdotal assessment, the majority of the tools have been evaluated by one method (45.5%), 26.7% of the tools have been evaluated by at least two methods, 7.9% by 3 methods and 2.0% by 4 methods.

In the next subsections we give some examples and observations regarding tool evaluation.



Evaluation method		%
EMP-TA	Empirical – Technical analysis	37.6
EMP-SU	Empirical – Surveys	32.7
EMP-LO	Empirical – Learning outcome evaluations	29.7
ANC	Anecdotal assessment	15.8
ANL	Analytical assessment	9.9
None		11.9

Combination	%
1 method, not ANC	45.5
2 methods	16.8
3 methods	7.9
4 methods	2.0

FIGURE 2.21: Percentage of tools (n=101) with evaluation method, and combinations of evaluation methods.

2.9.1 Analytical (ANL)

We found a small number of tools that are based on validated approaches, such as a learning theory. The LISP TUTOR, GRACE and JITS are based on the ACT-R cognitive architecture [13]. In ACT-R procedural knowledge is defined as a set of production rules that model human behaviour in solving particular problems. CHANG00 is based on the completion strategy for learning programming by Van Merriënboer [267]. This strategy is based on exercises in which (incomplete) model programs written by an expert should be completed, extended or modified by a novice programmer.

2.9.2 Empirical assessment

Empirical assessment analyses qualitative data or quantitative data. We distinguish three types.

Looking at the learning outcome (EMP-LO)

The nature of these experiments varies greatly. For example, PROPL was evaluated in an experiment with 25 students. The students were either in the control group that used a simple, alternative learning strategy, or in the group that used ProPL. Students both did a pre-test and a post-test to assess the changes in the scores. Some tools have used a less extensive method, for instance by omitting a control group, and comparing the pass rates from the year the tool was used against previous years. The test group size also varies greatly.

Evaluating the programming game Code Hunt was done by tracking the activity of 407 users over two weeks. Some users got all hints, some users got occasional hints, and some users never got any hints. The results show that the users who got hints played longer and won more levels than the users who did not get any hints. In addition, users who got occasional hints played longer than users that always got hints.

Student and teacher surveys (EMP-SU)

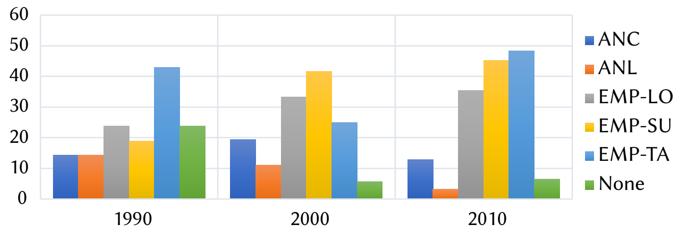
We have found that the number of responses in some cases is very low, or is not even mentioned. Some papers mention a survey, but do not show an analysis of the results, in which case we do not assign this label.

Technical analysis (EMP-TA)

SIPLeS-II was assessed using a set of 525 student programs, measuring the number of correctly recognised solutions, the time needed for the analysis, and a comparison to the analysis of a human tutor. In some cases, this type of analysis is done for a large number of programs, only counting the number of recognised programs. In other cases, researchers thoroughly analyse the content of generated hints, often for a smaller set of programs because of the large amount of work involved.

The authors of Jin12 analysed 200 student submissions for one simple programming problem ('calculate the pay for mowing the lawn around a house'). In a first experiment the 37 correct solutions from this set were used to generate hints for 16 randomly selected correct submissions. The hints were appropriate for 14 of these 16 submissions (87.5%). In their second experiment, the authors manually selected a similar correct solution, 'which was not necessarily the best match' for 15 randomly selected incorrect submissions. The generated hints based on their differences were meaningful for 10 (66.6%) of the 15 incorrect submissions. The authors propose concrete ways to increase this percentage.

The technique from Jin12 was used in the Jin14 tutor, which has a 'guided-planning' and an 'assisted-coding' component. The tutor was evaluated by



Evaluation method		**1990** %	**2000** %	**2010** %
ANC	Anecdotal assessment	14.3	19.4	12.9
ANL	Analytical assessment	14.3	11.1	3.2
EMP-LO	Empirical - Learning outcome evaluations	23.8	33.3	35.5
EMP-SU	Empirical - Surveys	19.0	41.7	45.2
EMP-TA	Empirical - Technical analysis	42.9	25.0	48.4
None		23.8	5.6	6.5

FIGURE 2.22: Percentage of tools with evaluation method in the 1990s (n=21), 2000s (n=36) and 2010s (n=31). The legend and table have the same order as the bars in the chart.

comparing it to three other variants: 'coding-only', 'planning-only' and alternating between variants. They conducted an experiment with 85 students who did a pre-test, worked with the tutor for some time and then completed a post-test. The group that did the 'planning-coding' variant had the largest increase in learning.

Other

Another way to evaluate a tool is to compare it to other related tools based on a set of criteria or functions (e.g. JAVA SENSEI). The feedback provided by the FIT JAVA TUTOR was assessed for their appropriateness by experts. The authors of this tool also did a 'Wizard of Oz' experiment to assess if the example-based feedback helped students to achieve higher scores on their work. Instead of the ITS, a human expert provided the feedback that students requested. Some tools were evaluated as part of a course that used the tool, or in the context of a specific didactic method, making it more difficult to isolate and measure the effect of the tool itself.

2.9.3 Trends

Figure 2.22 shows how the distribution of evaluation methods changes over the last three decades. The use of surveys shows a steady increase from 19.0% in the 1990s, 41.7% in the 2000s and 45.2% in the 2010s. Despite the relapse to 25.0% in the 2000s, technical analysis is still the method seen most in this decade with 48.4% in the 2010s, closely followed by surveys and learning outcome evaluations. The number of tools for which we have not found any type of evaluation drops from 23.8% to around 6%, but the percentage of tools for which we only found anecdotal evidence remains nearly the same. It should be noted, however, that some evaluations of tools from this decade are still to be published, so this figure only gives a general impression.

2.10 Discussion

In this review we intend to find an answer to our research question concerning feedback generation for programming exercises: ‘what is the nature of the feedback, how is it generated, can a teacher adapt the feedback, and what can we say about its quality and effect?’ In this section we take a closer look at the answers we have found to the four sub-questions, discuss the relation between these answers, and give some observations and recommendations that follow from our review.

2.10.1 Feedback types

Looking at the type of feedback given by tools, which we investigated in the first research question, we have found that feedback about mistakes is the largest type used in tools. In the first iteration of our review we found that feedback on test failures was the largest subtype [145]. After deciding to omit purely test-based tools in this final review, this subtype was still the second largest. Generating feedback based on tests is a useful way to point out errors in student solutions and emphasizes the importance of testing to students. It is therefore a valuable technique, and relatively easy to implement using existing test frameworks. Most tools that use automated testing support class 3 exercises, because black-box testing does not require using a specific algorithm or design process. The only aspect that matters is whether the output meets the requirements of the exercise. However, only giving test-based feedback will not in all cases help a student to fix an incorrect program.

The largest feedback subtype from our review is solution error feedback. This type of feedback can be implemented with varying depth and detail. To really help a student, just pointing at an error may not help. Feedback on how to proceed is necessary to both fix problems and to progress towards a solution when stuck. This type of feedback is mostly seen in the form of error correction hints, and much less as hints on task-processing steps or on program improvements. This is especially the case for class 3 tools supporting exercises that can be solved by multiple (variants of) strategies. Class 2 tools provide more procedural support, but a disadvantage is that they do not support alternative solution strategies, and may restrict a student in his or her problem-solving process. Finally, the very low percentage of tools that give code examples based on the student's actions is unfortunate, because studying examples has proven to be an effective way of learning. Looking at the changes over the decades, feedback on test failures and solution errors remain the most common types. However, the diversity of feedback types in the current century has increased, which is a positive development.

Many of the tools we have investigated are AA tools, which are often used for marking large numbers of solutions. If marking is the only purpose, one could conclude that more elaborate feedback is not necessary. However, if we want our students to learn from their mistakes, a single mark or a basic list of errors only is not sufficient. Moreover, we have noticed that many authors of AA tools claim that the intention of the feedback their tool generates is student learning.

Suzuki et al. [254] analysed posts on a programming discussion forum to identify the different types of hints that teachers give. The authors found 10 types, five of which could be generated using program synthesis techniques: transformation hints (how to correct mistakes), location hints (where to correct mistakes), data hints (the expected value or type of a variable at some location), behaviour hints (describing dynamic program behaviour), and example hints (explaining how code should work by showing input and output examples). The authors found that hints teachers give often focus on *why* the student's code failed, and that teachers generally do not provide exact fixes.

There is a vast amount of studies that identify and classify difficulties and errors that novice programmers encounter in their learning process. Qian and Lehman [219] have conducted a literature review of such studies of student difficulties in introductory programming. The authors distinguish between difficulties in syntactic knowledge, conceptual knowledge (not knowing how a programming construct works or how code executes) and strategic knowledge (using code to solve problems), and describe several approaches and tools to support students in dealing with these difficulties. Considering our classification of feedback, misconceptions regarding syntactic knowledge can be addressed by compiler error feedback. Conceptual knowledge can be addressed by the two KC (knowledge about concepts) feedback types: explanations on subject matter and examples illustrating concepts. Strategic knowledge can be addressed by feedback on solution errors, all feedback on knowledge about how to proceed and knowledge on task constraints. Feedback on test failures, style and performance can also help.

Offering more variety in feedback type in one tool than is currently the case, and carefully considering the level of detail in a generated feedback message would contribute to giving more effective feedback, similar to the feedback of a human tutor.

2.10.2 Feedback generation techniques

By answering our second research question we came across many techniques for generating automated feedback, which we can relate to the different feedback content types. Well-known ITS techniques such as model tracing and constraint checking are used in some tools, but not on a very large scale. Model tracing is strongly related to hints in the 'knowledge on how to proceed' category, often producing next-step hints. Constraint checking and intention-based diagnosis mainly produce hints on solution errors and/or error correction. Automated testing is often combined with static analysis of the abstract syntax tree, sometimes incorporating transformations to simplify its structure or to ignore irrelevant details. We observe that automated testing generates test-based feedback. However, it is also used as a mechanism for recognising a solution as correct or incorrect before further analysing a program using another technique. For generating style-related hints, basic static analysis or external tools (that might also perform static analysis) are used.

An upcoming trend is the use of data-driven technology to base feedback on historical student data, such as paths students have taken and programming mistakes they have made. Hint generation with data-driven techniques produces feedback on solution errors, correcting these errors, taking a next step and providing the student with related examples. Feedback generation based on recognising plans and errors from a knowledge base is less popular nowadays. Data-driven techniques, however, require the presence of large sets of data, complicating the authoring of new exercises. We must also find an answer to the question if students learn the right problem-solving strategies if hints are only based on other students' past behaviour. Perhaps combining the successful techniques of the past with new (data-driven) enhancements will bring new opportunities.

2.10.3 Tool adjustability

The 2014 working group report of the ITiCSE conference by Brusilovsky et al. [44] discusses the adoption of 'smart learning content' (SLC) in computer science courses on a larger scale. SLC tools are interactive tools offering, for instance, visualisation, simulation, automated assessment, coding support, or problem-solving support, in which feedback plays a prominent role to help students in their learning. One of the main problems the authors found among teachers is not being able to customise tools to their own needs, and that tools do not match their own teaching methods. The report proposes an architecture that promotes flexible integration and customisation of SLC tools.

In our review, we found that tools use various dynamic and static analysis techniques. More sophisticated techniques, such as model tracing and intention-based diagnosis, appear to complicate adding new exercises and adjusting the tool. However, the question whether or not a tool can be adapted easily is difficult to answer, and depends on the amount and complexity of the input. We have found that very few papers explicitly describe this, or even address the role of the teacher. In the latter case we assume that there is no such role and the tool can only be adjusted by the developers. When a publication does describe how an exercise can be added, it is often not clear how difficult this is. Some publications mention the amount of time necessary to add an exercise, such as one person-week for Bridge. We conclude that teachers cannot easily adapt tools to their own needs, except for test-based AA systems.

2.10.4 Tool evaluation

To answer the last question, we have investigated how tools are evaluated. Most tools provide at least some form of evaluation, although for 27.7% of the tools we could only find anecdotal evidence, or none at all. The evaluation of a tool may not be directly related to the quality of the feedback, so the results only give a general idea of how much attention was spent on evaluation. The many different evaluation methods make it difficult to assess the effectiveness of feedback. Moreover, the quality (e.g. presence of control groups, pre- and post-tests, group size) of empirical assessment varies greatly. Finally, the description of the method and results often lacks clarity and detail.

Gross and Powers [96] provide a rubric for evaluating the quality of empirical tool assessments, and have applied this rubric to the evaluation of a small set of tools. Collecting data for this rubric would provide us with more information, but the effort is beyond the scope of this review. Just as Gross and Powers conclude, the lack of information on assessment greatly complicates this task. Pettit and Prather also endorse the need for developers of AA systems to work together instead of creating tools in isolation, and to pay more attention to evaluating their effectiveness [211].

In the future, it would be interesting to compare tools that give different types of feedback, to assess the effectiveness of different (combinations of) feedback types. Furthermore, extensive technical analyses are needed to verify to what extent a tool can correctly recognise (in)correct solutions and generate appropriate hints, and for which subset of exercises.

2.10.5 Classifying feedback

In this section we compare our classification of feedback to another classification from a recent paper. This paper by Le provides ‘a classification of adaptive feedback in educational systems for programming’ [159]. The author describes ‘adaptive feedback’ as feedback specific for the actions of an individual student, possibly combined with information from a student model. The classification consists of five main feedback types: ‘yes-no’, ‘syntax’, ‘semantic’, ‘layout’ and ‘quality’. Although the author states that these types are based on generic classifications of feedback types (including Narciss), we cannot clearly derive from the paper how the author’s classification relates to these general classifications. If we compare these types to ours, yes-no feedback relates to Narciss’ ‘knowledge of result/response’, which we do not include in our review. Syntax feedback relates to our ‘compiler errors’ type. Le distinguishes two levels in the semantic feedback category: intention-based (feedback on the solution strategy for solving a particular task) and code-based (feedback on coding errors with respect to a particular task), which are related to ‘solution errors’, both subtypes of ‘knowledge about how to proceed’, and ‘hints on task-processing rules’. Layout feedback corresponds to ‘style issues’ and quality feedback to ‘performance issues’. Although we do see similarities between both categorisations, ours differs in three ways: it resembles general feedback types at the top level, it includes a broader range of types, and it is slightly more fine-grained.

2.10.6 Threats to validity

There are a number of factors that could influence the validity of our review during the search for publications and the coding of the tools. We could have missed some papers because their authors use terms different from our database search string. We take this into account by inspecting references, but papers with few references might not have been found this way. For this reason, we could have missed some recent papers that typically do not have many references yet.

The substantial amount of work involved in this review, and therefore the time it took to execute the coding, may have had an effect on the interpretation of labels, in particular the borderline cases. In some cases we could not find clear answers to our research questions and had to speculate on what the correct label was. However, in these cases a second author was often consulted.

For answering RQ4, we might not have found some evaluations of recent tools that were conducted at a later date, because we have only included papers up to 2015. Validation of a tool is usually done by its authors, but in some cases may be performed by other authors. We did not do a search of these 'external' evaluations.

2.11 Conclusion

We have analysed and categorised the feedback generation in 101 tools for learning programming, selected from 17 earlier reviews and 2 databases. We have reported findings on the relation of feedback content, technique and adaptability. We have found that, in general, the feedback that tools generate is not very diverse, and mainly focused on identifying mistakes. In tools that support class 3 exercises, test-based feedback is most common and very few of these tools give feedback with 'knowledge on how to proceed'. Class 2 tools give more feedback on fixing mistakes and taking a next step, but at the cost of not recognising alternative strategies. Offering multiple feedback types in one tool and considering the level of detail would contribute to giving more effective feedback, similar to the feedback of a human tutor. We already see more diversity of feedback types in tools developed in the 21st century.

We have found many different techniques used to generate feedback, both general and specific to the programming domain, often strongly related to a specific type of feedback. The upcoming trend of data-driven tutors shows promising results, but also poses questions for which we need to find answers regarding the effectiveness of the hints they produce. Requiring large data sets also complicates the authoring of new exercises. We observe that teachers cannot easily adapt tools to their own needs, except for providing test data as input for a tool. Finally, the quality of the evaluation of tools varies greatly and is still an area for improvement. Better technical analyses should make it clearer what the features and limitations of a tool are, and better experiments that measure learning should give more insight into the effectiveness of a tool.

Automated feedback for programming exercises has been an active field of research for many decades. Generating effective feedback messages and making it easy for teachers to adapt tools to their own needs will remain a challenge, although many useful techniques have already been employed in practice. New technologies combined with techniques from this review that have shown to be effective will undoubtedly help new students wanting to learn how to program even better in the future.

Chapter 3

Code Quality Issues in Student Programs

This chapter is a published paper [141].

Abstract Because low quality code can cause serious problems in software systems, students learning to program should pay attention to code quality early. Although many studies have investigated *mistakes* that students make during programming, we do not know much about the *quality* of their code. This study examines the presence of quality issues related to program flow, choice of programming constructs and functions, clarity of expressions, decomposition and modularization in a large set of student Java programs. We investigated which issues occur most frequently, if students are able to solve these issues over time and if the use of code analysis tools has an effect on issue occurrence. We found that students hardly fix issues, in particular issues related to modularization, and that the use of tooling does not have much effect on the occurrence of issues.

3.1 Introduction

Students who are learning to program often write programs that can be improved. They are usually satisfied once their program produces the right output, and do not consider the quality of the code itself. In fact, they might not even be aware of it. Code quality can be related to documentation, presentation, algorithms and structure [249]. Fowler [84] uses the term 'code smells' to describe issues related to algorithms and structure that jeopardise code quality. A typical example is duplicated code, which could have been put in a separate method. Another example is putting the same code in both the true-part and the false-part of an if-statement, even though that code could have been moved outside the if-statement. Low quality code can cause serious problems in the long term, which affect software quality attributes such as maintainability, performance and security of software systems. It is therefore imperative to make students and lecturers aware of its importance.

For a long time, researchers have been interested in how students solve programming problems and the mistakes that they make. Recently, large-scale data mining has made it possible to perform automated analysis of large numbers of student programs, leading to several interesting observations. For example, Altadmri and Brown [12] investigated over 37 million code snapshots and found that students seem to find it harder to fix semantic and type errors than syntax errors.

Although many studies have investigated the errors that students make, little attention has been paid to code quality issues in student programs. While Pettit et al. [210] looked at code quality aspects and found that several metrics related to code complexity increased with each submission, their study does not elaborate on the causes of these high metric scores. Aivaloglou and Hermans [5] analysed a large database of Scratch projects (a block-based, visual programming language) by measuring complexity and detecting different code smells. Although the complexity of most Scratch projects was not high, the researchers found many instances of these code smells.

In this study we analyse a wider range of code quality issues, and observe their appearance over time. Our data set, taken from the Blackbox database [41], contains over two million Java programs of novice programmers recorded in four weeks of one academic year. First, we investigate the type and frequency of code quality issues that occur in student programs. Next, we track the changes that students make to their programs to see if they are able to solve these issues. Finally, we check if students are better at solving code quality issues when they have code analysis tools installed.

The contributions of this paper are: (1) a selection of relevant code quality issues for novice programmers, (2) an analysis of the occurrence and fixing of these issues, and (3) insight into the influence of code analysis tools on issue occurrence.

The remainder of this paper is organised as follows: Section 3.2 elaborates on related studies on student programming behaviour. Section 3.3 describes the research questions, the data set we used, the code quality issues we have selected to investigate, and the automatic analysis. Section 3.4 shows the results for each research question, which are discussed in Section 3.5. Section 3.6 concludes and describes future work.

3.2 Related work

This section discusses previous research into student programming habits related to code quality. We also consider studies that have analysed student programming behaviour on a large scale.

Pettit et al. [210] have analysed over 45,000 student submissions to programming exercises. The authors monitored the progress that students made over the course of a session, in which students submit their solutions to an automated assessment tool that provides feedback based on test results. For each submission they computed several metrics: lines of code, cyclomatic complexity, state space (number of unique variables) and the six Halstead complexity measures (calculations based on the number of operators and operands of a program). The authors also distinguish between sessions in which the number of attempts within a specific time frame is restricted. The main conclusion from the study is that although the metric scores increase with each submission attempt, restricting the number of attempts has a positive influence on the code quality of students. Second, the authors argue that instructors should also consider coding style and quality, because focusing solely on testing may result in inefficient programs. The study does not elaborate on the particular problems that cause high complexity scores.

Aivaloglou and Hermans [5] analysed a database of over 230,000 Scratch projects. Scratch is a block-based programming language that is often used to teach children how to program. Besides investigating general characteristics of Scratch programs, the authors also looked at code smells, such as cyclomatic complexity, duplicate code, dead code, large scripts and large sprites (image objects that can be controlled by scripts). Translating to the object-oriented domain, a large script is comparable to a large class and a large sprite to a large method. In 78% of over 4 million scripts the cyclomatic complexity is one. Only 4% of the scripts has a complexity over four. In 26% of the projects the researchers identified code clones (12% for exact clones), consisting of at least five blocks. It should be noted that Scratch only supports procedure calls within sprites, leaving copy-pasting code as the only option. Dead code occurs in 28% of the projects. Large scripts (with at least 18 blocks) are present in 30% of the projects and large sprites (with at least 59 blocks) in 14% of the projects.

Breuker et al. [39] investigated the differences in code quality between first- and second-year students in approximately 8,400 Java programs in 207 projects, using a set of 22 code quality properties. They found that for half of the properties there were no major differences. For the remaining properties, some differences could be attributed to increased project size and complexity for second-year students. Finally, second-year students performed better because their code had smaller methods, fewer short identifiers, fewer static methods and fewer assignments in while and if-statements.

Much more research into code smells exists for professional code. For example, Tufano et al. [262] investigated the repositories of 200 software projects, answering the question when and why smells are introduced. They calculated five metrics related to the size and complexity of classes and methods, and proper use of object-orientation. They found that most smells first occur when a file is created and that, surprisingly, refactorings may introduce smells.

Altadmri and Brown [12] used data from one academic year of the Blackbox database to investigate what common student mistakes are, how long it takes to fix them, and how these findings change during an academic year. Although there are various other studies that look at these aspects, it had not been done on such a large scale before. Individual source files were tracked over time by checking them for 18 mistakes, and calculating how much time had passed before the mistake disappeared from the source file. One important observation from the study is that students seem to find it harder to fix semantic and type errors than syntax errors.

3.3 Method

This study addresses the following research questions:

RQ1 Which code quality issues occur in student code?

RQ2 How often do students fix code quality issues?

RQ3 What are the differences in the occurrence of code quality issues between students who use code analysis extensions compared to students who do not?

3.3.1 Blackbox database

Our data set is extracted from the Blackbox database [41], which collects data from students working in the widely used BlueJ IDE[1] for novice Java programmers. BlueJ, used mostly in first year programming courses, has a simplified user interface and offers several educational features, such as interacting with objects while running a program.

The Blackbox database stores information about events in BlueJ triggered by students, such as compiling, testing and creating objects. Blackbox stores data on sessions, users, projects, code files and tests, which are linked to these events. A *source file* is a file of which there may be multiple versions called *snapshots*, which are unique instances of the source file at a certain *event*.

The database has been receiving data constantly since June 2013, and contains millions of student programs to date. BlueJ users have to give prior consent (opt-in) to data collection, and all collected data is anonymous. Permission is required to access the database. In this study we have investigated programs submitted in four weeks of the academic year 2014–2015 (the second week of September, December, March and June). From the Blackbox database we extracted data on source files, snapshots, compile events, extensions and startup events, which we stored in a local database. We only extracted data on programs that are compilable.

3.3.2 Data analysis

We performed an automatic analysis of all programs in our data set that compiled successfully. To enable replication and checks, we have published the

[1]`www.bluej.org`

code online.[2] We counted the source lines of code (SLOC) for each file using the cloc tool.[3] Although this metric is sensitive to style and formatting and therefore not very accurate, it provided us with an indication of the size of a program.

Issues (RQ1)

Stegeman et al. [249] have developed a rubric for assessing code quality, based on their research into professional code quality standards from the software engineering literature and interviews with instructors. The rubric is based on a model with ten categories for code quality. We omit the categories that deal with documentation (the names, headers and comments categories) and presentation (the layout and formatting categories). Our study focuses on the remaining five categories that deal with algorithms and structure, because they are the most challenging for students:

- **Flow.** Problems with nesting and paths, code duplication and unreachable code.
- **Idiom.** Unsuitable choice of control structures and no reuse of library functions.
- **Expressions.** Expressions that are too complex and use of unsuitable data types.
- **Decomposition.** Methods that are too long and excessive sharing of variables.
- **Modularization.** Classes with an unclear purpose (low cohesion) and too many methods and attributes, and tight coupling between classes.

For each category, we selected a number of issues to investigate by applying the PMD tool to a limited set of student programs to identify the issues that occur most frequently. PMD[4] is a well-known static analysis tool that is able to detect a large set of bad coding practices in Java programs. We also used the Copy/Paste Detector tool (CPD)[5] included with PMD for duplicate detection.

[2] `github.com/hiekekeuning/student-code-quality`
[3] `github.com/AlDanial/cloc`
[4] `pmd.sourceforge.io/pmd-5.5.2`
[5] `pmd.sourceforge.io/pmd-5.4.1/usage/cpd-usage.html`

In PMD a *rule* defines a bad coding practice, and running PMD results into a report of rule violations. In this paper we use the term *issue* to refer to a rule in PMD. The PMD version we used offers 26 sets consisting of issues that all deal with a particular aspect.

We discarded sets of issues using the following criteria:

- An issue is too specific for Java, such as issues that apply to Android, JUnit and Java library classes.
- An issue is too advanced, strict or specific for novice programmers, such as exceptions, threads, intermediate-level OO concepts (abstract classes, interfaces) and very specific language constructs (e.g. the final keyword).
- An issue falls under the documentation or presentation categories.
- An issue points at an actual error.

Our first selection consisted of 170 issues in 12 sets. We used the default value for issues with a minimal reporting threshold, such as the value 3 for reporting an if-statement that is nested too deeply. Additionally, we added 'code duplication' as three issues that fire for duplicates of 50, 75 and 100 tokens. Our initial analysis was applied to a smaller set of programs from four different days throughout the academic year 2014–2015. For each unique source file we recorded for each issue if it occurred in some snapshot of that file.

For a more detailed analysis we made a selection of the 170 issues. For each issue we decided whether it should be included or not, based on the criteria mentioned above. We also discarded all issues in the 'controversial' set, 'import statements' set and the 'unused code' set, and issues that occurred in fewer than 1% of the unique files. Table 3.1 shows our final set of issues, now grouped according to the categories of Stegeman et al.

We ran PMD for these 24 issues on all compilable programs in the final data set of four weeks and stored the results in our local database. We cleaned the database by removing all data of the files that could not be processed and files with 0 LOC. For each of these 24 issues, we counted in how many unique source files it occurred at least once, and how often. We also calculated the differences in occurrence over time.

TABLE 3.1: Selected issues (report level) per category. Detailed explanations can be found at `pmd.sourceforge.io/pmd-5.5.2/pmd-java/rules`.

Flow	
CyclomaticComplexity (10)	Strict version that counts boolean operators as decision points.
ModifiedCyclomaticComplexity (10)	Counts switch statements as a single decision point.
NPathComplexity (200)	
EmptyIfStmt	
PrematureDeclaration	
Idiom	
SwitchStmtsShouldHaveDefault	
MissingBreakInSwitch	
AvoidInstantiatingObjectsInLoops	
Expressions	
AvoidReassigningParameters	
ConfusingTernary	
CollapsibleIfStmts	
PositionLiteralsFirstInComparisons	
SimplifyBooleanExpressions	
SimplifyBooleanReturns	
Decomposition	
NCSSMethodCount (50)	Counts Non-Commenting Source Statements, report level in statements.
NCSSMethodCount (100)	
SingularField	The scope of a field is limited to one method.
CodeDuplication (50)	Only identified in single files, not over projects.
CodeDuplication (100)	
Modularization	
TooManyMethods (10)	Excludes getters and setters.
TooManyFields (15)	
GodClass	
LawOfDemeter	Call methods from another class directly.
LooseCoupling	Use interfaces instead of implementation types.

Fixing (RQ2)

For RQ2 we examine the changes in a source file over time. For each issue we calculated the number of *fixes* and the number of *appearances*. As an example, let us assume that source file X has 6 snapshots in which the occurrences of issue Y are `2 1 3 0 4 2`. The number of fixes is 6: the total number of issues that were solved in a subsequent snapshot (1 + 0 + 3 + 0 + 2). The number of appearances is 8: the total number of issues that were introduced in a subsequent snapshot (2 + 0 + 2 + 0 + 4 + 0). These metrics are simplified measures to investigate fixing: we cannot be sure the student really fixed the problem, or simply removed the problematic code.

Extensions (RQ3)

BlueJ users may install various extensions to support their programming, such as UML tools, submission tools and style checkers. We generated a list of all extensions used in the selected four weeks of the year 2014–2015. We selected extensions related to code quality from the 29 that were active in at least 0.05% of all BlueJ-startups in those weeks:

- Checkstyle[6] (9,626 start-ups), a static analysis tool for checking code conventions.
- PMD (3,751 start-ups), the tool used for our analysis.
- PatternCoder[7] (507 start-ups), which helps students to implement design patterns.

Findbugs[8] translates Java code into bytecode, and then performs static analysis to identify potential bugs. It is a relevant tool, but with 242 start-ups not used often enough. We also excluded a small number of extensions that we could not find information about.

For RQ3, we calculated the occurrence of issues for each of the extensions, and for source files for which no extensions were used.

[6] `checkstyle.sourceforge.io`
[7] `www.patterncoder.org`
[8] `findbugs.sourceforge.net`

TABLE 3.2: Data set summary.

Initial data set (4 days)	Unique source files	90,066
	Snapshots	439,066
Final data set (4 weeks)	Unique source files	453,526
	Snapshots	2,661,528
	Avg events per source file	5.87
	Median events per source file	2
	Max events per source file	700
	Average LOC per source file	52.75
	Median LOC per source file	27

3.4 Results

Table 3.2 shows some general information on the data sets taken from the academic year 2014–2015.

3.4.1 All issues (RQ1)

Table 3.3 shows the summary of checking the initial data set of four days for 170 issues. For each unique source file we recorded for each issue if it occurred in some snapshot of that file. In total we found 574,694 occurrences of 162 different issues (8 issues did not occur in any file). The top 10 issues is shown in Table 3.4.

In the controversial set, seven issues were found in at least 5% of the unique source files. DataFlowAnomalyAnalysis is at the top of the list with 38.6%. This issue deals with redefining variables, undefinitions (variables leaving scope) and references to undefined variables, which may not always be a serious problem. AvoidLiteralsInIfCondition is second with 14.0%. For other issues such as AtLeastOneConstructor and OnlyOneReturn it is also questionable whether they are problematic in novice programmer code, therefore we decided to further omit all issues in this set.

The top 10 also includes issues that we omit in the remainder of this study. The two issues that occur in the most files, 84.2% for MethodArgumentCouldBeFinal and 61.3% for LocalVariableCouldBeFinal, are both in the optimization set and point at the possibility to use the final keyword to indicate that a variable will not be reassigned. A reason for these high percentages may be that this language construct is not being taught to novice programmers.

Table 3.3: Summary of running PMD on the initial data set, showing per PMD set the number of issues that were seen, the percentage of unique files in which at least one issue from that set occurred, the median of the occurrences in % and the maximum.

Set	Issues seen	% of files with issues from set	Median %	Max %
Type resolution	4/4	26.04	3.96	20.1
Optimization	12/12	91.75	2.71	84.2
Unused code	5/5	26.86	2.50	16.2
Code duplication	3/3	4.99	2.28	5.0
Code size	13/13	13.69	1.40	8.2
Controversial	21/22	65.10	1.37	38.6
Import statements	6/6	10.61	1.02	8.5
Design	54/57	81.73	0.32	38.0
Unnecessary	8/8	10.25	0.11	9.6
Empty code	10/11	5.18	0.08	2.2
Coupling	3/5	41.98	0.04	39.7
Basic	23/24	2.52	0.02	1.3

UseVarargs deals with the ‘varargs’ option introduced in Java 5, allowing parameters to be passed as an array or as a list of arguments. UseUtilityClass points at the option to make a class with only static methods a utility class with a private constructor. ImmutableField detects private fields that could be made final.

3.4.2 Selected issues (RQ1)

We now focus on the selection of 24 issues in five categories (Flow, Idiom, Expressions, Decomposition, Modularization), which we applied to our final data set of four weeks. In total we found over 24 million instances of these issues. Table 3.5 shows in how many unique source files an issue occurs at least once, and the average number of occurrences per KLOC. To calculate this last value, we first calculated the average for each source file, and then the overall average, so the number of snapshots of a source file does not influence the total.

LawOfDemeter stands out as an issue with a very high number of occurrences. Upon closer inspection, it was not always clear why this issue was

TABLE 3.4: Top 10 issues.

Set	**Issue**	**In % of files**
Optimization	MethodArgumentCouldBeFinal	84.2
Optimization	LocalVariableCouldBeFinal	61.3
Coupling	LawOfDemeter	39.7
Controversial	DataflowAnomalyAnalysis	38.6
Design	UseVarargs	38.0
Design	UseUtilityClass	36.2
Design	ImmutableField	27.8
Type Res.	UnusedImports	20.1
Unused Code	UnusedLocalVariable	16.2
Controversial	AvoidLiteralsInIfCondition	14.0

reported, and it has been suggested online that there might be false positives. We therefore decided to omit this issue in the remainder of this study.

It is expected that SingularField occurs quite often with 8.2%, because most of the snapshots in our data set are unfinished programs. CyclomaticComplexity and the more lenient ModifiedCyclomaticComplexity version occur quite often with 7.7% and 5.2% respectively, which could point to serious problems, but that depends on the type of code. LooseCoupling occurs in 6.7% of the files implying that students do not always have knowledge of the use of interfaces. Duplicate50 occurs much more often than Duplicate100 with 4.7% against 1.3%. We argue that the lower threshold of 50 tokens is more suitable for novice programmers, whose programs are relatively short, so duplicates can be spotted more easily.

Figure 3.1 shows the occurrence of issues by the month in which they appeared, grouped by category. In the week of September the number of issues is quite low, probably because most courses had just started and only a limited set of topics would have been introduced. For the other three months we cannot see major differences, other than an increase in decomposition issues. In March we see a slight decrease in issues mainly in the flow and expressions category, but towards the end of the academic year the values slightly increase.

TABLE 3.5: Per issue, column I shows the percentage (%) of unique files in which the issue occurs, column II shows the average number of occurrences per KLOC.

Cat	**Issue**	**I**	**II**
M	LawOfDemeter	38.7	42.6
D	SingularField	8.2	3.8
F	CyclomaticComplexity	7.7	1.5
M	LooseCoupling	6.7	2.1
I	AvoidInstantiatingObjectsInLoops	6.3	1.6
E	AvoidReassigningParameters	5.7	1.7
F	ModifiedCyclomaticComplexity	5.2	0.8
M	TooManyMethods	5.0	0.3
D	Duplicate50	4.7	0.7
E	ConfusingTernary	4.4	0.7
D	NcssMethodCount50	3.9	0.3
E	PositionLiteralsFirstInComparisons	3.5	1.6
F	NPathComplexity	3.3	0.3
E	SimplifyBooleanExpressions	3.1	0.8
F	PrematureDeclaration	2.6	0.4
M	GodClass	2.1	0.1
F	EmptyIfStmt	2.0	0.3
E	SimplifyBooleanReturns	1.9	0.4
I	SwitchStmtsShouldHaveDefault	1.7	0.3
I	MissingBreakInSwitch	1.4	0.2
D	Duplicate100	1.3	0.1
E	CollapsibleIfStatements	1.3	0.2
M	TooManyFields	1.2	0.1
D	NcssMethodCount100	1.0	0.0

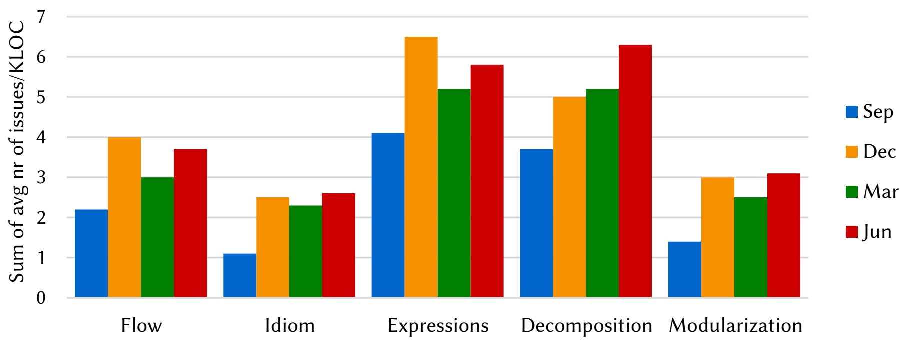


FIGURE 3.1: Issues over time.

3.4.3 Fixing (RQ2)

Table 3.6 shows our fix metrics for each issue. EmptyIfStmt is solved in almost half of the cases, which can be expected because an if-statement with no code in it is probably not finished. The same can be said for SingularField: a student might start with defining the field of a class that is needed for methods that will be added later. On the bottom of the list we find four issues from the modularization category (GodClass, LooseCoupling, TooManyFields, TooManyMethods) that are fixed in fewer than 5% of the appearances.

Overall the rate of fixing issues is low. Either students do not recognise these issues in their code, or do not care to fix them. It should be noted that our data set was not cleaned of source files that continued to be fixed beyond the weeks (Monday to Sunday) we investigated, missing some possible fixes.

3.4.4 Extensions (RQ3)

Table 3.7 shows general information on the use of extensions. Figure 3.2 shows the differences in occurrence of issues between source files for which extensions were and were not active. The figure shows that there is only a small difference between the use of a tool compared to using no tool. Students using no tool even have a slightly smaller number of issues with 18.2 issues on average per KLOC versus 19.7 for students that use some tool.

Table 3.6: Issue fixes.

Cat	Issue	Appeared	Fixed	%
F	EmptyIfStmt	18,460	9,064	49.1
D	SingularField	103,004	30,152	29.3
F	PrematureDeclaration	21,008	5,891	28.0
D	Duplicate100	35,033	7,388	21.1
E	CollapsibleIfStatements	15,087	2,579	17.1
D	Duplicate50	91,951	15,520	16.9
E	AvoidReassigningParameters	76,359	10,023	13.1
I	MissingBreakInSwitch	9,594	1,033	10.8
F	NPathComplexity	20,549	2,129	10.4
E	ConfusingTernary	36,391	3,558	9.8
E	SimplifyBooleanReturns	12,612	1,162	9.2
E	SimplifyBooleanExpressions	48,965	4,347	8.9
F	ModifiedCyclomaticComplexity	56,822	4,475	7.9
I	AvoidInstantiatingObjectsInLoops	78,588	6,167	7.8
I	SwitchStmtsShouldHaveDefault	12,507	961	7.7
D	NcssMethodCount50	23,569	1,790	7.6
F	CyclomaticComplexity	85,426	6,240	7.3
D	NcssMethodCount100	6,178	410	6.6
E	PositionLiteralsFirstInComparisons	86,536	4,833	5.6
M	GodClass	9,575	437	4.6
M	LooseCoupling	57,039	2,056	3.6
M	TooManyFields	5,539	175	3.2
M	TooManyMethods	23,003	515	2.2

TABLE 3.7: Extension use.

Name	**Snapshots**	**KLOCs**	**Unique source files**
Checkstyle	73,553	7,756	10,833
PMD	26,126	1,840	4,299
PatternCoder	2,433	113	609

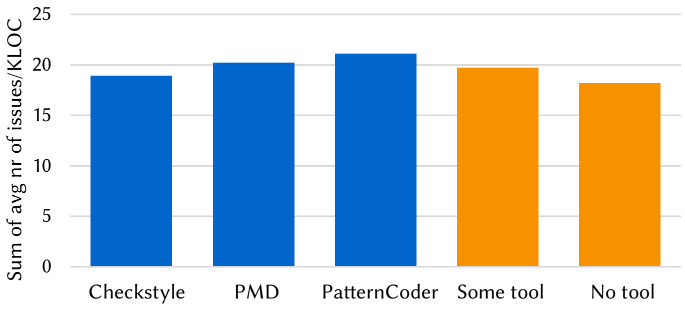


FIGURE 3.2: Issues and extension use.

3.5 Discussion

One of our main findings is that most issues are rarely fixed, especially when they are related to modularization. Another finding is that the use of tools has little effect on issue occurrence. Compared to the study of Scratch projects by Aivaloglou and Hermans [5], we found lower percentages of files that contain duplicates, large classes and large methods. Some reasons might be that block-based code cannot be directly compared to statement-based code and that block-based programming is targeted at a younger audience. Another reason is that we investigated single source files instead of projects. Our study supports the work of Pettit et al. [210] by observing that quality issues are not often solved, although we cannot confirm the positive effect of restricting submission attempts, because our data set does not contain information on submissions.

From working with PMD as a source code analyser we have noticed some problems with regard to suitability for students. PMD integrates with many IDEs and also provides an extension for users of BlueJ. We found that many of the checks PMD can perform are not suitable for novice programmers, and may cause confusion with students that might result in neglecting the tool.

We advise educators to customize the tool by selecting a small set of relevant checks and adjusting threshold values. Other recommendations for using PMD for assessing programming exercises have been proposed by Nutbrown and Higgins [201].

The main focus of the field of automated feedback and assessment of programming exercises has been on functional correctness of programs, although some tools incorporate feedback on quality aspects as well [145]. This is often done by integrating a lint-like tool or calculating metrics such as cyclomatic complexity and LOC (e.g. [8], [17]). Many professional IDEs detect code quality issues and offer refactorings, but these are often too advanced for novices and not intended to support learning. We argue that these tools need to be better suited to novices, and should be used at various moments during learning and not only for assessment.

3.5.1 Threats to validity

The designers of the Blackbox project mention some restrictions of their data set that also affect this study [41]. First, BlueJ is often used in courses that use an 'objects-first' approach. Second, it is unknown on what task the student is working, and what the requirements of this task are, such as using a particular language construct. Third, we know nothing about the users of BlueJ. We expect them to be novices, but some programs have probably been written by instructors or more experienced programmers.

We have a limited data set of four weeks in one year. We also cannot be sure that we have all snapshots, events might be missed because something went wrong (e.g. no internet connection) or a user continued to edit the code in another program. Because we store weeks, we miss some snapshots that were compiled just before or after the week. However, because of its size we believe our data set has enough information to answer our research questions. Only tracking single files and not complete BlueJ projects gives an incomplete view of the presence of duplicates.

Vihavainen et al. [273] have investigated the effect of storing student data of different granularity: submission-level, snapshot-level (e.g. compiling, saving), and keystroke-level (e.g. editing text), and found that data might be lost if only snapshot events are studied. Although the Blackbox data set also stores keystroke events, we believe that researching compile events provides us with sufficient information. For a more detailed analysis, investigating keystrokes could provide more insight into how students fix quality issues.

Although this study focuses on Java programs, we believe that the findings may apply to other languages too. The issues we investigated are not Java specific and can also be seen in other modern object-oriented languages. For functional and logic languages some issues are not applicable or should be adjusted for the paradigm.

3.6 Conclusion and future work

In this study we have explored quality issues in 2.6 million code snapshots written by novice programmers using the BlueJ IDE. We have composed a list of issues that are relevant for novices. We found that novice programmers develop programs with a substantial amount of code quality issues, and they do not seem to fix them, especially when they are related to modularization. The use of tools has little effect on the occurrence of issues. Educators should pay attention to code quality in their courses, and automated tools should be improved to better support students in understanding and solving code quality issues. Further research is required to better understand how students deal with quality issues, for example by investigating the changes made in snapshots. Also, it is of importance to examine the reasons why students produce low-quality code: they may be unaware of it, or they simply do not know how to fix their code. Paying attention to code quality in education is vital if we want to keep improving our software systems.

Chapter 4

How Teachers Would Help Students to Improve Their Code

This chapter is a published paper [142].

Abstract Code quality has been receiving less attention than program correctness in both the practice of and research into programming education. Writing poor quality code might be a sign of carelessness, or not fully understanding programming concepts and language constructs. Teachers play an important role in addressing quality issues, and encouraging students to write better code as early as possible.

In this paper we explore to what extent teachers address code quality in their teaching, which code quality issues they observe and how they would help novices to improve their code. We presented student code of low quality to 30 experienced teachers and asked them which hints they would give and how the student should improve the code step by step. We compare these hints to the output of professional code quality tools.

Although most teachers gave similar hints on reducing the algorithmic complexity and removing clutter, they gave varying subsets of hints on other topics. We found a large variety in how they would solve issues in code. We noticed that professional code quality tools do not point out the algorithmic complexity topics that teachers mention. Finally, we give some general guidelines on how to approach code improvement.

4.1 Introduction

An increasing number of studies have focused on the *quality* of programs written by novices, as opposed to the *correctness* of student programs, which has had quite some attention in research the past decades [42], [219]. These studies show that student programs contain a substantial amount of various quality issues, which often remain unsolved [62], [141], [210]. While there has not been much research into the reasons why quality issues remain unsolved, one can imagine that students are satisfied once their solutions pass all tests. They might not even be aware of quality aspects such as maintainability, performance and testability, or simply do not know how to satisfy them. Although a wealth of tools exists to analyse and refactor code, they are often not targeted at novices. Therefore, teachers play an important role in raising awareness of quality issues and encouraging students to improve functionally correct code.

There is little information on how to support students with improving the quality of their code, and what teachers consider to be a high-quality program. We conducted a study in which we collected this knowledge from experts. We asked 30 experienced educators who teach programming how they perceive the role of code quality in their courses. We showed them a number of functionally correct programs that have several issues related to quality, and asked them which hints they would give to help improve the program. We also asked them to describe the steps they would want the student to take to refactor the program into an improved version.

This paper (1) gives insight into how teachers assess the quality of novice programs, (2) shows how their hints compare to feedback generated by tools, (3) analyses how teachers would rewrite poor student code, and (4) describes how they would approach this rewriting in a stepwise way. These insights can be used to improve the development of courses and tools.

Section 4.2 gives some background and discusses related work. Section 4.3 describes the research questions, and how we collected and analysed the data. Section 4.4 shows the results for each research question, which are discussed in Section 4.5. Section 4.6 concludes and describes future work.

4.2 Background and related work

This section establishes the meaning of central terms used in this paper and summarizes related work on code quality in education.

4.2.1 Code quality terms and definitions

Code quality deals with the directly observable properties of source code, such as algorithmic aspects (flow, expressions, language constructs) and structure (decomposition, modularization). Some examples of code quality issues are (1) duplicated code, (2) an expression that could be shortened, and (3) unnecessary conditional checks. Although layout and commenting are certainly indicators of code quality, these aspects are beyond the scope of our study.

Fowler [84] uses the term *code smells* to describe characteristics in code that might indicate that something is wrong with the design of functionally correct code, which can have an impact on its quality. In the long term, low quality code may affect software quality attributes such as maintainability, performance and security. There are many tools available (e.g. PMD, SonarQube, Resharper, linters) to automatically detect quality issues and code smells in a program.

Code refactoring is improving code step by step while preserving its functionality. The well-known work by Fowler [84] describes a collection of refactorings, mainly focused on the structure of the code. Code Complete [188], a well-known handbook for software construction, describes refactorings on multiple levels: data-level (e.g. inline an expression), statement-level (e.g. use return instead of a loop control variable), routine-level (e.g. extract method), class implementation, class interface and system-level. Some IDEs offer support for refactorings, such as renaming variables and extracting methods. These IDEs execute a refactoring in a single step, which would not give novices much insight into how refactoring works.

An ITiCSE working group [37] investigated which quality aspects are considered important by teachers, students and developers. In our study we zoom in on how teachers assess the quality of student code, focussing on data-, statement- and routine-level refactorings, which are most relevant for the programs that beginners write.

4.2.2 Code quality in education

Multiple studies have investigated the quality of student programs. Pettit et al. [210] analysed submissions to an automated assessment system and found that several complexity metrics increased with every submission. Keuning et al. [141] detected many quality issues in over 2 million student programs, which were hardly ever fixed. Whether the student used a quality tool or not did not decrease the amount of issues. Breuker et al. [39] found no clear quality improvement between the code of first- and second-year students. De Ruvo et al. [62] investigated a set of 19.000 code submissions on 16 semantic style indicators, which address small issues such as unnecessary return statements, and too complex if-statements. They found instances in both code of novices and more experienced students. Luxton-Reilly et al. [174] investigated differences between correct solutions to programming exercises, identifying variation in structure, syntax and presentation. The authors found that even for simple exercises there are numerous variations in structure, and in some instances the teacher's solution was not the most popular one among students.

Although professional code quality analysers and refactoring tools are being used in education (e.g. [201]), there are also some tools designed specifically for education that give feedback on code quality, such as Style++ [8], FrenchPress [32], and AutoStyle [288]. AutoStyle gives stepwise feedback on how to improve the style of correct programs, based on historical student data.

Educators have designed several projects that teach students about refactoring, usually for more advanced students [2], [68], [243], [250]. Experienced educators studied the quality of object-oriented examples in Java textbooks [36]. They found several issues, in particular related to object-oriented thinking.

4.3 Method

The research questions this study addresses are:

RQ1 To what extent do teachers address code quality in their programming courses?

RQ2 What kind of hints related to code quality do teachers give to students, and how do these hints compare to the output of code quality tools?

RQ3 Which (stepwise) approach do teachers suggest to help students improve their programs, and what does the final improved program look like?

4.3.1 Study design

We designed a questionnaire (see Appendix A) in which participants answer five questions about themselves and four short questions on the role of code quality. Next, we give our definition and scope of code quality and present three student programs of poor quality (see Section 4.3.1). We ask (1) how they would assess the program, (2) which hints they would give, and (3) how they would want the student to improve the program step by step, by typing the code after each step. We tested the questionnaire with a teacher who is not involved in this research, and adjusted the questionnaire according to his feedback.

We invited university teachers with at least two years of experience in teaching CS/programming-related courses to participate in our study. We did not ask professional programmers, because they do not necessarily have experience with teaching programming. We sent our invitation to 66 teachers from various institutes and countries, and asked them to forward the invitation to colleagues and other acquainted teachers.

Programs

The first exercise was taken from another study, including the most popular student solution (see Program 1) [174]. Its description is: 'Implement the sumValues method, which adds up all numbers from the array parameter, or only the positive numbers if the positivesOnly boolean parameter is set to true.'

We designed the second exercise ourselves: 'Write the code for the method unevenSum. This method should return the sum of the numbers at an uneven index in the array that is passed as a parameter, until the number -1 is seen at an uneven index.' We collected 78 solutions from an institution one of us works at. We composed the first solution (Program 2a) by mixing a number of actual student solutions. Program 2b is an actual correct student solution (with variable names translated into English).

We ran three well-known static analysis tools on the three programs: PMD with the full set of rules[1], Checkstyle[2], and SonarLint[3] with default checks and full checks. Because Checkstyle always reported a subset of the PMD and/or SonarLint messages (besides layout), we omit Checkstyle messages from this paper.

4.3.2 Data analysis

We analysed the answers to the open question on which hints a teacher would give both qualitatively and quantitatively. We labelled the hint topics using an

[1]`pmd.github.io/pmd-6.9.0`
[2]`checkstyle.sourceforge.io` (version 8.14)
[3]`www.sonarlint.org/eclipse` (version 4)

open coding method, and categorized them using the rubric by Stegeman et al. [249]. This rubric has been developed for assessing student code quality, based on a model with ten criteria. We assigned each topic to one of the four criteria that deal with algorithms and structure, which are *flow* (nesting, code duplication), *idiom* (choice of control structures, reusing library functions), *expressions* (complexity, suitability data types) and *decomposition*. The open coding was performed by one author. Another author checked the labelling of a randomly chosen 10% of the hints (with 89% agreement), and differences were discussed.

For the question on improving the program stepwise we performed a series of actions on the submitted programs. First, we removed steps in which the code was not changed (possibly a copy-paste issue). Next, because most participants probably did not use a compiler, we corrected syntax errors such as missing brackets and misspelled names, and converted to Java syntax (all given programs were written in Java, although we did not explicitly mention this). We also corrected some other small errors that were clearly unintentional. All programs were tested with a set of test cases. We assigned each program to a *cluster* based on the control flow of the program (loops, conditionals, branching and methods), because control flow shows the main structure and complexity of a method (the scope of the programs in our study). Clustering allows us to investigate similarities and differences without being distracted by details. We identified *transformations* as the add/edit/delete steps between two adjacent program *states* in a sequence.

4.4 Results

4.4.1 Background of teachers

In total, 30 participants took part in our study. All participants teach programming and other CS-related courses at university level in 3 different countries: The Netherlands (27), Sweden (3) and China (1). The 28 participants that reported their institute, teach at 15 different institutes: 10 in The Netherlands, 4 in Sweden and 1 in China, with between 1 and 5 teachers per institute. A few teach at more than one institute and country. The teachers have between 2 and 33 years of teaching experience, with an average of 11.4 years, and a median of 9 years. A total of 90% teach first year courses, 80% teach second year courses, and 70% teach courses for students in their third year or higher.

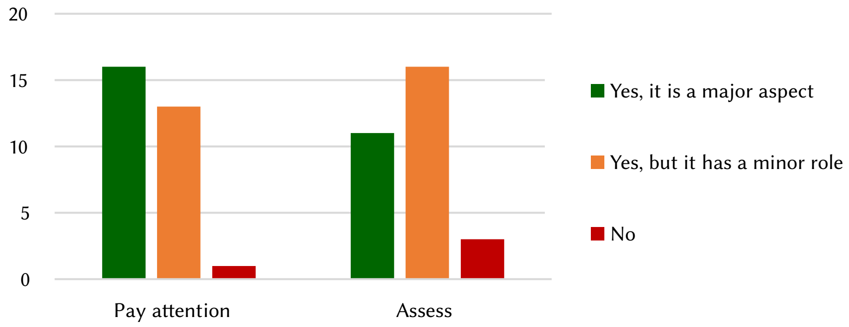


FIGURE 4.1: Responses to questions 'Do you pay attention to code quality while teaching programming to first- and second-year students?' and 'Do you explicitly assess/grade code quality aspects in programming assignments?'.

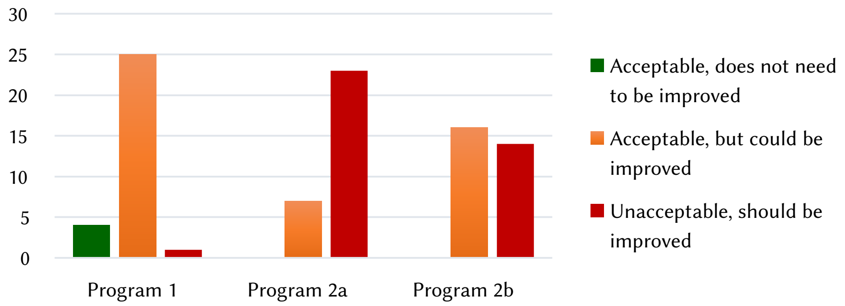


FIGURE 4.2: Responses to question 'How would you assess this solution in a formative situation (e.g. feedback during a lecture or lab)?' for all three programs.

4.4.2 Role of code quality (RQ1)

We asked teachers if code quality appears in the learning goals of their first- and second-year programming courses, to which 23 replied with 'yes' and 7 with 'no'. Figure 4.1 shows to what extent teachers address and assess code quality aspects. Code quality clearly has a smaller role in assessment than in teaching.

A total of 11 out of 30 teachers do not advise or prescribe tools that deal with code quality/refactoring to their students. The 19 that do advise or prescribe tools, mostly mention static analysis tools (e.g. SonarQube, Checkstyle, linters) and IDE functionality or their plugins (e.g. Resharper). To a minor

PROGRAM 1: Most popular solution to exercise 1: 'the sumValues method adds up all numbers from the array parameter, or only the positive numbers if the positivesOnly boolean parameter is set to true.'

```
int sumValues(int [] values,
              boolean positivesOnly) {
    int sum = 0;
    for (int i = 0;i < values.length;i++) {
        if (positivesOnly == true) {
            if (values[i] >= 0) {
                sum += values[i];
            }
        }
        else {
            sum += values[i];
        }
    }
    return sum;
}
```

extent testing and code reviewing is mentioned. All tools mentioned are professional tools, and not explicitly intended for education.

4.4.3 Program hints and steps (RQ2 and RQ3)

Program 1

Running PMD on Program 1 reports that the for could be replaced by a foreach, and that unnecessary comparisons in boolean expressions should be avoided. SonarLint only reports on the equals true. Figure 4.2 shows how the teachers would assess this solution in a formative situation. Most teachers (25) answered 'acceptable, but could be improved'.

We asked the teachers to describe all hints they would give to a student to improve this program. Table 4.1 shows all hint topics and the number of mentions. For the issue that was pointed out the most, the flow inside the loop, some participants focussed more on the duplication, and others more on the complex if-structure.

Next, we asked the teachers how they would want the student to edit (refactor) the program step by step. In total 3 teachers did not provide any steps: 1 found the method specification itself problematic, the others did not

Table 4.1: Hint topics for program 1, as reported by 30 teachers. Indented topics are more specific.

Category	**Description**	**Count**
Expressions	Remove equals true (line 5)	20
	Do not add 0 to sum (line 6)	2
Flow	Improve flow in loop	1
	Improve nested ifs (line 5–12)	17
	Remove duplicated `sum += ..` (line 7 and 11)	11
Idiom	Change type of loop (line 4)	11
	Use a higher-level function	3
Decomposition	Move common code to method	2
	`positivesOnly` check to method (line 5)	2
Other	General	2
	Misc. mentioned once (various categories)	5

give a reason. The remaining 27 teachers provided 2.8 steps on average, with a median of 3 (min 1, max 5) and a total of 76 program states, of which 11 were incorrect.

Regarding the type of loop, 10 teachers transformed the for into a foreach at various stages of the process, always as a single step in which nothing else was done. As a last step, 2 teachers replaced the loop by a functional style solution, calling a higher-level sum function on the array. Teachers mentioning this approach mostly said that they would only suggest it to more advanced students.

Regarding the flow in the loop, in 8 final programs the duplicated sum increment was still present. Removing the duplication by merging the ifs into a single statement, was often done as an early step, after removing the `==true`. In 2 cases a continue was used to skip a value that should not be added. However, we noticed merging the ifs was problematic: all 11 incorrect programs contained a merging mistake. Some of those mistakes were fixed in a next step.

We assigned each (intermediate) program to a cluster based on its control flow, identifying 13 clusters. The final programs were distributed over 10 clusters; excluding 4 incorrect final programs, we counted 9 clusters, as shown in Table 4.2.

TABLE 4.2: Correct end clusters for program 1.

Start:	12x:	3x:	2x:	6x: Others
`for` `if` `if` `else` `return`	`for/foreach` `if` `return`	`for/foreach` `if` `if` `else` `return`	`for/foreach` `if` `continue` `return`	

```
Start:     12x:            3x:             2x:
for        for/foreach     for/foreach     for/foreach
  if         if              if              if
    if     return              if              continue
  else                       else            return
return                     return
```

TABLE 4.3: Hint topics for program 2a.

Category	Description	Count
Expressions	Remove self assignment (line 14)	12
	Remove equals false (line 6)	7
	Use compound operator += (line 5)	2
Flow	Exit from loop when done	20
	Improve flow in loop	6
	Remove redundant if (line 9)	10
	Remove unnecessary else (line 13–15)	7
	Reverse if-else (line 6–15)	2
Idiom	Change type of loop (line 5)	8
Other	Fix functional error stop condition	13
	General hints	9
	Add tests	3
	Misc. mentioned once (various categories)	7

Program 2a

PMD reports three ‘dataflow anomalies’ for the `total` and `stop` variables in Program 2a. PMD considers this a low-priority issue that might not be problematic. PMD also points out equals false and the self-assignment. SonarLint with default settings also reports on equals false and even gives two messages on the self-assignment. With full checks, it mentions that the if-else if (lines 7–11) should end with an else, and a constant should be used for magic number 2.

The program contains a functional error regarding the stop condition. We instructed participants to ignore this error when answering the questions. However, the first 10 participants did not see this note and read that it was a correct solution. Figure 4.2 shows the response to the question on formatively

Program 2a: Solution to exercise 2: ‘the method unevenSum should return the sum of the numbers at an uneven index in the array that is passed as a parameter, until the number -1 is seen at an uneven index.’

```
int unevenSum(int [] array) {
    int total = 0;
    boolean stop = false;

    for (int i = 1;i < array.length;i = i + 2) {
        if (stop == false) {
            if (array[i] >= 0) {
                total += array[i];
            } else if (array[i] < 0) {
                stop = true;
            }
        }
        else {
            total = total;
        }
    }
    return total;
}
```

assessing the solution, to which most teachers (23) answered it was unacceptable. The fact that the solution was incorrect could have contributed to this score.

Table 4.3 shows the hint topics for this program. Teachers often mentioned improving the complex flow in the loop, exiting from the loop when the stop condition was met, changing the loop type and removing clutter in the expressions.

Looking at the edit steps, 28 teachers provided 3.0 steps on average (min 1, max 5), with a median of 3 steps and a total of 83 program states. Two teachers did not provide any steps (they did not know what to suggest or what a student would have to do). Of the 83 program states, 8 were not functionally correct according to either stop condition (-1 or negative). We excluded 1 of the 28 sequences from the analyses below because it had unclear steps.

Although we did not explicitly ask this, 15 teachers tried (sometimes unsuccessfully) to fix the functional error. Most of them (11) started fixing in the first step; 3 of the others first removed clutter. In general, almost all clutter (the else with self-assignment, the redundant if) was removed by everyone mostly in the first or second step. The `==false` was mostly removed as part of another step.

The loop type was changed into a while 8 times at various stages of the process, and 1 teacher replaced the loop by recursion. Almost all participants changed the program to exit from the loop when the stop condition was met: 5 used a break, 5 used a return, and a majority of 14 added a stop condition to the loop header. Exiting from the loop was also done at various stages. The `stop` variable was eliminated from 20 sequences, and 7 kept it.

In total 20 clusters were identified, and the final programs were in 12 clusters. Table 4.5 shows the 10 final clusters if we exclude the 5 incorrect final programs.

Program 2b

Running PMD on the body of Program 2b reports multiple dataflow anomalies, and reports that the variable `number` could be made ‘final’. SonarLint with full checks mentions magic number 2 on line 7. Figure 4.2 shows the formative assessment.

The hint topics for this program are shown in Table 4.4. The main topics were exiting from the loop when the stop condition is met, improving the

Program 2b: Actual student solution to exercise 2: 'the method unevenSum should return the sum of the numbers at an uneven index in the array that is passed as a parameter, until the number -1 is seen at an uneven index.'

```
 1  int unevenSum(int[] array) {
 2      int answer = 0;
 3      int index = 0;
 4      boolean value = true;
 5
 6      for(int number: array) {
 7          if(index % 2 == 0) {
 8              index++;
 9          } else {
10              if(number == -1) {
11                  value = false;
12              }
13              if(value) {
14                  answer = answer + number;
15              }
16              index++;
17          }
18      }
19      return answer;
20  }
```

Table 4.4: Hint topics for program 2b.

Category	Description	Count
Flow	Exit from loop when done	14
	Improve flow in loop (line 7–16)	5
	Remove `value` variable	3
	Reorder conditionals (line 7–16)	3
	Improve flow in else (line 10–16)	2
	Duplicated increment (line 8 and 16)	9
	Skip even indices	6
Idiom	Change type of loop (line 6)	14
Other	General hints	9
	Rename variable	7
	Misc. mentioned once (various categories)	9

complex flow in the loop, removing the duplicated increment, and replacing the foreach by another type of loop.

For the steps, 26 teachers provided 2.9 steps on average (min 1, max 5), with a median of 3 steps and a total of 76 program states. Of the 4 teachers that did not provide steps, 2 teachers advised that the student should start over instead of rewriting the program. The other 2 did not give a reason.

Teachers made multiple mistakes rewriting this program: of the 76 program states, 21 functionally incorrect programs were created by 10 teachers. The majority of the mistakes were related to incorrect indexing (not looping through only the odd indices). This mistake was often made in a step that also transformed the foreach into a for. We excluded 2 sequences from the analyses below because they had unclear steps, leaving 24 valid sequences.

Although in 2a the for-loop was mostly kept, followed by a few while-loops, in 2b we only counted 9 for-loops and 2 while-loops. The foreach was kept 13 times. Transforming the foreach into a for was mostly done in the first step, transforming to while was always done after a transformation into a for. Of the 11 that used a for or while, 6 skipped the even indices by incrementing the index by 2. For the remaining, 14 kept the modulo check, 2 introduced a boolean that switched between true and false, and 2 made a mistake.

Exiting from the loop was solved rather differently than in program 2a: more teachers used a break (10 vs. 5) and fewer teachers (6 vs. 14) added a stop condition to the loop header. This step was usually done somewhat later in the sequence.

All but 1 teacher removed the duplicated increment, mostly in an early step. Sometimes this was done in 2 steps: first moving the increment outside the if-else, followed by reversing the if-else and removing the then empty else. Transforming the foreach into a for also eliminates the duplication by moving the increment to the loop header.

Variables were renamed in a few cases: 4 renamed `answer` to `sum` or `total`, and `value` was occasionally renamed to `done` or `stop` if it was still used. Renaming was done at various stages.

In total we identified 31 clusters, and the final programs were in 14 clusters. Excluding 8 buggy final programs, we counted 11 clusters, as shown in Table 4.5. We would expect that the final programs after improving 2a match those of 2b (the error from 2a does not have to affect the cluster). However, only 5 participants ended in the same cluster for program 2a and 2b.

TABLE 4.5: Correct end clusters for program 2a and 2b.

Program 2a (n=22)

Start:
```
for
  if
    if
    else
      if
  else
return
```

6x:
```
for/while
return
```

4x:
```
for/while
  if
  else
return
```

2x:
```
for
  if
  else
    break
return
```

2x:
```
for
  if
    return
  else
return
```

2x:
```
for
  if
  else
    return
return
```

2x:
```
for
  if
    break
return
```

4x:
Others

Program 2b (n=16)

Start:
```
for
  if
  else
    if
    if
return
```

4x:
```
foreach/for
  if
    if
      break
return
```

2x:
```
foreach/for
  if
    if
    if
return
```

2x:
```
for/while
return
```

8x:
Others

4.5 Discussion

In this section we answer the research questions and discuss the results in more depth.

RQ1. The results of RQ1 show that while code quality is certainly an important topic for most teachers, its role is smaller in the summative assessment of student code.

RQ2. Based on teacher feedback on three low-quality implementations of simple methods, the hints that teachers would give deal with improving control flow, choosing representative names, using suitable language constructs, removing clutter, and optimising the algorithm. Although some hint topics are mentioned by a majority, other topics are only mentioned by much smaller subgroups, implying that teachers do not consider the same things important. This finding contradicts Nutbrown and Higgins' claim that their assessors were in agreement on assessment criteria [201].

Regarding the form of the feedback, we noticed several aspects. Hints were often formulated as a question, such as 'Is there any code duplication that you could remove?' The amount of detail considering *why* something should be improved varied: for transforming a for into a foreach, an example with more motivation is: 'If we are traversing all values in the array, couldn't we use another type of loop?' Other hints mention that the 'other type of loop' is a foreach, and some even provide the syntax of the foreach header.

When comparing teacher hints to what professional tools report, we see many differences. Tools do not give feedback with increasing detail as teachers would do. Issues related to control flow and algorithmic optimisations are not pointed out by tools. A main reason is that tools do not know what the code they analyse should do. Also, default tool settings usually have high thresholds, so minor issues such as small duplicated blocks are usually not reported. Our findings support and complement the finding of Nutbrown and Higgins that static analysis tools miss context-specific issues [201].

RQ3. While others have already shown the great diversity in student solutions (e.g. [174]), our study also shows this diversity in teacher solutions. Remarkable is the difference in final states for program 2a and 2b that solve the same problem. Program 2b was probably the most problematic, and this starting point could have influenced the final program. For example, about half of the teachers kept the foreach in 2b, but a foreach is never introduced in 2a. Perhaps there are simply multiple ways that are equally fine, however, not all hints seem to be addressed in the final programs for program 2b. Finally, some teachers could have lost their focus for the last program, which would also explain the number of mistakes.

The large variety in steps and final programs makes it difficult to give an approach on how to improve a problematic program. General guidelines we can extract from the data are: remove clutter first, fix errors early, keep testing along the way (even teachers make mistakes), rename to meaningful names, and do larger refactorings later one step at a time. Starting over could sometimes be advisable. However, learning *why* a program has flaws, and *how* to address those flaws step by step could be a valuable learning experience. We provide some examples of stepwise improvement sequences with hints for the programs discussed in Appendix B.

If educators want to assess code quality, there should be agreement on what a high-quality program is. Code quality should not be a teacher's personal preference. One could argue that quality is not that important for novices, however, we argue that several issues pointed out in the previous section are caused by misunderstanding certain language constructs. Improving a solution may be a valuable way of learning more about how a programming language works. Although the issues mentioned in this paper are of a low level, and the topic of 'refactoring' is mostly associated with restructuring complex object-oriented software, we advise to start refactoring early with the statement/method level, to the class level later in a study program.

4.5.1 Threats to validity

Because we only discussed three programs, we cannot generalise to all novice programs. However, the programs cover a broad set of constructs, and in this study we particularly aimed to make code quality, a potentially vague topic, more concrete by working with actual code. Conducting interviews could give us higher-quality data, but would not have given us insight into the diversity of the responses. It would be an interesting next step to discuss the various responses with the teachers to arrive at a more general view of which hints to give and which steps to take.

This study does not consider the responses of students to hints. A teacher would possibly adapt and give a more concrete hint when a student does not understand the initial hint. This was even mentioned by some participants in their responses. There might also be some differences between the type of hints for absolute beginners and second-year students.

4.6 Conclusion and future work

This paper describes a study in which we asked teachers for their opinion on the quality of student code and how they would help students to improve it. While teachers find the topic of code quality important, they have different views on how to improve code. Teachers mostly agree on issues related to reducing algorithmic complexity and removing clutter, but they give different subsets of hints. Professional code quality tools do not point out these algorithmic complexity topics that teachers mention. We also discussed the great diversity in the final programs, which is influenced by the initial state of the code, and derived some general guidelines in how to approach an improvement sequence.

In future work we intend to use our findings to build better tools that help students improve the quality of their code. This research also calls for more debate on what a high-quality solution would look like for a novice. Experiments in the classroom with students are required to further study how we should learn students to improve their code, to which this study contributes.

Chapter 5

A Tutoring System to Learn Code Refactoring

A modified version of this chapter has been submitted for publication [143].

Abstract In the last few decades, numerous tutoring systems and assessment tools have been developed to support students with learning programming, giving hints on correcting errors, showing which test cases do not succeed, and grading their overall solutions. The focus has been less on helping students write code with good style and quality. There are several professional tools that can help, but they are not targeted at novice programmers.

This paper describes a tutoring system that lets students practice with improving small programs that are already functionally correct. The system is based on rules that are extracted from input by teachers collected in a preliminary study, a subset of rules taken from professional tools, and other literature. Rules define how a code construct can be rewritten into a better variant, without changing its functionality. Rules can be combined to form rewrite strategies, similar to refactorings offered by most IDEs. The student can ask for hints and feedback at each step.

We evaluate the tutoring system by comparing it to existing tools, and demonstrating that the functionality of the system closely matches how teachers would help students with improving their code.

5.1 Introduction

Student misconceptions have always had much attention in studies on student programming [42], [219]. The focus has been mostly on programming mistakes resulting in functionally incorrect code. At the same time, there may be numerous functionally correct solutions to the same programming exercise [174], which are not always equally good. In 2017 an ITiCSE working group investigated how professionals, educators and students perceive code *quality*, finding a great diversity in its definition [37]. Their study also concluded that the topic of code quality is underrepresented in education. Recently, there has been increased attention to the style and quality of student solutions. Poor coding style and quality may lead to incomprehensible code that has low maintainability and testability, which is an issue even for professional software developers, not to mention novices. While one might argue that novice programmers should not be bothered too much with style and quality, these quality issues might point at underlying misconceptions. Also, *refactoring* code is an important skill that every programmer should possess, and novices are usually confronted early with code analysis and refactoring tools, which are increasingly a part of modern IDEs.

Teachers play an important part in how critically students view their code, but large class sizes prevent them from giving personalised feedback on student solutions. In a previous study we have investigated how teachers would help students to improve their code [142]. We showed experienced teachers a number of functionally correct, but imperfect student solutions and asked them which hints they would give and how they would want a student to refactor code. We compared the teacher hints with the output of professional static code analysis tools, and concluded that these tools are not suitable for giving meaningful feedback to novices.

This paper describes a *tutoring system* to complement human tutoring, giving hints and feedback on exercises in which students improve code. The contributions are (1) the design of a tutoring system that helps students learn about code improvement that better suits the requirements for novice programmers, and (2) the validation of this system based on a comparison to existing tools and teacher input. The system is available online.

Section 5.2 provides background and describes related tools. Section 5.3 describes the method. Section 5.4 shows an example session. Section 5.5 shows the design, which is evaluated and discussed in Section 5.6. Section 5.7 concludes and describes future work.

5.2 Background and related work

This section provides some background on the topic of code quality and code refactoring, and discusses professional tools and their use in education, as well as tutoring systems specifically intended for education.

5.2.1 Code quality and refactoring

The aim of our system is to teach students about code quality in the context of small programs, which are mostly single methods. The definition of code quality we employ revolves around the directly observable properties of source code, such as algorithmic aspects (flow, expressions, language constructs) and structure (decomposition, modularization). Layout and commenting are also relevant aspects, but are beyond the scope of our research because they are not that complex and existing tools are often good enough to support students.

Several terms are used to indicate problems with code quality, such as flaws, issues, violations and the well-known code *smells* as introduced by Fowler [84]. Code smells are characteristics in code that might point at a problem with the design of the code, although it is functionally correct. These problems can have an impact on quality attributes such as maintainability, performance and security.

Code refactoring is improving code step by step while preserving its functionality. Fowler [84] provides a collection of refactorings, mainly focused on the structure of the code. Code Complete [188], a well-known handbook for software construction, describes refactorings on multiple levels: data-level, statement-level, routine-level, class implementation, class interface and system-level. Our study focusses on data-, statement- and routine-level refactorings, which are most relevant for the programs that beginners write. Examples of such issues are code duplication, overly complex or unnecessary constructs, and unsuitable language constructs.

Multiple studies have investigated the presence of flaws in student code that are not functional errors [39], [62], [72], [141], [210]. These studies show that flaws are abundantly present, and there is not a great deal of improvement for certain issues. Because studies show varying results, fixing issues and preventing them in future tasks seems to be highly dependent upon the context. There is also some evidence that the presence of flaws that may point at actual bugs during the process correlates with submitting incorrect code [72].

5.2.2 Professional tools

Relevant professional tools are either static analysis tools or refactoring systems. Both are often integrated in IDEs; static analysis tools are usually also available as a stand-alone tool. Static analysis tools automatically detect quality issues and code smells in code, and generate a list of issues as output, which are usually violated rules. Examples are FindBugs, Checkstyle, PMD, SonarQube, Resharper, and linters. Several IDEs offer support for refactoring code, either as integrated functionality or as an extension that can be installed. Some examples are Visual Studio, Eclipse, and IntelliJ (including IDEs from the same company, such as PyCharm and PHPStorm). A 2012 study showed that refactoring tools are being used infrequently, and that programmers perform quite a lot of low-level refactorings (at the block-level) [195].

Some research exists on the use of professional tools in education. Nutbrown and Higgins [201] have studied whether static analysis tools can be used for summative assessment of student programs. The authors designed a grading mechanism based on the ratings of PMD rules, and compared the automated grades to the grades of instructors. They conclude that the correlation was not strong enough and some manual assessment was still needed, in particular for context-specific issues. Edwards et al. [73] explored whether the FindBugs tool can be used to help struggling students, and found a subset of tool warnings that correlate with incorrect code. However, the authors have not used the tool with students yet.

5.2.3 Tutoring systems

A systematic literature review of tools that generate automated feedback for programming exercises shows there has been a lot of work focussing on the mistakes that students make, but less work on the style and quality of student programs [147]. The study also found that there is much more emphasis on assessment than on guidance to help students improve their programs. Many of these tools are automated assessment tools, which are usually more focussed on grading finished programs. Another type of tool is the Intelligent Tutoring System (ITS), which helps students by guiding them step by step towards a solution [268]. VanLehn [269] found in his experiments that ITSs were nearly as effective as human tutoring. Several ITSs exist for the programming domain [58], offering adaptive feedback (the 'inner loop'), navigational support (the 'outer loop') and several additional features such as programming plan support, reference materials and worked examples.

There are also some tools designed specifically for education that analyse code quality. FrenchPress [32] is a plugin that reports student-friendly messages for a small set of programming flaws. Style++ generates a report with style issues such as commenting, naming and code size [8]. WebTA [265] is a programming environment that reports on failed tests, common errors, and also more stylistic issues. AutoStyle [52] gives stepwise feedback on how to improve the style of correct programs. An experiment with students using AutoStyle has shown improvements, but students also still struggled with improving style [288]. AutoStyle is different from our tool because it relies on historical student data.

5.2.4 Teachers' perspective and conclusion

We recently conducted a study with 30 experienced CS teachers, investigating how they address code quality in general, and having them assess student code of low quality [142]. We asked which hints they would give and how the student should improve the code step by step. We compared their suggestions to the output of PMD, Checkstyle and SonarQube. Based on these findings, other literature, and some of our own observations, we generally consider professional tools in their current form problematic for novice programmers. We summarize the reasons:

i) The terminology and phrasing of messages can be too hard to understand by novices.

ii) All issues are reported at once, which may overwhelm the student and cause cognitive overload.

iii) Not all reported issues are relevant for novices.

iv) Because these tools do not know what the programmer is working on, feedback is not tailored to the current task and its requirements, and the level of the student.

v) IDEs execute a refactoring in a single step, which does not give novices much insight into how a refactoring works.

vi) IDEs may offer code changes that are *possible*, but not necessarily *useful*.

5.3 Method

We address the following research questions:

RQ1 How do we design a tutoring system to learn code refactoring with hints and feedback that closely match how teachers would help students?

RQ2 How does the system compare to related professional tools?

RQ3 To what extent does the system recognise edits to programs?

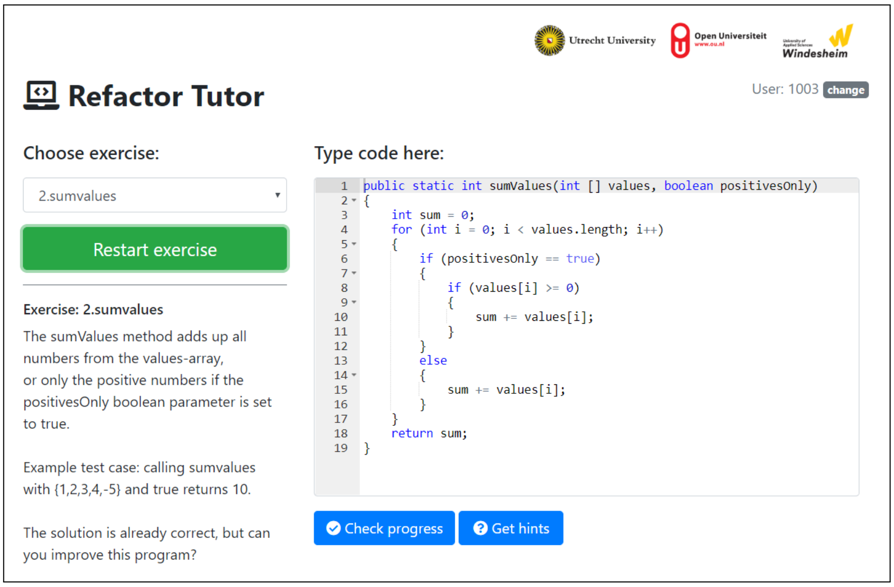
Our method is based on the 'design science' approach, which is described by Wieringa [287] as 'design and investigation of artifacts in a context'. A design cycle is composed of problem investigation, treatment design and treatment validation, and is part of a larger engineering cycle, in which the treatment is implemented and evaluated in the real world. A design cycle will typically be executed multiple times in a project.

This paper focuses on the initial design cycle, for which problem investigation has mostly been done in our preliminary study (see Section 5.2.4). Section 5.5 describes the design of our tutoring system (RQ1). Throughout the design and validation we use two exercises for which we have teacher data: SumValues (Program 1) and OddSum (Program 2a) from our previous study [142].

We evaluate the system in three ways. First, in Section 5.4 we introduce the exercises and show two fictional tutoring sessions, using the actual output of the system. Second, in Section 5.6.1 we compare the system with two professional tools (RQ2): PMD[1], a well-known static analysis tool mostly used for Java, and IntelliJ[2], an increasingly popular Java IDE that provides many code analysis and refactoring options. We ran PMD with the full ruleset on the two starting programs for the exercises, and identified which analyses and suggestions IntelliJ gave on those programs. We compared that output to what teachers suggested in our preliminary study. Additionally, in Section 5.6.1 we show to what extent the behaviour of the system matches with data for the two exercises on how teachers would want the student to solve an exercise step by step (RQ3). For each intermediate step we let our system generate a set of hints, and then checked if the teacher step was in this set. Third, in Section 5.6.1 we disclose some findings on a large group of students using the tutoring system.

[1]`pmd.github.io/pmd-6.9.0`

[2]IntelliJ IDEA Community Edition 2019.2, `www.jetbrains.com/idea`



Exercise: 2.sumvalues

The sumValues method adds up all numbers from the values-array, or only the positive numbers if the positivesOnly boolean parameter is set to true.

Example test case: calling sumvalues with {1,2,3,4,-5} and true returns 10.

The solution is already correct, but can you improve this program?

```
public static int sumValues(int [] values, boolean positivesOnly)
{
    int sum = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (positivesOnly == true)
        {
            if (values[i] >= 0)
            {
                sum += values[i];
            }
        }
        else
        {
            sum += values[i];
        }
    }
    return sum;
}
```

FIGURE 5.1: Web application for the tutoring system.

5.4 A tutoring session

In this section we demonstrate how our system works by showing two tutoring sessions for different exercises. The tutoring system can be accessed online.[3] The target audience are students in higher education who already know the basics of programming (control structures, loops, arrays, methods, classes, etc.), who would typically be CS majors. The system offers exercises in which an exercise specification and a functionally correct, but inelegant program is given. It is the student's task to improve (refactor) the program to make it more elegant/efficient/readable. Currently there are six exercises, and new similar exercises can be added easily.

Figure 5.1 shows a screenshot of the web interface of the system. For the code editor we used the open source Ace editor[4], which supports syntax highlighting, automatic indentation, highlighting matching parentheses, code folding and more. The student has two ways to ask for feedback during programming: CHECK PROGRESS and GET HINTS. The CHECK PROGRESS button

[3]`www.hkeuning.nl/rpt`
[4]`ace.c9.io`

checks the current state of the program and reports on mistakes (syntax errors, failed test cases, known incorrect steps) or successful steps. The system presents hints in a tree structure, of which the first option is shown by default. The student can click on EXPLAIN MORE (denoted by the ↳ symbol) to get a more detailed hint, or click on ANOTHER HINT (denoted by the ⤵ symbol). In the examples below we show parts of the hint tree and fold certain branches for clarity (the ↳ and ⤵ symbols indicate there are more hidden hints), and we only show affected code fragments, omitting details and highlighting the major changes. We show the output of the system as text in a sans-serif font.

5.4.1 Example 1: Sum of values

The first exercise was originally taken from another study [174], with the most popular student solution as the starting program.

Tutor The sumValues method adds up all numbers from the array parameter, or only the positive numbers if the positivesOnly boolean parameter is set to true. The solution is already correct, but can you improve this program?

```
int sumValues(int [] values, boolean positivesOnly) {
    int sum = 0;
    for (int i = 0;i < values.length;i++) {
        if (positivesOnly == true) {
            if (values[i] >= 0) {
                sum += values[i];
            }
        }
        else {
            sum += values[i];
        }
    }
    return sum;
}
```

Student The student asks for a hint right away.

Tutor The tutor responds with a tree of hints that the student expands step by step by clicking on the ↳ icon.

- Can you simplify a boolean expression? ⤵
  - == true may be removed.

- Try to use this example code: `positivesOnly`

Student The student removes the equals true and asks the tutor to check her step, and then asks for a new hint.

```
    for (int i = 0;i < values.length;i++) {
        if (positivesOnly) {
            ..
        }
        ..
    }
```

Tutor That was a correct step, well done!

- Can you simplify the condition in the if? ↴

Student The student notices that when `values[i]` contains 0, addition has no effect.

```
            if (values[i] > 0) {
                sum += values[i];
            }
```

Tutor That was a correct step, well done!

- There is some duplication, can you simplify the if-statements to remove it? ↴
  - Can you combine the conditions, so you only need 1 if? ↳

Student The student incorrectly combines the conditions.

```
        if (! positivesOnly && values[i] > 0) {
                sum += values[i];
        }
```

Tutor The tutor recognises this step as an incorrect attempt to combine two conditions.

Student The student expands the last hint, uses the code example for the first step and asks for the next hint.

```
if((positivesOnly && values[i] > 0) || !positivesOnly)
        sum += values[i];
```

Tutor Can you shorten the expression between the brackets?

```
if (values[i] > 0 || !positivesOnly)
    sum += values[i];
```

Tutor

- Can you replace the for-loop by a foreach-loop? ↳

Student The student rewrites the for-loop.

```
    for (int i: values) {
        if (i > 0 || !positivesOnly)
            sum += i;
    }
```

Tutor Well done, no more improvements left.

5.4.2 Example 2: Odd sum

The starting program for this exercise is composed by mixing a number of actual student solutions collected at the university one of us works at.

Tutor The method oddSum returns the sum of the numbers at odd indices in the array parameter, until the number -1 is seen at an odd index. The solution is already correct, but can you improve this program?

```
int oddSum(int [] array) {
    int total = 0;
    boolean stop = false;

    for (int i = 1;i < array.length;i = i + 2) {
        if (stop == false) {
            if (array[i] != -1) {
                total += array[i];
            } else if (array[i] == -1) {
                stop = true;
            }
        }
        else {
            total = total;
        }
    }
    return total;
}
```

Student The student explores her options by clicking on alternative ⤵, and makes three edits.

Tutor

- Do you know a shorter way to write a calculation? ↳
- You're assigning a variable to itself, is that useful? ↳
- Can you simplify a boolean expression? ↳ ⤵

```
    for (int i = 1;i < array.length;i += 2) {
        if (!stop) {
            ..
        }
        // removed the else-block
    }
```

Tutor All test cases still pass.

- Can you find an else-if condition that is not necessary?
  - We don't need the check in the else-if, because we know it will be true. ↳
- Can we immediately exit from the loop once we're done?

```
            if (array[i] != -1) {
                total += array[i];
            } else {
                stop = true;
            }
        }
```

Student The student removes the if.
Although there are more hints left, we omit the remainder of this session.

5.5 Design

This section describes a technical view of the design of the tutoring system. The tutor supports refactoring strategies consisting of multiple steps, which transform an imperfect solution into an improved solution. Appropriate feedback messages are attached to the steps in the strategy. The tutor supports exercises of class 3 according to the classification of Le and Pinkwart [164], implying the student can follow multiple solution strategies to solve an exercise. The supported programming language is a subset of Java, however, the system uses an internal data type of a fairly generic object-oriented language, so using another language should be feasible.

Sections 5.5.1 and 5.5.2 focus on the implementation of the domain knowledge, and Section 5.5.3 on how the system generates its output.

5.5.1 Implementation

We have developed our tutoring system on top of the Ideas[5] framework for developing ITSs. Tutors built with Ideas (Interactive domain-specific exercise assistants) can provide stepwise automated hints for exercises in various domains, such as mathematics and programming [109]. Various feedback services are offered, such as next-step hints, validation of steps, and showing complete solution paths. *Rules* and *strategies* have to be specified to provide these services. Rules are transformations on the data type of the domain, such as refining or rewriting (parts of) a student program. In the refactoring context, a simple example of such a rule is rewriting `x==true` into `x` (more in Section 5.5.2). Each rule or refactoring should preserve the functionality of the program.

We use *normalisations* to transform (parts of) a program to a *normal form*, by applying a large set of rewrite rules that are not necessarily refactorings, such as changing the order of a calculation (`y+1+x` ≈ `1+x+y`) and removing syntactic sugar (`x+=2` ≈ `x=x+2`).

[5]`hackage.haskell.org/package/ideas`

Normalisation has been used before in programming tutors, such as in the work of Xu and Chee [290], Rivers et al. [225] (using the term 'canonicalization'), and others [88], mainly to recognise more variants of the same program. In the context of refactoring, it also simplifies the definition and implementation of these refactorings, because fewer variants have to be considered. As an example, consider the following contrived code example:

```
if (p) {
    x++;
}
else
    x = x + 1;
```

Our tool would normalise the `{x++;}` and the `x = x + 1;` from the if-else, recognising they are similar, making the if-else obsolete altogether. To generate feedback specific for a student's implementation, normalisations have to be 'undone' [225], which we currently have not implemented, causing the hints with code examples to not exactly match with the student's code at times.

5.5.2 Rules and strategies

The rules are the main building blocks of the tutoring system. We have based the rules on several sources from the literature as well as best practices from software engineering:

- Rewrite steps suggested by teachers, identified in our previous study [142].
- Semantic Style Indicators (SSIs) identified in student programs by DeRuvo et al. [62]. An SSI is defined as 'a pattern of a short sequence of statements that in some circumstances could be considered sub-optimal'. Currently 10 of their 16 SSIs are implemented in our system.
- Semantics-preserving variations (SPVs) that occur in student programs, of which Xu and Chee [290] distinguish 13 types. An SPV changes the computational behaviour of a program while preserving the computational results. We account for several of these variations in our rules and other normalisations (see previous section).
- Rules from professional tools, in particular the code suggestions from PMD and IntelliJ (see Section 5.6).
- Equality rules from arithmetic and logic (e.g. absorption or identity operations).

The IDEAS framework also supports the definition of *buggy rules*, which describe transformations that are not valid and change the computational semantics of the code. At the moment we have only implemented a small set of these buggy rules.

Rules are combined to define more elaborate *strategies*, which describe the step-by-step solution to a problem. In strategies, rules can be combined in sequences, chosen as (prioritised) options, and navigation rules can traverse the abstract syntax tree to apply rules at specific locations. From our teacher input we derived that teachers advise to clean-up code first before moving on to more complex refactorings; we implemented this in the strategy by enforcing cleanup rules before enabling certain other rules.

5.5.3 Feedback services

The system offers the following feedback services:

Hint tree Hints are generated by calculating the first possible steps of the strategy. The hint tree contains all available hints in a hierarchical structure, as described in Section 5.4. A feedback script is used to store the hint messages attached to each step. The script contains key-value pairs that can easily be adjusted by a teacher.

Hints remaining This function calculates the current number of top-level hints.

Diagnosis This function checks the current state of the student program. First, our internal parser processes the program. If that causes an error, the Java compiler from the Oracle JDK is called to get a better error message. Next, the system tries to recognise if a known buggy rule has been applied. If not, test cases are used to verify that the program still has the required functional behaviour. If the system detects that a known rule has been correctly applied, we report that the student just successfully applied that rule. If multiple rules have been applied, or unknown edits have been done, we just report that the program is correct.

TABLE 5.1: All hints reported by more than one teacher, and if they are suggested (●), partially (◐), or not (○).

Description	**Refactor Tutor**	**PMD**	**IntelliJ**
Expressions			
Remove equals true/false	●	●	●
Do not add 0 to sum	●	○	○
Use compound operator +=	●	○	●
Remove self-assignment	●	●	●
Conditionals			
Improve nested if-structure	●	○	○
Remove redundant conditional check	●	○	○
Remove empty if/else	●	◐	●
Loops			
Change for-loop into foreach	●	●	●
Change for-loop into while	●	◐	●
Exit loop when done	●	○	○

5.6 Evaluation and discussion

Section 5.6.1 compares the tutoring system to two professional tools and shows to what extent the system aligns with how teachers would teach students about their code. We also give some preliminary results of an evaluation with students. In Section 5.6.2 the results are discussed and threats to validity are described in Section 5.6.3.

5.6.1 Evaluation

Comparison to professional tools

Table 5.1 shows for each hint mentioned by more than one teacher for the SUMOFVALUES and ODDSUM exercises, if our tool, PMD, and IntelliJ also give that hint. IntelliJ gives similar suggestions as PMD for SUMVALUES by highlighting the associated code fragments. IntelliJ also gives contextual suggestions for code changes when clicking on a particular code fragment, proposing a total of 37 changes for these 12 lines of code, of which 26 are unique. Finally, a set of refactorings are offered for which the programmer has to provide additional input.

Although PMD does not suggest it for ODDSUM, it does have a rule that can transform a for into a while, but only under stricter conditions. What

PMD suggests and is not mentioned by teachers are three 'dataflow anomalies' for the total and stop variables, which is considered a low-priority issue that might not be problematic. Additionally, IntelliJ offers numerous other code edits with no clear goal: 49 changes (26 unique), of which two are concrete warnings: the self-assignment and the `==false`.

Teacher data evaluation

In our preliminary study we asked teachers how they would want a student to refactor programs step by step. We derived high-level rules and hints from these steps, as seen in Table 5.1. In this section we do a more technical evaluation to check if edits to actual program states are recognised.

We analysed data of 27 teachers who provided 76 new program states (excluding the start state) for SumValues. We excluded 11 functionally incorrect programs that our tutor rightly identified, 5 programs that used language constructs our tutor does not support, and 3 other invalid states. We let our system generate all available hints for the remaining 57 programs: for 43 the edits the teacher did in the step were in this hint set (75.4%), for 3 some edits were in the set but some were not (5.3%), and for 11 none of the edits were in the set (19.3%).

For the OddSum exercise 27 teachers provided 66 valid program states. We found that for 41 programs the teacher edits were in the hint set (62.1%), and for 16 the edits were partially in the set (24.2%). We noticed that some teachers solved some issues differently from the concrete hints the system gave, but we mark these edits as successful because the hint does not appear any more.

We can conclude that the hints generated for the majority of the states lead to what the teacher would suggest to do next. Usually multiple hints are available for a state (even for final states), allowing for the various solution paths we saw in the teacher data.

Student evaluation

Because this paper is primarily a software report describing a tool, we have focussed on the functional and technical evaluation of the system. However, we have recently conducted an experiment with 133 students using the system and have performed a detailed analysis of the findings [144]. We can report that the hints help students to solve refactoring exercises and that they value working with the system. We also derive several improvements from this analysis to be incorporated in the next cycle of our design science process.

5.6.2 Discussion

In this section we summarise and discuss how our tool attempts to solve the problems listed in Section 5.2.4:

i) Terminology and phrasing are targeted at novices, and can be adjusted by teachers. Most high-level hints are phrased as questions, which teachers often did as well.

ii) Issues can be shown gradually by letting the student ask to make a hint more specific, or to request a different hint.

iii) We have selected a subset of issues relevant for novices. In future work teachers should be able to switch off certain rules they may find unsuitable for a particular group of students or course.

iv) Although the issues we support go beyond what professional tools detect, we consider exercise- and student-specific feedback to be future work.

v) Our system can guide a student through more complex refactorings step by step.

vi) Our system does not offer edits with no particular goal, instead we offer edits based on the input of experienced teachers.

The proposed system is a practice tool encouraging students to critically assess code and think of alternative solutions. It provides an opportunity to explore other language constructs, and more carefully consider control flow and structure. The hints are suggestions that should trigger further discussion among teachers, and between teachers and students. Although novices often produce verbose code, because they might find it easy to understand (or perhaps it just worked), at a certain time they should move beyond that. We therefore advice the system to be used by students with some programming experience who are ready for the next step.

5.6.3 Threats to validity

We have evaluated the system using two exercises, which may raise questions on the generalisability of the feedback mechanism of the system. However, the majority of the rules are not specific to an exercise, and can be reused for other exercises as well. We do need to expand the set of rules, and also implement a more dynamic way of devising rules, which is described as future work in Section 5.7. Our future work analysing how students use the tutoring system will give us more insight into the effectiveness of the system from their perspective.

5.7 Conclusion and future work

This paper describes the functionality and design of a tutoring system that teaches students about improving code. We have shown a tutoring session in which students receive hints on how to improve an already correct piece of code, and get feedback on the correctness of their steps. We have shown that the behaviour of the tutor matches with how teachers want students to improve their code. We also show that the tutor goes beyond what professional static analysis tools and IDEs do, and better meets the needs of students.

We already conducted an experiment in which students used the system, logging their interactions. We have analysed this data to see if students ask for hints, if they follow up the hints, and if that leads to a good solution. Students also filled out a survey to give their perspective on code quality and the helpfulness of hints, and their opinion of the system.

As part of our design science process, we will iteratively improve the system based on the findings. We plan to add new features, such as having the teacher provide model solutions, from which additional improvement rules can be extracted and dynamically used in the system. Also, we want to add more buggy rules. Future experiments could compare the effects of using the tutoring system to those of professional tools, and study the effect on student code quality in the long run.

Chapter 6

Student Refactoring Behaviour in a Programming Tutor

A modified version of this chapter has been submitted for publication [144].

Abstract Producing high-quality code is essential for professionals working on maintainable software. However, awareness of code quality is also important for novices. In addition to writing programs meeting functional requirements, teachers would like to see their students writing understandable, concise and efficient code. Unfortunately, time to address these qualitative aspects is limited. We have developed a tutoring system for programming that teaches students to refactor functionally correct code, focussing on the method-level. The tutoring system provides automated feedback and layered hints. This paper describes the results of a study of 133 students working with the tutoring system. We analyse log data to see how they approach the exercises, and how they use the hints and feedback to refactor code. In addition, we analyse the results of a student survey. We found that students with some background in programming were generally able to identify issue in code and solve them (on average 92%), that they used hints at various levels, and we noticed occasional learning in recurring issues. They struggled most with simplifying complex control flow. Students generally valued the topic of code quality and working with the tutor. Finally, we derive improvements for the tutoring system to strengthen students' comprehension of refactoring.

6.1 Introduction

Learning to program has been a major area of research for many decades [173], [228]. Researchers have studied the mistakes that students make [42], the misconceptions they have [219], and how we could help them solve their mistakes and correct misconceptions using various teaching approaches [272]. While a major goal is to write code that is functionally correct, it is also important that the code is understandable, concise and efficient. These aspects have been receiving less attention, although we have noticed an increase in studying the non-functional aspects of code.

Studies have found numerous qualitative issues in student code [62], [72], [141], [210]. If we assess students solely based on functional behaviour through test cases, they might not see the importance of writing high-quality code. However, at a certain point, students will have to write larger programs together with other students, for which they need to use and adjust existing code. The need to analyse and refactor low-quality code will then become apparent. It is therefore vital to introduce students to the concept of code quality, raise awareness of its importance, and introduce them to code improvement in an accessible way.

Unfortunately, it is well-known that universities and other learning institutes struggle to give each student the personal attention and feedback they need, due to growth in enrolment figures and limited staffing means [47]. Tools can give students some complementary support. We have designed a tutoring system for programming that teaches students to refactor functionally correct programs. The system focuses on method-level refactorings, such as rewriting a complex expression, removing unneeded code, and replacing a language construct by a more suitable alternative. The functionality of the tutoring system is based on input from teachers and how they would want students to rewrite their code, which we investigated in an earlier study [142]. The system offers refactoring exercises: it is the student's task to rewrite functionally correct code. The student can check the program against test cases to ensure it still works correctly, and ask for hints with increasing detail if he or she does not know how to proceed.

This paper describes the results of a study of 133 students working with the tutoring system. The students are more experienced novices who have already taken programming courses before. We analyse log data to find out how they use the system, and if they are able to identify issues and rewrite the programs. We also describe the results of a student survey on the tutor. We discuss the findings on how students refactor and how the system can help them, as well as future improvements and implications for its use.

The contributions of this work are: (1) a first exploration of how students refactor code; (2) an analysis of the issues they struggle with and which hints they need to deal with those issues; (3) insight into how students value code quality.

This paper is organised as follows: Section 6.2 discusses related work. Section 6.3 describes the tutoring system. Section 6.4 describes the method of this study. The results are shown in Section 6.5, and discussed in Section 6.6. Section 6.7 concludes and describes future work.

6.2 Background and related work

6.2.1 Code quality in education

Our goal is to make students who already have some basic programming knowledge aware of the qualitative aspects of their programs, and teach them how to refactor their programs to make them easier to understand, more efficient, and to make the best use of language constructs. We define code refactoring as improving code step by step while preserving its functionality [84]. We focus on single methods and how to improve their directly observable properties such as control flow, expressions, and choice of language constructs. Layout, naming and commenting are outside our scope.

Maintaining high-quality code is a major topic in the field of software engineering. There is evidence of the persistence of code smells in large software systems (e.g. [262]). For programmers to produce good code it is vital that they learn to be aware of flaws and refactor code as soon as possible. We should therefore incorporate this into our Computer Science curricula, which has not always been done [37]. Kirk et al. [149] studied learning outcomes of 141 introductory programming courses, and found that for 71% of these courses code quality is not part of the learning outcomes. For the courses that do mention code quality in learning outcomes, it is unclear what exactly is being taught.

We also argue that writing code of good quality is closely related to thoroughly understanding the mechanisms of programming. Programming misconceptions can relate to syntactical, conceptual, and strategic knowledge [219]. The latter two categories, conceptual misconceptions (misunderstanding how programming constructs work and how a program is executed) and strategic misconceptions (problems with applying syntactical and conceptual knowledge to a specific task) can be the cause of delivering functional, but incomprehensible or inefficient code.

Several studies investigated non-functional problems in student code [39], [62], [72], [141], [210], from which we learn they are evident and often remain unfixed. Studies show mixed results regarding whether students address issues, which is apparently more the case when they are being assessed on it. In cases where they are not, issues remain present (e.g. [141]).

6.2.2 Tutoring systems for programming

There has been a wealth of studies describing digital tools and environments that help students with learning programming [58], [119], [147]. These tools enable students to learn whenever and wherever they want, and alleviate teacher workload. Many of these systems support the student with (automated) feedback. Feedback is an essential aspect in teaching [107], [239], having the potential to exert great influence on learning, assuming it is delivered in an appropriate manner. Feedback can be *summative* (focused on the outcome) or *formative*, the latter defined by Shute as 'information communicated to the learner that is intended to modify his or her thinking or behavior for the purpose of improving learning' [239]. While assessment tools are mainly focussed on grading programs and giving summative feedback on final submissions, (intelligent) programming tutors help students during the stepwise process of solving exercises.

Intelligent Tutoring Systems (ITSs) have been studied extensively for various domains [268]. VanLehn found that ITSs were nearly as effective as human tutors [269]. ITSs have an *inner loop*, giving stepwise hints and feedback, and updating the student model; some ITSs also have an *outer loop*, selecting a suitable next task. Focussing on the inner loop, several aspects are important for offering feedback and hints [268]. A hint should preferably be given when a student really needs it, but it can be tricky to predict and control this. The suggested step should be analogous to what the teacher would advise, but should also support the student if he or she has already embarked on a certain solution path. The manifestation of hints is often gradual: a general hint, followed by a more descriptive hint, and finally a *bottom-out hint*, which is the actual step to be taken. Feedback usually consists of simply indicating correctness or incorrectness, and error-specific messages. For the domain of programming, several ITSs have been developed that offer features supporting both the inner loop and the outer loop [58]. These Intelligent Programming Tutors often teach a specific aspect (such as recursion), or support building small programs.

6.2.3 Automated feedback on code quality

In a paper that describes the details of the design of our tutoring system, we argue why professional code quality tools are unsuitable for novice programmers [143]. One type of such a tool is the static analyser that reports on violated issues in source code, which can be run inside an IDE or standalone. Examples are PMD, Checkstyle, SonarQube, and linters. Problematic for novices are the technical terminology used by those tools, and the possibly very long lists of reported issues that are not always relevant in the context of novice programs. Other types of tools are refactoring tools and other code transforming tools, often integrated in IDEs. These tools execute refactorings in one step, giving little insight into the inner workings. Some IDEs offer numerous code edits, often without a clear goal. Our tutoring system aims to overcome these problems by offering a student-friendly introduction to code refactoring, using understandable language, and layered feedback for a selection of relevant issues.

Professional tools have been used in the context of education though, and are surely relevant for the more experienced programmer [73], [171], [201]. Jansen et al. [125] have used the Better Code Hub (BCH) tool in education. This tool checks a codebase in GitHub against 10 software engineering guidelines, such as 'write short units of code' and 'write code once'. They found some evidence of increased student code quality, whereas the opinions of students about using the tool varied.

There are also systems analysing code quality specifically aimed at students. FrenchPress [32] is an Eclipse plugin that points students at flaws in their Java programs. The plugin checks code for seven issues related to misuse of fields, the public modifier, booleans, and loop control variables, and presents student-friendly messages. The tool was used in a trial by around 45 students for four exercises, with the result that between 36% and 64% self-reported that they would change their code according to the feedback received.

The Style++ tool provides students with a report on style issues such as commenting, naming and code size [8]. The tool has been used and evaluated by a great number of students, but how students use the tool has not been studied. The authors have seen an increase in the quality of programs, but that could also be partly attributed to the fact that submitted programs had to be approved by the tool.

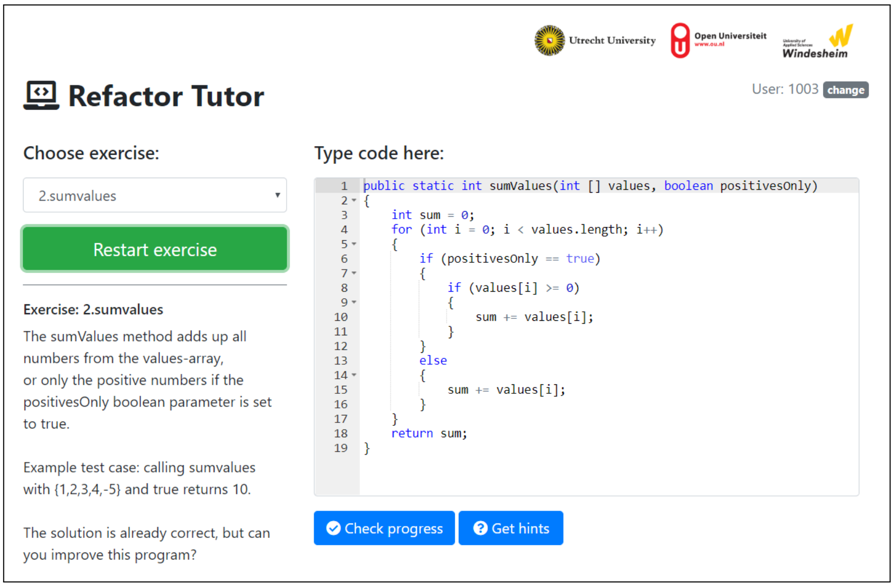
WebTA [265] is a programming environment for students in which they can receive feedback continuously, both during programming and as a grading system for final submissions. The feedback can point to failed tests, common errors, and more stylistic issues referred to by the authors as 'antipatterns'. Examples of the latter that are most related to our system are useless language constructs, and declaring variables or resources inside a loop. The issues are based on what the authors have seen in student submissions. We cannot derive from the paper how feedback on these patterns is presented to students, and the effect of using the tool has not been measured yet.

Qiu and Riesbeck [220] have developed the Java Critiquer, a system that points students to quality problems in their code. The rules in the system have to be incrementally authored by teachers by writing and refining regular expressions and other necessary scripts. We could not find a study on how students used the system.

AutoStyle [52] gives stepwise, data-driven feedback on how to improve the style of correct programs, consisting of teacher-written hints on clustered programs and automatic hints on features that could be added or removed. An experiment with students using AutoStyle has shown improvements, especially in recognising good coding style, but students also still struggled with improving code [288]. AutoStyle is different from our tool because it relies on historical student data, which has the disadvantage that this data is not always available, and requires teachers to write hints beforehand. Moreover, different from the AutoStyle studies, we have performed a quantitative study of log data on how students approach code improvements at the method level. We are not aware of any other such study.

6.3 The Refactor Tutor

This section describes the tutoring system for refactoring. The target audience of the system are students (typically CS majors) who already know the basics of programming (control structures, loops, arrays, methods, classes, etc.). The system offers exercises in which a problem specification and a functionally correct, but problematic program is given. The student's task is to improve (refactor) the program to make it more concise, efficient, and understandable.



Exercise: 2.sumvalues

The sumValues method adds up all numbers from the values-array, or only the positive numbers if the positivesOnly boolean parameter is set to true.

Example test case: calling sumvalues with {1,2,3,4,-5} and true returns 10.

The solution is already correct, but can you improve this program?

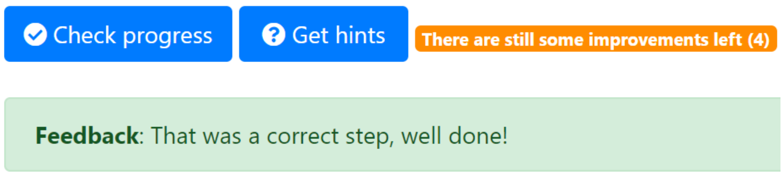
```java
public static int sumValues(int [] values, boolean positivesOnly)
{
    int sum = 0;
    for (int i = 0; i < values.length; i++)
    {
        if (positivesOnly == true)
        {
            if (values[i] >= 0)
            {
                sum += values[i];
            }
        }
        else
        {
            sum += values[i];
        }
    }
    return sum;
}
```

FIGURE 6.1: Web application for the tutoring system.

Figure 6.1 shows a screenshot of the web interface of the system. To implement the code editor we used the open source Ace editor[1], which supports syntax highlighting, automatic indentation, highlighting matching parentheses, code folding, and more. The student has two ways to ask for feedback during programming: CHECK PROGRESS and GET HINTS. The CHECK PROGRESS button checks the current status of the program, resulting in one of the following diagnoses:

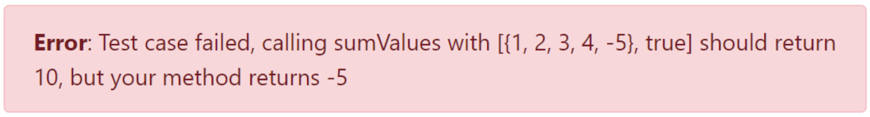
- **Expected**. The student has just executed a step recognised by the system, as shown in Figure 6.2.
- **Correct**. The submitted program is functionally correct, but the system does not know what the student has done.
- **Similar**. The student has not changed anything, or went back to the previous state after doing an incorrect edit.
- **Buggy**. The student has taken a known incorrect step.

[1] `ace.c9.io/`



Feedback: That was a correct step, well done!

FIGURE 6.2: Feedback acknowledging a correct step and an indicator of the number of improvements left, shown after clicking the CHECK PROGRESS button.



Error: Test case failed, calling sumValues with [{1, 2, 3, 4, -5}, true] should return 10, but your method returns -5

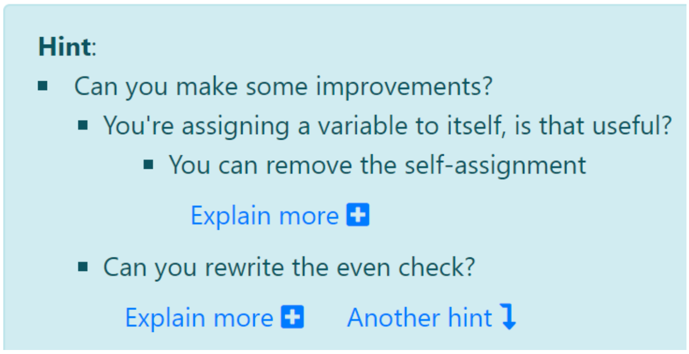
FIGURE 6.3: Error message for a failed test case.

- **Failed test case**. The student has changed the functionality of the program. The first failed test case is shown, as can be seen in Figure 6.3.
- **Compiler error**. The student has used an unsupported language construct or made a syntax error. In the latter case we show the number of errors and the first error message from the Oracle Java compiler.

For the first three diagnoses (dealing with a correct program) the system also shows how many improvements there are left (see Figure 6.2).

The GET HINTS button generates hints for the current program state, after checking that the program is still correct by executing a progress check in the background. If it is not correct, the diagnosis is shown, otherwise hints are generated. The system presents hints in a tree structure, of which the first option is shown by default. The student can expand a hint (by clicking EXPLAIN MORE) to get a more detailed hint, or click on ANOTHER HINT to get a different hint. An example of a partly collapsed hint tree is shown in Figure 6.4.

The hints the system gives are based on rules, derived from teacher suggestions for a set of imperfect student programs, collected in our earlier study that investigated how teachers would give feedback on improving code [142]. Other rules are based on other studies [62], [290], a subset of rules from professional static analysis tools considered suitable for novices, and equality rules from arithmetic and logic. The system also contains some 'buggy rules' that


Hint:

- Can you make some improvements?
  - You're assigning a variable to itself, is that useful?
    - You can remove the self-assignment

      Explain more
  - Can you rewrite the even check?

    Explain more Another hint

FIGURE 6.4: Partly collapsed hint tree.

describe common mistakes. The tutoring system, its motivation, design and example sessions are described in more detail elsewhere [143].

The system can be accessed online[2], and consists of a web-based interface and a backend that processes JSON requests and replies with JSON responses. All requests and responses are logged in a database on the server. These requests are: retrieving the exercises, loading an exercise, checking progress, asking for hints, and expanding a hint. The current state of the code is attached to the requests. The system was tested before the experiment by two teachers and one teaching assistant.

Six exercises are offered with varying difficulty. The programs to be refactored for each exercise contain between two and four quality issues at the start, as shown in Table 6.1. These issues correspond to the rules described earlier. Some issues become apparent after dealing with initial issues. Certain issues reappear in a later exercise, sometimes in a slightly different way.

For the last exercise (exercise 6) no starting code is given, only a description of its functionality and a set of test cases. Feedback and hints are available for all exercises, and are generated dynamically for the current state of the student code. Programs 1–5 show the start code and description for the first five exercises. Exercise 2 is taken from another study [174], exercise 1 and 6 are from the Codingbat website[3], and exercise 3, 4 and 5 are our own.

[2]`www.hkeuning.nl/rpt`
[3]`codingbat.com/java`

Table 6.1: The issues that appear (•) or may appear later (◦) in the exercises.

	Ex1	**Ex2**	**Ex3**	**Ex4**	**Ex5**
Expressions					
Simplify boolean expression	-	•	•	-	-
Use compound operator	•	-	•	•	-
Optimise calculation	-	•	-	-	-
Remove self-assign	•	-	•	-	-
Improve odd/even check	•	-	-	-	-
Branching					
Remove duplication	-	•	-	-	-
Remove useless else if	-	-	•	-	•
Remove empty statement	◦	-	◦	-	-
Extract from if else	-	-	-	•	-
Loops					
For to foreach	•	•	-	-	-
For to while	-	-	-	-	•
Exit loop early	-	-	◦	-	◦
Replace loop by calculation	-	-	-	•	-

6.4 Method

Our research questions are:

RQ1 How do students solve refactoring exercises? Which steps do they take, and which mistakes do they make?

RQ2 When do they ask for a hint? For which issues do they need hints? How do they respond to a hint?

RQ3 What do students think about working with a refactoring tool?

6.4.1 Study design

We conducted the experiment at Windesheim University of Applied Sciences in the Netherlands in the week of 14 October 2019. This week was the final week of the term for (mostly) second-year IT-students specialising in Software Engineering, who were all doing a C# programming course. All students had followed at least two previous programming courses: web programming in

PHP, and object-oriented programming in Java. A minority of the students are specialising in other fields, such as embedded systems or business IT.

The C# programming course consists of 7 lecture weeks, followed by a practical exam (programming exercises on a laptop). During the course the students work on a large assignment that they must complete in order to pass the course. The course assumes the programming knowledge from the PHP and Java courses, in which all basic language constructs (variables, branching, loops, methods, object-oriented concepts, etc.) are discussed. The C# course transfers to the C# language, first discussing the C# equivalents for known language constructs, and then handling more advanced topics such as delegates and events, generics, LINQ/functional programming constructs and unit testing.

The course was taught in groups of approximately 25 students. There were seven groups that were taught by four different lecturers. One of the researchers is a colleague of these lecturers, but did not teach the course that year. The experiment required one hour per group, in which the lecturer and one researcher were present.

In this experiment all students worked with the system. We did not offer a pre- and post-test, because we want to focus on how students use the tool, how they respond to feedback and hints, and how they edit the code.

The experiment consisted of three parts: the first 15 minutes were used to introduce the topic of code quality, to demonstrate the tool, and to explain the experiment. The students were asked to fill in a form to provide consent for using their data, and to give some general information such as age, gender and previous knowledge. All students were given a unique ID to login to the system. If they did not give consent to use their data, they could log in anonymously and still do the exercises.

In the next 30 minutes the students individually worked on six exercises in the system on their own laptop. The students received an information sheet with some notes on how the system works, and the Java syntax for certain language constructs supported by the system. The interface of the tutoring system and its feedback is in English. This is not the students' native language, but the students use other English study material as well. The researcher helped with questions on how the tool works, but not with questions on how to solve the exercises. The students were also asked not to consult other students.

The final 15 minutes were spent to fill in a short survey with questions about their experience with the system. The survey had three Likert-scale questions and three open questions.

6.4.2 Analysis

After the experiment the log database was cleaned by removing records from anonymous IDs and IDs that did not give consent. We also removed activity from outside the 30 minutes of the experiment, extended with an overrun of 10 minutes. We checked the dataset for abnormal activity and removed log sequences with more than 50 identical actions per minute. We suspect some of those sequences are not from the user interface, but were fired by a script.

Because the number of program states in this study is large, we performed an automated analysis. To check whether our system correctly identifies programs as 'ready', we manually inspect a random subset of 10% of the ready end states for each exercise.

We analysed the open survey questions in which we asked for pros and cons of the system by grouping similar comments. The end of the survey had a text field for additional comments that often contained pros and cons as well. We moved those answers to either the pro or con section, depending on their content. The literal texts we show in the results section are translated from the students' native language.

6.5 Results

In total 143 students (of around 200 enrolled, and some who had to retake the course) attended the lecture in which the experiment took place, of which 135 students gave consent to use their data, and 8 did not. For 1 of the 135 students that gave consent we could not find log data, and all records from 1 student were removed from the data set due to abnormal behaviour (sending an excessive number of requests), resulting in 133 students for the analysis.

The students were between 17 and 31 years old (average 20.5, median 20, 1 student did not provide age). A total of 86% identified as male, 8% as female, and 5% did not provide their gender. All but 1 student attended the web programming course and 130 (98%) passed. All but 4 students attended the Java programming course and 122 (92%) passed. Of all students, 10% reported they had no programming experience besides school, 60% had a little, 13% had a lot, and 23% had experience from a previous education. A total of 122 students (92%) are in their second study year and have chosen a software engineering

Table 6.2: Summary of tutor events, between parentheses the number of unique students.

Exercise	**Startups**		**Diagnoses**		**Generated hint trees**		**Hint expansions**	
1 Even	228	(133)	1537	(133)	205	(97)	122	(53)
2 SumValues	222	(133)	2539	(133)	456	(112)	573	(99)
3 OddSum	167	(123)	1329	(118)	261	(86)	226	(59)
4 Score	120	(103)	775	(97)	121	(51)	88	(30)
5 Double	93	(76)	402	(69)	55	(27)	26	(10)
6 HaveThree	75	(60)	376	(44)	18	(11)	14	(6)
Total	*905*		*6958*		*1116*		*1049*	

profile, 2 students (2%) are retaking the course, and 9 students (7%) are IT students with a different profile that are taking the course as part of an elective minor.

The students produced 12,254 log entries. The main events are summarised in Table 6.2. Adding up all diagnose-events produces a total of 6,985 program states submitted by students.

6.5.1 Solving exercises (RQ1)

Figure 6.5 shows the number of students who started and completed the exercises. We only include attempts for an exercise with at least one action (such as check or hint request). We define 'ready' as the system not having any hints left, 'gave up' as students who performed at least one action for an exercise, but did not deal with all issues and moved on to a new exercise, and 'time up' as working on the exercise when the experiment stopped. For 'gave up' we also calculated the number of hints remaining for the last known and valid program state. The first two exercises were started by all students, with a gradual decline for the subsequent exercises. In total 52% of the students got to exercise 5, and exercise 6 was attempted by only 33%. We can see from Table 6.2 that a larger percentage (45%) started exercise 6, but because students had to write code from scratch, they probably did not have enough time left to do a check or hint action. Exercise 2 and 3 have by comparison the most students that did not complete it.

Table 6.3 shows for each exercise how much time students spent working on them, excluding timeups. For all exercises except exercise 6, students

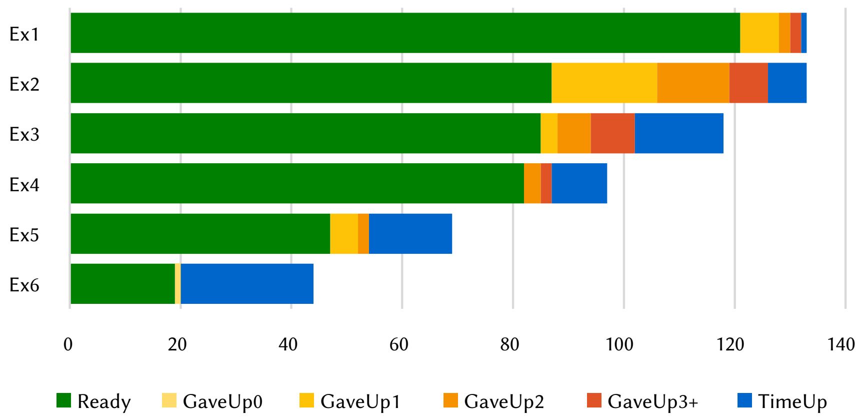


FIGURE 6.5: For each exercise the number of students that solved all issues (green), did not solve all issues and continued with another exercise (yellow/orange/red, depending on the number of open issues), and were working on an exercise when their time was up (blue).

TABLE 6.3: Time on task.

Exercise	**Min**	**Max**	**Mean**	**Median**
1 Even	1:33	17:55	7:06	6:44
2 SumValues	1:38	27:34	10:12	9:33
3 OddSum	1:54	16:15	6:41	6:23
4 Score	1:11	14:06	4:41	4:14
5 Double	1:05	13:45	3:49	3:20
6 HaveThree	3:41	14:39	7:43	6:58

worked on it a minimum of between 1 and 2 minutes. Exercise 2 stands out because students spent more than average time working on it.

Next we zoom in on the diagnoses students received while working on the exercises. A diagnosis is calculated when a student clicks on CHECK PROGRESS, but also when a student asks for a hint, because hints are only generated once the current program state is functionally correct. Again, we only include sessions in which the student has performed at least one action. We exclude timed-out sessions because their diagnose count would not be representative for a full session. Figure 6.6 gives insight in how many diagnoses a student receives, which varies per exercise with a median between 4 and 17. We also identify several outliers. We manually inspected some of these sessions, in

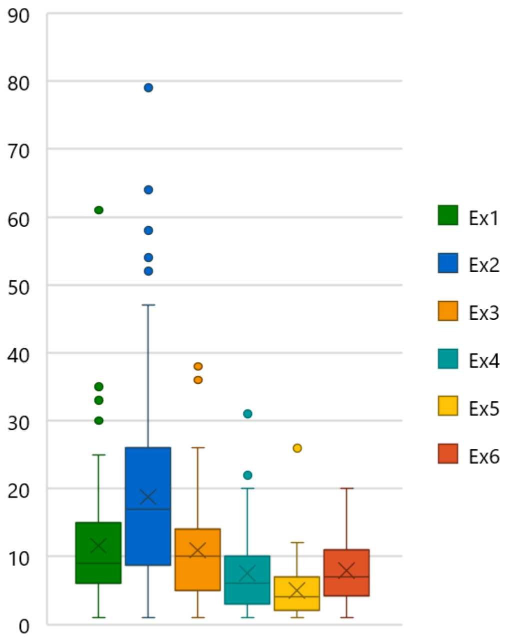


FIGURE 6.6: For each exercise a boxplot showing the distribution of the number of diagnoses per student.

which we often noticed a large number of identical error diagnoses given in a short time frame, giving the impression the student kept clicking the button again and again.

Figure 6.7 shows the distribution of the various types of diagnoses for all sessions. Table 6.4 shows the diagnoses with a functionally correct result, and Table 6.5 the diagnoses with a problematic result. For exercises 1 to 4 we see a fair amount of expected (a single recognised step) diagnoses in relation to correct diagnoses (multiple or unknown steps resulting into a correct program). This is much less the case in exercises 5 and 6. Exercise 6 shows a rather different distribution with many more failed diagnoses, which is to be expected because this exercise required writing code from scratch.

Considering problematic diagnoses, failed tests was a major category (22%) for exercise 2. Failed tests can be an actual test case failing, the inability to execute a test because the student changed the method header, or some other runtime error such as a suspected infinite loop. Although failed tests is a major

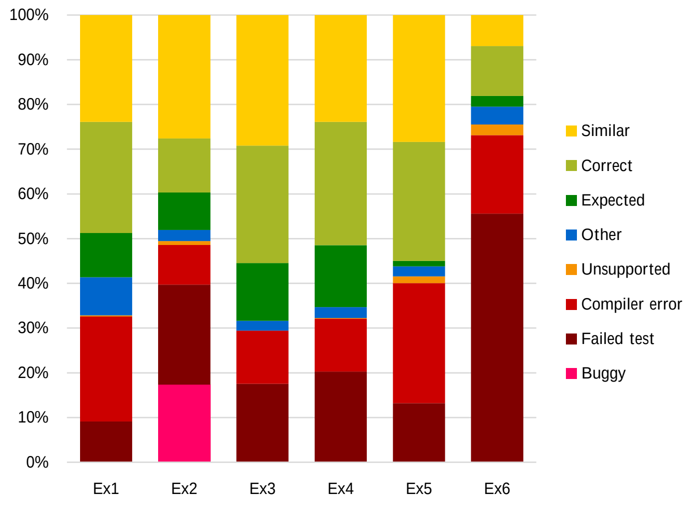


FIGURE 6.7: For each exercise the distribution of diagnoses.

category for all exercises, compiler errors are also pervasive. Note that one compiler error instance could contain multiple compiler errors. The number of compiler errors is the largest for the first exercise, which could be attributed to getting used to the Java syntax. We inspected the most common compiler error messages and noticed that the message generated when accidentally using the C# syntax for a foreach-loop was at the top of the list. Exercise 5 also has quite a large number of compiler errors compared to the other exercises, with the main error being a variable that 'might not have been initialized'.

We also noticed students using language constructs unsupported by our

TABLE 6.4: For each exercise the total number of diagnoses for functionally correct solutions, and between parentheses the number of unique students receiving that diagnosis.

Exercise	**Expected**		**Correct**		**Similar**	
1 Even	152	(98)	382	(132)	367	(115)
2 SumValues	213	(103)	307	(120)	700	(116)
3 OddSum	172	(93)	349	(113)	388	(98)
4 Score	107	(61)	214	(92)	185	(62)
5 Double	5	(3)	107	(63)	114	(44)
6 HaveThree	9	(7)	42	(24)	26	(15)
Total	*6589*		*1401*		*1780*	

Table 6.5: For each exercise the total number of problematic diagnoses.

Exercise	**Failed Test**		**Compiler error**		**Unsup-ported**		**Other**	
1 Even	140	(45)	361	(101)	4	(3)	131	(46)
2 SumValues	567	(108)	226	(82)	22	(19)	63	(28)
3 OddSum	233	(68)	158	(71)	0	(0)	29	(16)
4 Score	157	(46)	92	(48)	1	(1)	19	(9)
5 Double	53	(27)	108	(44)	6	(3)	9	(5)
6 HaveThree	209	(40)	66	(28)	9	(5)	15	(6)
Total	*1359*		*1011*		*42*		*266*	

Table 6.6: For each exercise the failed test case seen by most students, and its possible cause.

Exercise	**Failed test**	**Students**
1 Even	[{1, 2, 3, 4, 5}] should return 2, but your method returns 0	23 (17%)
2 SumValues	[{1, 2, 3, 4, -5}, true] should return 10, but your method returns 5	92 (73%)
3 OddSum	[{44, 12, 20, 1, -1, 3, 5, -1, 99, 4}] should return 16, but your method returns 0	20 (20%)
4 Score	[2, 7] should return 8, but your method returns 5	24 (28%)
5 Double	[1000.0, 4] should return 18, but your method returns 0	10 (19%)

tutoring system. For example, in six events a student used the `?:` ternary operator, which is a shorthand for an if-statement. There were also 20 events with a program using the & operator, which is an unusual choice and not needed for any of the exercises. Other examples were calling Java library methods, casting, and defining multiple methods.

The ‘other’ category are diagnoses that were less clear, and contained some internal errors caused by bugs. Students using constructs such as `return sum+=1` and `if(x=1)` with assignments in (boolean) expressions received messages that should have been clearer: the system dealt with these constructs insufficiently. We also noticed that students often (mostly in the first exercise) mixed a foreach with standard array indexing, causing an error message not suitable for the actual problem. The ‘buggy’ category for exercise 2, accounting for 17% of the diagnoses, will be explained in Section 6.5.2.

6.5.2 Hint seeking (RQ1 and RQ2)

Figure 6.8 shows how many hints students have seen for each exercise, including the top-level hint of the initial hint tree, and hint expansions and alternatives. Timed-out sessions and sessions with no activity were excluded. The most hints were seen for the second exercise, with a median of 7.5. The medians for the other exercises are at most 2, but there are quite some outliers with students viewing many more hints. We observe a decreasing number of hints for the last four exercises.

Next, we focus on the individual exercises to investigate which hints were shown and if students were able to solve issues with or without help. We exclude students whose time was up for a particular exercise. Tables 6.7 to 6.11 show the main hints for exercises 1 to 5, and are discussed in detail in the next subsections. Recurring issues (sometimes in a slightly different manifestation) are indicated with the ↻ symbol. Issues that might come up later during the exercise are marked with the ⋆ symbol. For these later issues we only look at students who received a hint for it. Some other issues that came up incidentally because of undesirable student edits are omitted from the tables. Table 6.6 shows the failed test case (input/output pair) that was seen by the largest number of distinct students. We investigate causes in the next few sections.

Exercise 1

Table 6.7 shows the main hints associated with the first exercise, organised in the tree structure in which the hints were shown to the students. All hints in this table are known issues and form the complete hint tree generated for the start program. We omitted four additional hints that were dynamically generated for students who made undesirable adjustments. The table shows the number of students that have seen the hint, and how many students have solved the issue (not including those whose time was up). The issue for which most hints were generated is replacing the for-loop by a foreach-loop, which was seen by 61% of the students. Almost a third of those students expanded that hint to see the code example. Some students did not make this change to the code in the end. A total of 43% of the students saw the hint on rewriting the even check, for which more than half expanded to see more detail, and almost a third of those saw the code example. A small number of students did not solve the issue. Removing a self-assignment statement and using a compound operator was easier to do for students without help.

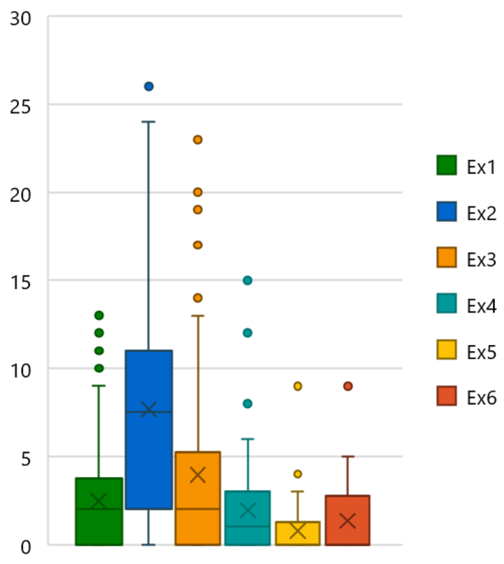


FIGURE 6.8: For each exercise a boxplot showing the distribution of the number of hints seen per student.

Our manual inspection of 10% of the ready programs confirmed that all issues were dealt with, although one student used +=1 instead of ++ and one program contained an unnecessary declaration, something the system cannot detect yet. The most common failed test case was often caused by replacing the statement `count=count` by `return count`, which is a curious misconception. We also saw some students using %2==2, which can never be true.

Exercise 2

Table 6.8 shows the main hints for the second exercise, for which we already noticed in the previous section that students struggled more. We omit hints that were only seen by one student. Most of the main hints were seen by more than half of the students, except replacing the for-loop by a foreach-loop. This was a recurring issue that, compared to the previous exercise, twice as many students changed without seeing a hint. In particular, students struggled a lot with the duplicated addition. More than two thirds of the students viewed some hint on this topic, and the majority of those students clicked all the way through to the code example. Those students were most likely (88%) to deal

PROGRAM 1: Start code for exercise 1.

```
 1  int countEven(int [] values) {
 2    int count;
 3    count = 0;
 4    for (int i = 0; i < values.length; i++)
 5    {
 6        if (values[i] % 2 != 1) {
 7          count = count + 1;
 8        }
 9        else {
10          count = count;
11        }
12    }
13    return count;
14  }
```

Description: The countEven method returns the number of even integers in the values-array.

Example test case: {1,2,3,4,5} returns 2. You don't have to deal with negative numbers.

The solution is already correct, but can you improve this program?

with the issue, but in the end only 75% of all students did. The test case that failed the most was seen by 73%, and was related to this problem.

The tutoring system contains a 'buggy rule' related to merging the two conditions for adding the array value (lines 5–12). This buggy rule detects the common mistake of incorrectly combining the conditions, resulting in the code below:

```
if (positivesOnly && values[i] >= 0) {
    sum += values[i];
}
else {
    sum += values[i];
}
```

We implemented this buggy rule because even teachers made this mistake, which we noticed when studying their refactoring sequences. However, our system reported the name of this issue ('buggycollapseif') instead of a more

TABLE 6.7: Hints seen and solved for exercise 1 (n=132). The 'solved by' column calculates the percentage based on the 'seen by' column.

Type of the most detailed hint seen	**Seen by**		**Solved by**		**Total solved**
Replace for by foreach-loop (line 4-12)					
No hint	52	(39%)	49	(94%)	
Top-level hint	54	(41%)	48	(89%)	121 (92%)
→ Code example	26	(20%)	24	(92%)	
Rewrite the even check using `==0` (line 6)					
No hint	75	(57%)	73	(97%)	
Top-level hint	19	(14%)	19	(100%)	128 (97%)
→ Detailed hint	26	(20%)	25	(96%)	
→ Code example	11	(8%)	10	(91%)	
Remove useless else with self-assign (line 9-11)					
No hint	119	(90%)	118	(99%)	
Top-level hint	6	(5%)	5	(83%)	130 (98%)
→ Detailed hint	4	(3%)	4	(100%)	
→ Code example	3	(2%)	3	(100%)	
Use the compound ++ operator (line 7)					
No hint	124	(94%)	123	(99%)	
Top-level hint	4	(3%)	4	(100%)	130 (98%)
→ Detailed hint	2	(2%)	2	(100%)	
→ Code example	2	(2%)	1	(50%)	

informative message, which caused confusion by students that we noticed during the experiment and while reading the surveys. Excluding timed-out sessions, we counted 415 diagnoses of this issue for 80 distinct students, who saw the message between 1 and 40 times with a median of 3. Of these students, 49 (61%) solved the issue, which is lower than the 75% of all students. We suspect that showing an explanatory message would increase the number of students solving the issue by understanding what went wrong, and is high on our list of improvements.

Many students also saw some hint on improving a boolean expression, such as removing `==true`, which most students did without a hint. Another issue in this category that came up during refactoring, was simplifying a complex expression, usually appearing after merging the two conditions for adding

PROGRAM 2: Start code for exercise 2.

```java
int sumValues(int [] values ,
              boolean positivesOnly) {
  int sum = 0;
  for (int i = 0;i < values.length;i++) {
    if (positivesOnly == true) {
      if (values[i] >= 0) {
        sum += values[i];
      }
    }
    else {
      sum += values[i];
    }
  }
  return sum;
}
```

Description: The sumValues method adds up all numbers from the values-array, or only the positive numbers if the positivesOnly boolean parameter is set to true.

Example test case: calling sumvalues with {1,2,3,4,-5} and true returns 10.

the array value. Students probably used the code example from the duplication hint, which first showed the disjunction of the two cases. Most students needed the code example to see how the expression could be simplified, which involved applying logic rules. Of all students, 63% saw the hint on using > instead of >=, often also expanding to the code example. Students that only saw the top-level hint were least likely to address the issue. In the end 90% changed the operator.

The manual inspection of 10% of the 'ready' programs revealed that two students chose an alternative solution by always adding a value to the sum, and either setting that value to 0 in certain cases, or later subtracting the value if it should not have been added. We have seen this type of solution in teacher solutions as well, and it would be a good candidate to discuss its pros and cons. In two cases the >= was still present but should have been detected, and one of those cases still contained the duplication in a different construct. The other 5 programs contained no issues.

Table 6.8: Hints seen and solved for exercise 2 (n=126). * We do not calculate total solved for incidental issues, because not all students had to deal with them.

Type of the most detailed hint seen	Seen by		Solved by		Total solved
Remove duplication by simplifying ifs (line 5-12)					
No hint	35	(28%)	26	(74%)	94 (75%)
Top-level hint	10	(8%)	5	(50%)	
→ Detailed hint	12	(10%)	2	(17%)	
→ Code example	69	(55%)	61	(88%)	
Boolean expression issues					
No hint	43	(34%)	40	(93%)	
Top-level hint	14	(11%)	13	(93%)	
→ Detailed hint for remove ==true (line 5)	11	(9%)	11	(100%)	123 (98%)
→ Code example	0	(0%)	–		
→ Detailed hint for complex expression ⋆	5	(4%)	4	(80%)	n.a.*
→ Code example	54	(43%)	51	(94%)	
Use > to avoid useless calculations (line 6)					
No hint	47	(37%)	43	(91%)	114 (90%)
Top-level hint	23	(18%)	18	(78%)	
→ Detailed hint	18	(14%)	17	(94%)	
→ Code example	38	(30%)	36	(95%)	
Replace for by foreach-loop (line 4-13) ↻					
No hint	113	(90%)	99	(79%)	109 (87%)
Top-level hint	5	(4%)	4	(80%)	
→ code example	8	(6%)	6	(75%)	

Exercise 3

Table 6.9 shows the hint-seeking behaviour for exercise 3. The issue not dealt with by the largest number of students (8) was removing a useless check in the else-if part of an if-else. All but one of the students that saw a hint for that issue fixed it, and for half of those students the top-level hint was sufficient.

The code for the exercise also contained multiple recurring issues, such as a useless else with a self-assignment similar to the code for exercise 1. However, a slightly smaller percentage (95% versus 98%) dealt with it. A possible reason for fewer students solving it might be that the code for exercise 3 was much more complex, distracting from these useless lines of code. However,

TABLE 6.9: Hints seen and solved for exercise 3 (n=102).

Type of the most detailed hint seen	**Seen by**	**Solved by**	**Total solved**
Remove useless `else` with self-assign (line 15-17) ↻			
No hint	83 (81%)	78 (94%)	
Top-level hint	15 (15%)	15 (100%)	97 (95%)
→ Detailed hint	0 (0%)	–	
→ Code example	4 (4%)	4 (100%)	
Use the compound += operator (line 4) ↻			
No hint	50 (50%)	45 (90%)	
Top-level hint	18 (18%)	17 (94%)	95 (93%)
→ Detailed hint	18 (18%)	18 (100%)	
→ Code example	16 (16%)	15 (94%)	
Replace ==`false` by not (!) operator (line 6) ↻			
No hint	82 (80%)	75 (94%)	
Top-level hint	12 (12%)	12 (100%)	95 (93%)
→ Detailed hint	5 (5%)	5 (100%)	
→ Code example	1 (1%)	1 (100%)	
Remove useless check in else-if (line 11)			
No hint	83 (81%)	76 (92%)	
Top-level hint	10 (10%)	10 (100%)	94 (92%)
→ Detailed hint	4 (4%)	4 (100%)	
→ Code example	5 (5%)	4 (80%)	
Exit loop when done ⋆			
Top-level hint	22 (21%)	21 (95%)	
→ Detailed hint for solution with variable	2 (2%)	1 (50%)	
→ Code example	15 (15%)	12 (80%)	n.a.
→ Detailed hint for solution with condition	0 (0%)	–	
→ Code example	2 (2%)	0 (0%)	
Replace break by loop condition ⋆			
Top-level hint	4 (4%)	4 (100%)	n.a.
→ Code example	8 (8%)	8 (100%)	

Program 3: Start code for exercise 3.

```
static int oddSum(int [] array) {
  int total = 0;
  boolean stop = false;
  for (int i=1; i < array.length; i=i+2)
  {
    if (stop == false) {
      if (array[i] != -1) {
        total += array[i];
      }
      else
        if (array[i] == -1) {
          stop = true;
        }
    }
    else {
      total = total;
    }
  }
  return total;
}
```

Description: The method oddSum returns the sum of all numbers at an odd index in the array parameter, until the number -1 is seen at an odd index.

Example test case: {44, 12, 20, 1, -1, 3, 5,-1, 99, 4} returns 16 (12+1+3)

we also observe that the top-level hint was more often helpful compared to exercise 1. The `==false` is also changed less often than the `==true` from exercise 2, but that might be because this change is slightly more difficult. The compound operator += hint was seen by 50% of the students, which could also be due to the fact that this hint appeared early in the ordering of hints. Students asked for various levels of detail, and managed to use the operator in all but 2 cases. We expect that students are less familiar with this operator than the more common ++ operator.

The loop in this exercise can terminate once -1 is seen, on which hints are given once all clutter has been removed. Of all students, 38% have seen a hint for this. The system offers two alternative solutions, of which the first (adding the stop-variable to the condition of the loop) has been seen by more students than the second (directly checking if the current array value contains -1 in the loop condition). The majority of students seeing these hints have also dealt with the issue.

The manual inspection of 10% of the 'ready' programs showed that all issues were dealt with, except some issues with exiting from the loop once -1 was seen. Some students used a return or break, which is a good alternative to the suggestions of the system. In a few of those cases the student still maintained the stop variable, which is not strictly necessary. There was also a solution with two continue statements, which we consider a somewhat far-fetched solution. We noticed that the for-loop was transformed into a while in one of these programs, which is good also considering the more complex loop condition. One student used a foreach while maintaining a counter for skipping even indices: we would consider converting this into a buggy rule.

Exercise 4 - 6

Table 6.10 shows the hint usage for exercise 4. The start code for this exercise contains an unnecessary for-loop that can be replaced by a simple calculation. Almost half of the students saw the top-level hint for this exercise, and quite a few also requested the code example. All of these students managed to replace the loop by a calculation. The code also contained an if-else statement with the same code at the end of both the if-part and the else-part, which can be moved after the if-else. Around a quarter of the students viewed the hint, of which the majority needed more detail. Only one student did not fix it despite the hints. The recurring compound `-=` operator was now replaced more often than in the previous exercise, and a much smaller percentage needed a hint for this issue. The most failed test case was often caused by inverting `day==6 || day==7` to `day!=6 || day!=7`.

In a manual inspection of 8 ready end states we noticed some calculations that can be combined, on which we do not give hints yet. This is not a major issue (calculating in steps can be clearer in certain cases), but we will consider adding this. We also saw 2 solutions in which calculating and returning could have been simpler, and we do not give hints on a composed negated expression. The other issues for this exercise were all solved.

Table 6.11 shows the hints for exercise 5. The start code contains a recurring issue: the useless check in the else-if that also appeared in exercise 3. For this exercise, a larger percentage of students solved it, and the same percentage viewed a hint (it was the first available hint), but only 1 student requested more detail for this exercise. Earlier exercises suggested to use a

Program 4: Start code for exercise 4.

```
public static int calculateScore
          (int changes, int day) {
  int score = 10;
  for (int i = 0; i < changes; i++) {
    score = score - 1;
  }
  if (day == 6 || day == 7) {
    return score;
  }
  else {
    score = score - 3;
    return score;
  }
}
```

Description: The calculateScore method calculates the score for a train trip. The highest score is 10. The score is based on the number of changes and the day of the week (Monday is 1, Sunday is 7).

The Railway Company has designed the following calculation:
Base score: 10
For each change: -1
Trip on a weekday: -3

Example test case: for a trip with 2 changes on a Wednesday (day 3), calculateScore(2, 3) returns a score of 5 (10-2-3).

foreach instead of a regular for-loop, whereas this exercise suggests using a while because of the unknown number of iterations and the more complex stop condition (however, solving it with a for with only the calculation inside would not trigger that hint anymore). In the end, 87% changed the loop. A few students (3) later received a hint on eliminating the break by using a suitable while condition; students not receiving the hint probably did this straight away.

We found no problems with 3 of the 5 end states we inspected manually. The other 2 contained extra checks that could have just been the loop condition.

For the final exercise 6 students had to write the solution from scratch for a slightly more complex problem requiring a loop and some conditionals.

PROGRAM 5: Start code for exercise 5.

```java
public static int hasDoubled
      (double savings, int interest) {
  double target = 2 * savings;
  int years;
  for (years = 0; ; ) {
    if (target > savings) {
      savings *= interest / 100.0 + 1;
      years++;
    }
    else
      if (target <= savings) {
        break;
      }
  }
  return years;
}
```

Description: Write a program that calculates in how many years your savings have doubled with the given interest (as a percentage).

An example: if your savings are 1000 and the interest is 4%, it will take you 18 years to double your savings (then you'll have more than 2000).

TABLE 6.10: For exercise 4, hints seen and solved by students (n=87).

Type of the most detailed hint seen	Seen by		Solved by		Total solved
Replace unneeded loop by calculation (line 4-6)					
No hint	48	(55%)	44	(92%)	
Top-level hint	23	(26%)	23	(100%)	83 (95%)
→ Code example	16	(18%)	16	(100%)	
Move statement from if + else outside (line 8+12)					
No hint	67	(77%)	62	(91%)	
Top-level hint	6	(7%)	6	(100%)	82 (94%)
→ Detailed hint	10	(12%)	9	(90%)	
→ Code example	4	(5%)	4	(100%)	
Use the compound -= operator (line 11) ↻					
No hint	71	(82%)	69	(97%)	
Top-level hint	8	(9%)	8	(100%)	84 (97%)
→ Detailed hint	6	(7%)	5	(83%)	
→ Code example	2	(2%)	2	(100%)	
Cleanup empty if or else ⋆					
Top-level hint	0	(0%)	–		
→ Detailed hint for empty if	3	(3%)	3	(100%)	
→ Detailed step	1	(1%)	1	(100%)	n.a.
→ Code example	3	(3%)	3	(100%)	
→ Detailed hint for empty else	2	(2%)	2	(100%)	
→ Code example	0	(0%)	–		

The exercise was attempted by only 44 students of which 20 finished it. There were no more hints left for the finished programs. The hints that were seen the most revolved around optimising the return statement for constructs such as `if(condition) return true; else return false;` that can be written as `return condition;`. A total of 7 students saw at least one of the hints for this program; the other hints were only seen by single students. We manually examined three final programs for which there were no more hints left, and found that one contained an unreachable statement (a bug in the system) and a return statement could have been optimised, which would require a quite sophisticated analysis of the control flow.

TABLE 6.11: For exercise 5, hints seen and solved by students (n=54).

Type of the most detailed hint seen	**Seen by**		**Solved by**		**Total solved**
Remove useless check in else-if (line 11) ↻					
No hint	44	(81%)	43	(98%)	52 (96%)
Top-level hint	9	(17%)	8	(89%)	
→ Detailed hint	0	(0%)	–		
→ Code example	1	(2%)	1	(100%)	
Replace for by while-loop					
No hint	40	(74%)	35	(88%)	47 (87%)
Top-level hint	11	(20%)	10	(91%)	
→ Detailed hint	1	(2%)	1	(100%)	
→ Code example	2	(4%)	1	(50%)	
Replace break by loop condition ⋆					
Top-level hint	3	(6%)	3	(100%)	n.a.
→ Code example	1	(2%)	1	(100%)	

6.5.3 Student evaluation (RQ3)

All 133 students filled in a short survey at the end of the tutoring session. First we asked 'what do you think about paying attention to code quality' on a 5-point Likert scale ranging from 'not important at all' to 'very important', to which 24% of the students answered very important, 67% answered important and 9% answered neutral. The next question was 'how difficult did you find the exercises' from 'very easy' to 'very difficult'. Of the 131 students who gave a valid answer, 3% found the exercises very easy, 14% easy, 68% were neutral and 15% found them difficult. The last closed question was 'how useful did you find the hints', from 'not useful at all' to 'very useful'. A total of 13% answered very useful, 64% said useful, 13% were neutral and 3% did not find the hints useful. Ten students answered that they did not use the hints.

The remaining subsections discuss the responses to the questions 'what did you like about the system' and 'what would you want to see different in the system'. These aspects can be broadly categorised under 'learning about code quality', 'feedback and hints', 'user interface', and 'exercises'.

Learning about code quality

We were pleased that multiple students said something along the lines of ‘fun’, ‘challenge’ and ‘good initiative’. Four students commented on the topic of code quality and its importance. One student wrote: ‘Code quality is one of the most important things in a team. It should be stimulated more and the tool helps with that.’ More than 40 students said something they liked about what they took away from working with the system. Some examples of responses are: ‘That you can see that the code looks simpler when you changed something and it is correct’, ‘You learn to look out for code that is unnecessarily complex or long’, and ‘It offers a good challenge in the available time for experienced programmers, and at the same time it is very accessible’.

Several students used the phrase ‘making you think’, such as ‘It lets you think about solutions that you normally wouldn’t think of’ and ‘You really have to think and act yourself. A pleasant way of learning.’ One student had an interesting idea: ‘It would be cool if there really was an app or website for the cleanliness of code (this but then with waaaay more exercises), because it is really good for learning code.’ Three students mentioned as an improvement that it would be useful to better explain what the benefit is of a certain suggestion. Four students gave concrete examples of constructs about which they doubted if they were actually ‘better’, such as `i += 2` instead of `i = i + 2`, two checks combined with && in the condition of a for-loop, or a foreach-loop instead of a for-loop.

Hints and feedback

About 48 students positively mentioned the hints and feedback the system gives. Most comments were about the usefulness and clarity of the hints, such as ‘the hints make you think’. Six students appreciate that hints can be gradually revealed and do not give away the solution straight away: ‘Often, the first was sufficient’ and ’Hints only point in the right direction, so you’ll come up with improvements yourself, unless you really can’t.’ Multiple students also mention that they like that you can track your progress by seeing how many improvements can be made. Two students would have liked to see that right at the beginning of an exercise.

There are also some issues with the feedback that several students pointed out. This seems to be the case for a few specific instances, such as the ‘buggy if collapse’: a common mistake many students made, for which they did not get a clear error message. This particular issue was pointed out by seven students. Other students also mentioned in general that feedback could have been clearer, which was mentioned more often for error messages than for hints. Some students gave suggestions on how to improve the feedback system, such as clearly indicating the location of errors. Students also pointed out some unexpected behaviour from the system, such as 'Errors where there should not be an error.' We had encouraged students to write down unexpected behaviour so we could look into it and fix it at a later stage. We also suspect some students were confused by the messages from the Java compiler. The effectiveness of compiler error messages has been a major research area for a long time, but is still a persistent problem causing confusion and frustration for students [24].

User interface

A total of 28 students mentioned the user interface in a positive way, mentioning its simplicity, usability and speed (7 times), or simply that it worked well. A total of 22 students were not happy about the fact that the current state for an exercise is not saved, so they lose their progress if they go to another exercise or accidentally leave the page. Nine students express their satisfaction with the code editor and its syntax highlighting, code formatting and useful shortcuts. The responses also contain several suggestions for improving the editor, such as a dark mode, auto-completion features and underlining syntax errors similar to what IDEs do.

There were also multiple suggestions about improving the interface in general. Five students requested smoother transition between exercises. Other comments mentioned more than once suggested undo/redo functions, and keep showing the hints even after a subsequent step with an error.

Exercises

Eleven students particularly mention the exercises in a positive way: they were clear and not too big. Nine students would like to see some changes: three students found some exercises unclear and the other students all had their own requests, such as more (variety in) exercises and different levels of difficulty.

The topic of language support was discussed in the improvement comments. Seven students would like to see C# support (or even other languages), the language they were mostly using at that moment. Four students requested support for language features such as functions and ternary operators.

6.6 Discussion

In this section we refer back to the research questions and discuss the findings in Section 6.6.1. We reflect on the implications for teaching code refactoring in Section 6.6.2 and the quality of the system's ruleset in Section 6.6.3, and discuss the threats to validity in Section 6.6.4.

6.6.1 Student refactoring behaviour

How do students solve refactoring exercises? Which steps do they take, and which mistakes do they make? (RQ1) The students in our study, who had already learned the basics of programming, managed quite well solving refactoring exercises. However, because the second exercise showed signs of struggle, fewer students completed the exercises after the second. On average, 92% of the quality issues that were present at the start were dealt with. Students regularly used the Check diagnosis function, which we view positively because a central aspect of refactoring is to maintain functional behaviour. We can see from the proportion of expected diagnoses that students also regularly took small steps. Compiler errors and failed tests are both still pervasive, even though the students started with functionally correct code. We want to decrease the number of failed tests by adding more buggy rules to provide specific feedback on incorrect steps. For the first few exercises, the majority of the students saw at least one hint. In the subsequent exercises, fewer students saw hints.

When do they ask for a hint? For which issues do they need hints? How do they respond to a hint? (RQ2) The majority of the students regularly requested hints. For all issues in the code that were known at the start, on average 34% received a hint (mean 22%). It varied per issue whether a student expanded the hint to get more detailed information. Even though some researchers warn against providing progressive hints ending with the correct answer [239], we did not see negative effects of this. We told students to just ask for the hints when needed, and noticed from the data that they requested hints at various levels. For only a few complex issues the bottom-out hint was also the most requested. The fact that the participants were second-year students might have contributed to this behaviour.

Not surprisingly, students had no real trouble with simple edits, such as using compound operators and removing self assignments. Students had the most trouble with complex control flow (nested if statements and breaking out of a loop) and composed expressions. These are the more algorithmic issues that are usually not flagged by professional static analysis tools, acknowledging the need for educational tools that do provide support for these issues.

For recurring issues we see progress in some cases. More students managed to solve them, or they needed a hint with less detail. Not in all cases we noticed an effect, which could be due to a different context and the student's inability to transfer what they had done before.

What do students think about working with a refactoring tool? (RQ3) All study participants answered questions on how they perceived code quality and working with the tool. We noticed the positive attitude of students towards code quality, which might be expected from students in their second year that have chosen the software engineering profile. They found the level of the exercises to be appropriate for the amount of time given and their skills, and considered the hints to be useful.

Overall, the students gave a positive evaluation of the tutoring system and appreciated working with it. We also noticed this in the classrooms, where almost all students worked on the exercises with focus, even though it was their last lecture week and there were other topics to discuss and a programming assignment to finish around that time.

Making students aware and letting them think about code quality instead of just writing code to meet the requirements of an assignment seems to be a valuable effect of using the tool. However, from the feedback we also learn that discussing the improvements the tool suggests is important. This could be done by either expanding the tool with explanations, but also by discussing these topics in the classroom (see Section 6.6.2).

Students generally value the functions the systems offers: the hints, checks and the progress indicator, although some bugs and unclear (error) messages understandably confused them. We should be aware that these negative experiences could cause resistance to the system.

It stood out that students valued features of our tutoring system that are often seen in IDEs, and they suggested adding more of those features. This particular group of students had been working in the Visual Studio IDE for some time, an environment offering many functions. However, we want the tutoring system to be free from distraction and only offer relevant UI features, so we will carefully consider which requests we will meet.

6.6.2 Teaching code refactoring

We have shown that students were able to solve refactoring exercises in the tool, and from the student comments we can derive it has made them think about a topic they normally do not spend much time on. From the survey we infer that these students, who are a bit more experienced than novices, do think it is an important topic. However, we think a greater effect can be reached by accompanying the tool by a lesson on code quality, and by discussing the suggestions of the tool afterwards. A few students mentioned that they did not agree with certain ‘improvements’. We should discuss that some of them are more stylistic, such as using the compound operator `*=`, but it might not be wise to use it when they do not fully understand how it works. We should emphasise that the system will also introduce them to alternative language constructs that they might not have encountered yet. For example, using a foreach-loop instead of a for-loop (if possible) has benefits and can protect the programmer from accidentally doing something unwanted. We would both like to incorporate this in the tool, as well as make it a part of the programming lessons.

Research on students working with educational code quality tools is scarce. The study by Wiese et al. [288] is the most relevant to our work. This study had a control group receiving only a quality metric (the ABC-metric [83]), and an experimental group receiving feedback from their AutoStyle tool. Both groups performed a pre-test and a post-test, and the authors described some case studies of particular student types. Because their students started out with their own code, and our study focusses more on how students worked in the tutor, we cannot directly compare results. However, both studies support the potential for using these kinds of systems, the call for more in-depth help for students, and more studies evaluating the results.

6.6.3 Quality of the rule set

We inspected some end states manually to explore the *soundness* and *completeness* of our rule set. Although we do not strive for completeness, because it is impossible to foresee every single solution to a programming problem, we want to minimise cases in which our system tells the student there are no hints left, while their solution is imperfect. Although we do not formally prove soundness, we believe the rules are sound because they represent mathematical and logical rules, and adhere to the semantics of programming language constructs. Continually checking a program against a set of test cases also ensures that the behaviour of the program is preserved.

We derive the following improvements from our manual inspections in which we identified some deviating solutions:

- Extending certain existing rules with more normalizations so that they fire for a larger number of cases. This is mostly the case for (nested) control flow.
- Adding rules related to the use of variables. We should take care designing these rules, as clarity is most important and arriving at the fewest number of lines of code is not.
- Implementing more buggy rules to intercept undesirable code edits.

In cases in which our analysis fails to detect a certain issue because it deviates too much from the base case, we might have to resort to high-level metrics, such as counting language constructs or nesting levels. We will further explore this in future work.

6.6.4 Threats to validity

The system contained some bugs at the time of the experiment, which confused some students and might have slowed down their progress. The fact that the students were learning a different programming language at the time of the experiment could have taken up some valuable time to recollect Java syntax for certain language constructs.

The granularity of the log data we analysed is at the snapshot-level, which is the middle level between submission-level and key-stroke-level, as identified by Vihavainen et al. [273]. The snapshots were recorded when students performed a check activity or hint request, implying that we do not have detailed information on all of their steps, and our analysis could have missed relevant edits that were undone before taking an action. Solving an issue is measured by calculating hints for subsequent program states, and determining the differences. Some issues might be solved in unexpected ways for which our system does not generate a hint, which could mean in some cases there was no actual improvement.

Another threat to validity is that fewer students worked on the last few exercises, which are probably the 'better' students. This could cause an overestimation on how well students performed on these exercises, and would explain the fact that fewer students received hints for these exercises.

The novelty of a new tutoring system could also have an effect on the enthusiasm of the students.

6.7 Conclusion and future work

This paper describes the results of an exploratory study of log data from students working on refactoring exercises, and the students' personal experiences thereof. The students worked for 30 minutes on 6 exercises at their own pace, in which they had to improve imperfect example solutions. All participants had at least a basic background in programming, and were mostly able to do the exercises, with regular checks to verify correctness. However, they also struggled with complex control flow. The students regularly requested hints, at various levels of detail. After seeing a hint about a particular issue, most students solved the same issue without a hint when they encountered it again. Overall, students valued the topic of code quality and working with the system, but also proposed valuable enhancements.

These results contribute to focussing the attention of teachers and tool designers, by incorporating the discussion of refactoring rules in education (in particular those related to complex expressions and control flow), paying attention to alternative language constructs, and providing feedback at various levels to meet the needs of individual students.

We will continue to improve and validate the system with students to make them more aware of the quality of code and the importance of refactoring. We intend to extend the ruleset and add more buggy rules. We also want to refine feedback messages and hints with more explanations on why an edit is useful. Support for methods could present new opportunities to improve code. It would be useful to present these considerations to students when they try to implement such a feature, making them think about the implications. Self-reflection questions after doing an edit ('why did you just replace the for by a foreach?') could be an effective addition to the system. Finally, we will work on further enhancing the UI while maintaining a focused environment that does not distract from the system's learning goals.

After incorporating improvements, we intend to conduct a different type of experiment with a control group and experimental group, and a pre- and post-test to determine learning gains. We also want to study the effect of using different types of feedback.

Chapter 7

Epilogue

In this final chapter, we reflect on the work in this thesis, look at some emerging trends in the field of automated feedback, and describe implications for future work. We started this research by exploring the state of the art of automated feedback for programming exercises. We then focussed on the topic of code quality and style, investigating to what extent student code exhibits quality issues. To investigate the role of teachers, we studied their opinions and the concrete feedback they would give on student code. Our findings resulted in the design of a programming tutoring system that helps students to refactor code step by step. Finally, we discussed how students perceived this tutoring system, and how they approached the exercises.

7.1 Conclusions

We now revisit and answer the central research question of this thesis by answering the corresponding subquestions:

How can automated feedback support students learning code refactoring?

RQ1 What are the characteristics of existing tools that give automated feedback on programming exercises?

We found that the most common feedback category was, unsurprisingly, about mistakes. This type of feedback is often in the form of showing failed tests, or pointing to incorrect constructs in the code. Feedback that helps students progress or improve is less common, in particular for exercises that require writing a solution from scratch. Many different techniques are being used for generating feedback in tools, often tied to a particular kind of feedback. The role of teachers in authoring exercises and feedback messages is limited. How rigorously a tool and its feedback are validated varies greatly, and leaves room for improvement.

RQ2 Which code quality issues occur in student code, and are these issues being fixed?

This study provided us with insight into quality issues in student code. We found various instances of all kinds of issues flagged by a professional static analysis tool, which often remained unfixed after emerging for the first time. We did not see a difference with students who had some type of code analyser installed. Although some related studies, published after our study, did find effects of using these tools, we argue that it very much depends on the context. In our large data set, students worked in many different settings, both controlled and non-controlled, so the need to fix issues was probably less than in settings in which students were being graded on quality issues.

RQ3 How would teachers help students to improve their code?

Teachers find code quality an important topic, but acknowledge its minor role in summative assessment. Most teachers could name improvements for the three imperfect student programs we showed them. We noticed, however, that professional analysis tools generally point to different issues than teachers do. Between teachers there were also various differences, although they generally agreed on algorithmic issues.

RQ4 How do we design a tutoring system giving automated hints and feedback to learn code refactoring?

We have designed a tutoring system that is based on rules derived from experts, the literature, and logic/maths. The rules capture rewrite steps for functionally correct programs, preserving the programs' semantic meaning. These steps are hierarchically combined into larger rewrite strategies, describing the ordering and prioritisation of substeps. Automated hints are derived by calculating next steps for a particular program state, using layered hint labels. Correctness feedback is calculated by running test cases, calling a compiler, and checking whether a student has performed an expected step, or a buggy step.

RQ5 What is the behaviour of students working in a such a tutoring system?

A large group of students worked on refactoring exercises in the tutoring system, in which they had to improve correct, but imperfect code. The students already had some background in programming from two earlier courses, and were able to identify most issues in the programs and rewrite them into improved programs. They requested hints at various levels, and regularly checked the correctness, as required when refactoring. Students struggled most with complex control flow, an aspect professional static analysis tools do not always deal with. We also noticed the students' positive attitude towards the topic and the system, although we observed some confusion about unclear feedback messages.

7.2 Recent trends

It has been four years since the most recent paper from our systematic literature review was published. Pasting our search query from the literature review in the ACM library search box returns hundreds of papers published after 2015, showing that the field has continued to develop. We have noticed in particular that the trend of using data-driven feedback has persevered (e.g. [173], [183], [218]). There also seems to be a renewed interest in creating ITSs for programming. The effect of (different types of) hints has also been studied more extensively. For example, Marwan et al. [183] have evaluated data-driven next-step hints in a programming tutor for a block-based language, studying the effect of textual explanations and self-explanation prompts. Price et al. [218] have developed a method for evaluating the quality of data-driven next-step hints, and have applied this method to different hint generation algorithms. Data-driven techniques are also being used for other types of feedback, such as program repair hints and example feedback. By contrast, Yang et al. [118] use refactoring rules to transform a program with the goal of finding the closest match to a *single* correct solution, before identifying fixes for incorrect blocks. We have also observed an increased focus on feedback generation at scale for large, MOOC-style courses about programming (e.g. [254], [278]). In some new hint generation techniques, teachers have an authoring role [182].

In their 2018 literature review on introductory programming [173], Luxton-Reilly et al. identify the following trends in tools for programming: the use of data-driven techniques and intelligent tutoring techniques to improve feedback, and 'addressing skills other than code-writing'. Our work relates to two of those trends: the use of intelligent tutoring techniques such as next step hints, and addressing a different type of skill, namely refactoring functionally correct code.

Our tutoring system is a rule-based system, mostly based on rules of experts, which is rather different from the current data-driven trend. We argue that when it concerns a topic such as code quality, for which there are no clear rights and wrongs, the role of the expert (teacher) is vital. The expert's role may take the form of authoring hints for clusters of similar programs, as seen in the data-driven AutoStyle tutor [52]. However, our research has shown that using a set of expert-devised rules is appropriate and useful for introducing students to the topic of refactoring. We do acknowledge the need to put teachers in charge more, through authoring feedback messages and explanations, and disabling and ordering certain rules. We consider this to be future work.

The introductory programming review mentioned earlier also found that papers on tools to support learning and teaching is a major category. Their concern, which is shared by others (e.g. [81], [211]), is related to the evaluation and dissemination of these tools, which is not always of high quality. The main focus of this thesis has been on the motivation and design of the tutoring system. We have also studied log data from the system to get more insight into how students approach refactoring and how they use feedback and hints. We have yet to study the learning effect of using the system, which is future work and is not a part of this thesis. Regarding dissemination, we have made the system available online for others to use, and we welcome studies with students from other countries and institutes. We have also planned to publish the code soon.

7.3 Future work and final thoughts

The work discussed in this thesis is merely a starting point. In the near future we will initiate the second design cycle [287], improving the tutoring system by incorporating findings from our studies. We will validate the system with a group of students who are less experienced than the group from our study, with an increased focus on studying learning effects. In this thesis we have also argued for the need of accompanying lessons to explain and discuss the rewrite rules from our system, as well as explaining the importance of code refactoring in general. The role of the student, and his or her rationale for improving code, is also a topic to be explored.

At a later stage, we would like to expand to higher-level refactorings. Automated feedback at the class level is still rare, and will probably test the limits of the techniques we are currently using.

The introduction of this thesis started with the debate whether programming is really that hard. This thesis does not aim to answer that complex question. However, we contribute by providing more insight into quality aspects of student programs, the helpfulness of the feedback students get from tools, the perceptions of teachers, and by delivering a tutoring system that helps students refactor code. It is unlikely that programming will ever be easy, but if we keep on studying how we can best support students, we should have more realistic goals for students to reach after their first courses.

Samenvatting

Een bekend probleem voor docenten in het programmeeronderwijs: een student schrijft een functioneel correcte oplossing, maar de code is inefficiënt, onnodig complex of onleesbaar. Helaas is het binnen het onderwijs niet altijd mogelijk om voldoende tijd en aandacht te besteden aan de kwaliteit van code. Uit onderzoek blijkt dat zelfs de code van professionele programmeurs veel ‘code smells’ bevat, met als gevolg dat kwaliteitskenmerken zoals onderhoudbaarheid in gevaar komen. De vele professionele tools die helpen bij het opsporen en verbeteren (‘refactoren’) van problematische code zijn meestal niet geschikt voor beginners. Dit proefschrift gaat over codekwaliteit in de context van studenten die leren programmeren en de kleine programma’s die zij schrijven, onderzoekt hoe studenten en docenten met codekwaliteit omgaan, en hoe tools en softwaretechnologie kunnen worden ingezet om ze hierbij te ondersteunen. Om studenten vroeg te leren hoe ze hun (functioneel correcte) code kunnen verbeteren, hebben we een tutorsysteem ontwikkeld die studenten leert hoe ze code kunnen refactoren. De tutor geeft automatische feedback op de stappen die de studenten nemen, en geeft hints als ze vastlopen.

De volgende vraag staat centraal in dit proefschrift:

Hoe kan automatische feedback studenten helpen bij het refactoren van code?

In dit proefschrift zoeken we naar een antwoord op deze vraag, waarbij elk hoofdstuk één van de vijf subvragen beantwoordt.

RQ1 Wat zijn de kenmerken van bestaande tools die automatische feedback geven op programmeeropgaves?

We hebben een systematisch literatuuronderzoek gedaan naar het automatisch genereren van feedback op programmeeropgaves. Het onderzoek heeft een brede scope: we bekijken het vroegste werk uit de jaren 60 tot en met artikelen uit 2015. Van 101 tools is onderzocht welk type feedback ze genereren,

welke technieken ze daarvoor gebruiken, in hoeverre de feedback aanpasbaar is, en op welke manier de tools zijn geëvalueerd. Om de feedbacktypes te categoriseren, hebben we een bestaande classificatie van Narciss (2008) gebruikt, die we hebben uitgebreid voor het programmeerdomein.

Uit het onderzoek blijkt dat feedback zich meestal richt op het identificeren van fouten in code, bijvoorbeeld door het tonen van mislukte test cases of het aanwijzen van incorrecte statements. Feedback is veel minder gericht op het oplossen van fouten, het helpen van studenten op weg naar een oplossing voor een programmeerprobleem, en het verbeteren van code die al correct is. We zien een toenemende diversiteit in de technieken die worden gebruikt om feedback te genereren, zoals data-gedreven technieken gebaseerd op grote hoeveelheden historische studentdata. Deze technieken bieden nieuwe mogelijkheden, maar zorgen vaak ook voor nieuwe uitdagingen. Verder zijn tools en de feedback die ze genereren vaak lastig aan te passen door docenten. De mate waarin tools zijn geëvalueerd op hun gebruik en effectiviteit wisselt sterk, en blijft een belangrijk aandachtspunt.

RQ2 Welke kwaliteitsissues komen voor in studentcode, en lossen studenten deze issues op?

Er zijn vele studies gedaan naar fouten die studenten maken in hun code. Daarentegen is er veel minder aandacht geweest voor de kwaliteit van de code. Om uit te zoeken in welke mate studentcode kwaliteitsissues bevat, hebben we een analyse gedaan van meer dan 2,5 miljoen codefragmenten in Java met behulp van de professionele statische analysetool PMD. We hebben een subset met checks (regels) van PMD geselecteerd op basis van een bestaande rubric voor het beoordelen van de kwaliteit van studentcode. Deze regels hebben te maken kwaliteitsissues op het gebied van de flow van code, de keuze voor programmeerconstructies, duidelijkheid van expressies, decompositie en modularisatie. We hebben diverse instanties van deze issues gevonden, die nauwelijks werden opgelost in latere submissies van hetzelfde codefragment. Dit laatste gold met name voor issues met betrekking tot modularisatie. We zagen geen effect van geïnstalleerde kwaliteittools op het aantal gevonden issues. In gerelateerd onderzoek dat na dit onderzoek verscheen was dit effect soms wel zichtbaar. We stellen dat dit effect sterk afhankelijk is van de context waarin een dergelijke tool wordt ingezet; in onze dataset werkten studenten in diverse settings (zowel gecontroleerd als niet-gecontroleerd), waardoor het effect waarschijnlijk minder was dan in settings waarin studenten beoordeeld werden op kwaliteitsissues.

RQ3 Hoe helpen docenten studenten hun code te verbeteren?

Docenten spelen een belangrijke rol bij het onder de aandacht brengen van codekwaliteit. Idealiter zou elke student persoonlijke feedback moeten kunnen krijgen van een docent op de kwaliteit van zijn of haar code. Helaas is dit lastig vanwege grote studentaantallen en beperkte tijd van docenten. We hebben door middel van een enquête onderzocht hoe docenten naar het onderwerp codekwaliteit kijken. Dertig docenten van diverse instituten hebben geparticipeerd, door hun mening te geven over codekwaliteit, en feedback te geven op drie imperfecte studentoplossingen voor een programmeerprobleem. Uit de resultaten bleek dat docenten codekwaliteit een belangrijk onderwerp vinden, maar dat het in de praktijk een beperkte rol speelt bij het summatief beoordelen van code. Bij de codefragmenten van studenten gaven de meeste docenten vergelijkbare hints over het verminderen van de algoritmische complexiteit en het opruimen van 'clutter'. Over andere aspecten gaven ze wisselende combinaties van hints. We zien een grote diversiteit in hoe ze de programma's zouden herschrijven naar een verbeterde versie. We zien veel verschillen bij het vergelijken van de hints die docenten geven met de uitvoer van professionele statische analysetools, met name rondom de algoritmische complexiteit waar deze tools voor dit soort kleine programma's geen feedback over geven.

RQ4 Hoe ontwerpen we een tutorsysteem dat automatische hints en feedback geeft bij het leren van code refactoring?

Op basis van de bevindingen uit de voorgaande onderzoeken hebben we een tutorsysteem ontworpen. Het systeem biedt refactoropgaves aan, waarin de student kleine programma's, die al functioneel correct zijn, moeten herschrijven naar een versie die beter, leesbaarder en/of efficiënter is. De student kan bij elke stap vragen om feedback en hints. Hints worden gegeven op verschillende niveaus: van een globale aanwijzing tot een concreet codevoorbeeld. Het is een regelgebaseerd systeem, waarbij de regels zijn gebaseerd op de input van docenten uit de voorgaande studie, een voor studenten geschikte subset van regels vanuit professionele tools, en regels uit de wiskunde/logica en andere bronnen uit de literatuur. We hebben het systeem geëvalueerd door het te vergelijken met professionele tools, en door te laten zien hoe het systeem overeenkomt met hoe docenten studenten zouden helpen refactoren.

RQ5 Hoe gedragen studenten zich in een tutorsysteem voor refactoring?

In de herfst van 2019 heeft een groep van 133 studenten gewerkt met de refactoring tutor. De tweedejaars studenten, die allen al een basis in programmeren hadden, werkten aan zes refactoropgaves, waarbij hun activiteiten werden gelogd in een database. Na afloop van het werken aan de opgaves werden ze gevraagd naar hun bevindingen via een enquête.

De studenten kwamen over het algemeen vrij goed uit de opgaves, hoewel ze zeker niet allemaal aan alle opgaves toe kwamen. In gemiddeld 92% van de gevallen waren ze in staat om de issues op te sporen en te refactoren, al dan niet met behulp van de hints. Uit de data maken we op dat de studenten hints op verschillende niveaus gebruiken, en geregeld kleine stappen nemen die ze verifiëren met de diagnose-functie die de functionele correctheid checkt, zoals gewenst bij het refactoren. De opgaves bevatten issues die, soms in aangepaste vorm, terugkomen in latere opgaves; voor bepaalde issues hadden de studenten dan minder hulp nodig ze op te lossen, hoewel dit effect niet altijd zichtbaar was in andere gevallen. De studenten hadden de meeste moeite met het versimpelen van complexe control flow. Aan dit aspect besteden professionele statische analysetools beperkt aandacht, wat in het bijzonder geldt voor kleine studentprogramma's.

De studenten geven aan dat ze over het algemeen waren te spreken over het werken met het systeem. Ze vinden het onderwerp van codekwaliteit belangrijk. Ze waarderen de hints en feedback van het systeem, maar noemen ook enkele kritische punten met betrekking tot onduidelijke feedbackberichten.

Het werk van dit proefschrift is slechts een startpunt. In diverse iteraties zal de tutor verbeterd worden, en gevalideerd worden met groepen studenten, waarbij we met name naar leereffecten willen kijken. Ook willen we kijken naar hoe minder ervaren studenten met refactoropgaves omgaan. Daarnaast benadrukken we in dit proefschrift dat het erg belangrijk is om aanvullende instructie te geven over het nut van refactoren en de betekenis van de diverse regels die in het systeem zitten. Op de lange termijn zal het waardevol zijn te kijken naar refactorings op een hoger niveau, zoals dat van klassen en interfaces. Daarnaast is het bestuderen van de motivatie van de student voor refactoren een interessant onderwerp.

Dit proefschrift draagt bij aan het verder verbeteren van het programmeeronderwijs, door het geven van inzicht in de kwaliteitsaspecten van programma's geschreven door studenten, de mate waarin feedback van tools bijdraagt aan die kwaliteit, de visie van docenten daarop, en door het aanbieden van een tutorsysteem waarin studenten worden geholpen code te refactoren.

Dankwoord

Toen ik 2007 besloot mijn Master te gaan halen in deeltijd, naast mijn werk als docent aan een hogeschool, had ik nooit gedacht dat ik vele jaren later een proefschrift zou gaan afronden. Ik beleefde zoveel plezier aan het doen van onderzoek voor mijn masterscriptie, dat ik helemaal niet wilde stoppen na dit al best wel lange traject. Ik ben mijn promotor Johan en co-promotor Bastiaan erg dankbaar dat ze me op het idee brachten een promotiebeurs voor leraren aan te vragen. Het was niet altijd makkelijk, zeker met een drukke baan ernaast, maar ik heb er nooit spijt van gehad dat ik het ben gaan doen, en het heeft me veel gebracht.

Johan, ik wist dat een goede promotor een belangrijke factor was voor een succesvol promotietraject. Daar heb ik bij jou nooit aan getwijfeld. Je maakte altijd tijd voor het (tijdig) lezen van stukken en bespreken van resultaten, en ik waardeer je zorgvuldigheid daarbij. Als ik naar mijn oorspronkelijke voorstel kijk, komt geen enkele onderzoeksvraag daaruit letterlijk terug in mijn proefschrift. Je hebt me altijd de vrijheid gegeven om een iets andere richting op te gaan, of voor een andere focus te kiezen. Natuurlijk gaf je daarbij altijd bijsturing om uiteindelijk bij een samenhangend proefschrift uit te komen. Van de vele clichés die er bestaan over lastige eigenschappen die promotoren kunnen hebben, herken ik er bij jou geen.

Bastiaan, bij het afronden van mijn masterscriptie had ik geen dankwoord geschreven, waarover je in 2014 mailde '…je hebt nog vier jaar de tijd om te bedenken wat je daarin zou willen zetten'. Het is iets langer geworden, maar gelukkig kan ik dat nu goedmaken! Toen ik je leerde kennen als mijn docent bij Software Evolution, wist ik al snel dat het een goed idee zou kunnen zijn om bij jou af te studeren, en later dus ook te promoveren. Ik waardeer je betrokkenheid, nauwkeurigheid, en betrouwbaarheid (met de Oxford komma, natuurlijk). Ik heb veel van je geleerd, zowel op het gebied van onderzoek doen, als het programmeren in Haskell. Ik heb het ook erg gezellig gehad op onze kamer, tijdens de vele overleggen, maar ook bij de wandelingen op de Uithof.

Verder wil ik jullie allebei bedanken voor de steun die ik van jullie heb gekregen in de moeilijke maanden die ik persoonlijk had in het halfjaar voor mijn promotie, ik heb die ontzettend gewaardeerd.

Ondanks dat ik buitenpromovenda was, heb ik toch een fijne werkomgeving gehad met regelmatig contact met andere onderzoekers. Ik ben er dankbaar voor dat ik een dag in de week een werkplek had op de Universiteit Utrecht. De jaarlijkse hackathons met Johan, Bastiaan, Alex, Josje, en wisselende deelnemers waren altijd erg nuttig en gezellig. Daarnaast heb ik vele interessante papers mogen bespreken met de reading group van Software Technology for Learning and Teaching van de UU, waar ik uiteindelijk ook mijn nieuwe werkomgeving heb gevonden.

Ik wil mijn vele collega's van HBO-ICT Windesheim bedanken voor hun interesse en steun. In het bijzonder wil mijn collega's Ernst, Eltjo, Thomas, Arjen, Yvette, Puja, Martijn, Freek en Henk van Team Software Engineering noemen: bijna iedereen van jullie heeft zelfs ook echt een inhoudelijke bijdrage kunnen leveren aan mijn onderzoek! Robbert, bedankt voor de mooie omslag, je steun tijdens het traject, en het vertrouwen dat ik het kon. Gido, bedankt voor je interesse en het delen van je ervaringen. Christina, Warna en Marjolein, bedankt voor de nodige afleiding de afgelopen jaren.

Mijn familie is altijd een belangrijke basis geweest in mijn leven. Ik wil mijn vader bedanken die alle Engelse teksten wel een keer (of meerdere keren) heeft voorzien van feedback, en mijn moeder voor het controleren van de Nederlandse teksten. Pap, mam, Johannes, Sijtze, Jolanda en Vera, bedankt dat jullie er altijd voor me zijn en altijd het vertrouwen hebben gehad dat ik het kon. Jullie zijn de beste!

Curriculum Vitae

Hieke Keuning, geboren te Hardenberg op 14 augustus 1981

1993 – 1999
VWO, Vechtdal College Hardenberg

2000 – 2004
Bachelor Informatica, Hanzehogeschool Groningen

2004 – 2005
Softwareontwikkelaar, App Software

2004 – 2020
Docent Software Engineering, HBO-ICT, Hogeschool Windesheim

2007 – 2014
Master Computer Science (deeltijd), Open Universiteit

2015 – 2020
Promovenda (deeltijd), Open Universiteit

2020 – heden
Universitair docent Informatica, Universiteit Utrecht

Appendix A

Code Refactoring Questionnaire

Introduction

Why?
We are interested in how lecturers teach code refactoring: improve the quality of code that is functionally correct, but not the most elegant/short/efficient implementation.
Who?
Lecturers that have been teaching programming and other computer science-related courses for at least 2 years.
How?
First we ask some general questions. Next, the questionnaire presents three code fragments and asks how you as a lecturer would help a student to improve the code. The questionnaire should take you between 15 and 20 minutes to complete.

Confidentiality

To help protect your confidentiality, you do not have to reveal information that will personally identify you. The results of this study will be used for research purposes only.

I give permission to use my responses for research purposes *[1]

◯ Yes

◯ No

[new section]

General information

What is your current occupation/job title? * [short answer]

In which country do you teach computer science-related courses? * [short answer]

At which institute(s) do you teach computer science-related courses? You can leave this field empty if you do not want to answer this question. [short answer]

How many years of experience in teaching computer science-related courses do you have? * [number > 0]

What courses do you teach? *

☐ First year courses

☐ Second year courses

☐ Third year+ courses

☐ Other: [short answer]

[new section]

Role of code quality

Code quality deals with the directly observable properties of source code, such as algorithmic aspects and structure. Some examples of code quality issues are:

[1]Fields marked with an asterisk (*) are required.

- Duplicated code.
- An expression that could be shortened.
- Putting the same code in both the true-part and the false-part of an if-statement instead of outside the if-statement.

Although layout and commenting are certainly indicators of code quality, these aspects are beyond the scope of our study.

Code refactoring is editing code step by step while preserving its functionality.

Does code quality appear in the learning goals of your first- and second-year programming courses? *

◯ Yes

◯ No

Do you pay attention to code quality while teaching programming to first- and second-year students? *

◯ Yes, it is a major aspect

◯ Yes, but it has a minor role

◯ No

Do you explicitly assess/grade code quality aspects in programming assignments? *

◯ Yes, it is a major aspect

◯ Yes, but it has a minor role

◯ No

If you advise or prescribe tools that deal with code quality/refactoring to your students, which ones are they? [short answer]

[new section]

Exercises

The following programming exercises are targeted at novice programmers in higher education. We assume that the code is written on a computer with the use of a compiler, and not on paper.

[new section]

Exercise 1

Given the following programming exercise:
'Implement the sumValues method that adds up all numbers from the array parameter, or only the positive numbers if the positivesOnly boolean parameter is set to true.'

```
int sumValues(int [] values, boolean positivesOnly) {
    // sumValues(new int [] {1, -2, 3, -4, 5}, false)
    //   should return 3
    // sumValues(new int [] {1, -2, 3, -4, 5}, true)
    //   should return 9
}
```

The listing below shows the method body of a common, functionally correct student solution.

```
int sum = 0;
for (int i = 0;i < values.length;i++) {
    if (positivesOnly == true) {
        if (values[i] >= 0) {
            sum += values[i];
        }
    }
    else {
        sum += values[i];
    }
}
return sum;
```

How would you assess this solution in a formative situation (e.g. feedback during a lecture or lab)? *

○ Acceptable, does not need to be improved

○ Acceptable, but could be improved

◯ Unacceptable, should be improved

Describe all hint(s) you would give to a student to improve this program. Prioritise the hints by numbering the hints and ordering them from important to less important. [long answer]

How would you want the student to edit (refactor) this program step by step? Type the code after each step by copying the code and applying edits. Leave the remaining fields empty after your final step.

Type the code after the first step. The original code has already been copied below and can be edited. [long answer]

Type the code after the second step. [long answer]

Type the code after the third step. [long answer]

Type the code after the fourth step. [long answer]

Type the code after the fifth step. [long answer]

Type the code after the sixth, seventh, eight, .., final step. Put the number of each edit step before the code. [long answer]

[new section]

Exercise 2 - Solution 1

Given another programming exercise for novice programmers:
'Write the code for the method unevenSum. This method should return the sum of the numbers at an uneven index in the array that is passed as a parameter, until the number -1 is seen at an uneven index.'

```
public int unevenSum(int[] array) {
    // unevenSum(new int [] {44,12,20,1,-1,3,5,-1,99,4})
    //   should return 16
}
```

The listing below shows the body of a solution, based on actual student solutions. This solution contains a functional error regarding the stop condition.

You may ignore this error when answering the questions.

```
int total = 0;
boolean stop = false;

for (int i = 1; i < array.length; i = i + 2) {
    if (stop == false) {
        if (array[i] >= 0) {
            total += array[i];
        } else if (array[i] < 0) {
            stop = true;
        }
    }
    else {
        total = total;
    }
}
return total;
```

[repeat exercise questions]

Exercise 2 - Solution 2

Given the programming exercise from the previous question:
'Write the code for the method unevenSum. This method should return the sum of the numbers at an uneven index in the array that is passed as a parameter, until the number -1 is seen at an uneven index.'

The listing below shows the body of an actual correct student solution (with some variable names translated into English):

```java
int answer = 0;
int index = 0;
boolean value = true;

for(int number: array) {
    if(index % 2 == 0) {
        index++;
    } else {
        if(number == -1) {
            value = false;
        }
        if(value) {
            answer = answer + number;
        }
        index++;
    }
}
return answer;
```

[repeat exercise questions]

[new section]

Completion

After completing these final questions, click below to submit your response.

Do you have any further remarks? [long answer]

If you would like to receive an update on the results of the study, please provide an email address. [short answer]

Appendix B

Stepwise improvement sequences with hints

Minor errors by teachers are corrected

Program 1

Example 1 (participant #163020)

```
int sum = 0;
for (int i = 0; i < values.length; i++){
  if (positivesOnly == true) {
    if (values[i] >= 0) {
      sum += values[i];
    }
  } else {
    sum += values[i];
  }
}
return sum;
```

Hint: could '== true', at the end of a boolean expression, be removed always?
→

```
int sum = 0;
for (int i = 0; i < values.length; i++){
  if (positivesOnly) {
    if (values[i] >= 0) {
      sum += values[i];
    }
  } else {
    sum += values[i];
  }
}
return sum;
```

Hint: Would iterating over values be more readable? (f.i. ‘for (int i : values) {...’)

→

```
int sum = 0;
for (int i : values){
  if (positivesOnly) {
    if (i >= 0) {
      sum += i;
    }
  } else {
   sum += i;
  }
}
return sum;
```

Hint: Does it make sense to add 0 to a number?

→

```
int sum = 0;
for (int i : values) {
  if (positivesOnly) {
    if (i > 0) {
      sum += i;
    }
  } else {
    sum += i;
  }
}
return sum;
```

Hint: We have duplication, in ‘sum += values[i]’. How could we eliminate it, using a binary operator connecting the 2 boolean expressions? Would this make your code more readable or less? Would this make your code more maintainable or less?

→

```
int sum = 0;
for (int i : values){
  if (i > 0 || !positivesOnly) {
    sum += i;
  }
}
return sum;
```

Done.

Program 2a

Example 1 (participant #162208)

```
int total = 0;
boolean stop = false;

for (int i = 1; i < array.length; i = i + 2) {
  if (stop == false) {
    if (array[i] >= 0) {
      total += array[i];
    } else if (array[i] < 0) {
      stop = true;
    }
  }
  else {
    total = total;
  }
}
return total;
```

Hint: Remove lines that does not change the current state (total = total).

→

```
int total = 0;
boolean stop = false;

for (int i = 1; i < array.length; i = i + 2) {
  if (stop == false) {
    if (array[i] >= 0) {
      total += array[i];
    } else if (array[i] < 0) {
      stop = true;
    }
  }
}
return total;
```

Hint: Use break or return instead of the boolean stop.

→

```
int total = 0;

for (int i = 1; i < array.length; i = i + 2) {
  if (array[i] >= 0) {
    total += array[i];
  } else if (array[i] < 0) {
    break;
  }
}
return total;
```

Hint: Replace the else if with else, since it always happens.

→

```
int total = 0;

for (int i = 1; i < array.length; i = i + 2) {
  if (array[i] >= 0) {
    total += array[i];
  } else {
    break;
  }
}
return total;
```

Hint: Reduce the number of nested statements.

→

```
int total = 0;

for (int i = 1; i < array.length && array[i] >= 0; i=i+2) {
  total += array[i];
}
return total;
```

Hint: [introduced a method for isPositiveNumber]

→

```
int total = 0;

for (int i=1; i < array.length && isPositiveNumber(array[i])
     ;i = i + 2) {
  total += array[i];
}
return total;
```

Done.

Example 2 (id #105352)

```
 1 int total = 0;
 2 boolean stop = false;
 3
 4 for (int i = 1; i < array.length; i = i + 2) {
 5   if (stop == false) {
 6     if (array[i] >= 0) {
 7       total += array[i];
 8     } else if (array[i] < 0) {
 9       stop = true;
10     }
11   }
12   else {
13     total = total;
14   }
15 }
16 return total;
```

Hint: Line 13: does nothing.

→

```
 1 int total = 0;
 2 boolean stop = false;
 3
 4 for (int i = 1; i < array.length; i = i + 2) {
 5   if (stop == false) {
 6     if (array[i] >= 0) {
 7       total += array[i];
 8     } else if (array[i] < 0) {
 9       stop = true;
10     }
11   }
12 }
13 return total;
```

Hint: Never use (expression == false), use (!expression) instead.

→

```
 1 int total = 0;
 2 boolean stop = false;
 3
 4 for (int i = 1; i < array.length; i = i + 2) {
 5   if (!stop) {
```

```
 6     if (array[i] >= 0) {
 7       total += array[i];
 8     } else if (array[i] < 0) {
 9       stop = true;
10     }
11   }
12 }
13 return total;
```

Hint: Line 8: is this boolean expression useful?

→

```
 1 int total = 0;
 2 boolean stop = false;
 3
 4 for (int i = 1; i < array.length; i = i + 2) {
 5   if (!stop) {
 6     if (array[i] >= 0) {
 7       total += array[i];
 8     } else {
 9       stop = true;
10     }
11   }
12 }
13 return total;
```

Hint: You can add the stop-criterion to the for-statement.

→

```
1 int total = 0;
2 boolean stop = false;
3
4 for (int i = 1; i < array.length && array[i] >= 0; i=i+2) {
5   total += array[i];
6 }
7 return total;
```

Done.

Program 2b

Example 1 (participant #111046)

```
1 int answer = 0;
2 int index = 0;
```

```
boolean value = true;

for(int number: array) {
  if(index % 2 == 0) {
    index++;
  } else {
    if(number == -1) {
      value = false;
    }
    if(value) {
      answer = answer + number;
    }
    index++;
  }
}
return answer;
```

Hint: If you are keeping an index for the even/uneven numbers, wouldn't it be easier to use a regular FOR loop instead of a FOREACH?

→

```
int answer = 0;
boolean value = true;

for(int index = 1; index < array.length; index += 2) {
  if(array[index]== -1) {
    value = false;
  }
  if(value) {
    answer = answer + array[index];
  }
}
return answer;
```

Hint: If we encounter a -1, can't we just return the answer?

→

```
int answer = 0;
boolean value = true;

for(int index = 1; index < array.length; index += 2) {
  if(array[index]== -1) {
    return answer;
  }
  if(value) {
    answer = answer + array[index];
  }
}
return answer;
```

Hint: If the number is not -1, can't we just assume we can add the number to the sum?

→

```
int answer = 0;

for(int index = 1; index < array.length; index += 2) {
  if(array[index]== -1) {
    return answer;
  } else {
    answer = answer + array[index];
  }
}
return answer;
```

Hint: Can we rewrite x = x + y to something shorter?

→

```
int answer = 0;

for(int index = 1; index < array.length; index += 2) {
  if(array[index]== -1) {
    return answer;
  } else {
    answer += array[index];
  }
}
return answer;
```

Done.

Example 2 (participant #144022)

```
int answer = 0;
int index = 0;
boolean value = true;

for(int number: array) {
  if(index % 2 == 0) {
    index++;
  } else {
    if(number == -1) {
      value = false;
    }
    if(value) {
      answer = answer + number;
    }
    index++;
  }
}
return answer;
```

Hint: Factor out index++ and negate condition.

→

```
int answer = 0;
int index = 0;
boolean value = true;

for(int number: array) {
  if(index % 2 == 1) {
    if(number == -1) {
      value = false;
    }
    if(value) {
      answer = answer + number;
    }
  }
  index++;
}
return answer;
```

Hint: Use a “classical” for-loop.

→

```
int answer = 0;
boolean value = true;

for(int index = 0; index < array.length; index++) {
  int number = array[index];
  if(index % 2 == 1) {
    if(number == -1) {
      value = false;
    }
    if(value) {
      answer = answer + number;
    }
  }
}
```

```
return answer;
```

Hint: Redundant variable value (poorly chosen name btw).

→

```
int answer = 0;

for(int index = 0; index < array.length; index++) {
  int number = array[index];
  if(index % 2 == 1) {
    if(number == -1) {
      return answer;
    }
    answer = answer + number;
  }
}
return answer;
```

Hint: Cosmetics.

→

```
int answer = 0;

for (int i = 0; i < array.length; i++) {
  int number = array[i];
  if (i % 2 == 1) {
    if (number == -1) {
      break;
    }
    answer += number;
  }
}
return answer;
```

Done.

Bibliography

[1] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and interpretation of computer programs.* MIT Press, 1985.

[2] S. Abid, H. Abdul Basit, and N. Arshad, "Reflections on Teaching Refactoring: A Tale of Two Projects", in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2015, pp. 225–230. DOI: 10.1145/2729094.2742617.

[3] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, *Agile software development methods: Review and analysis*, 478. VTT Technical Research Centre of Finland, 2002, ISBN: 951-38-6009-4.

[4] A. Adam and J.-P. Laurent, "LAURA, a system to debug student programs", *Artificial Intelligence*, vol. 15, no. 1-2, pp. 75–122, 1980. DOI: 10.1016/0004-3702(80)90023-5.

[5] E. Aivaloglou and F. Hermans, "How Kids Code and How We Know: An Exploratory Study on the Scratch Repository", in *ICER Conference on International Computing Education Research*, 2016, pp. 53–61. DOI: 10.1145/2960310.2960325.

[6] K. Ala-Mutka, "A survey of automated assessment approaches for programming assignments", *Computer Science Education*, vol. 15, no. 2, pp. 83–102, 2005. DOI: 10.1080/08993400500150747.

[7] K. Ala-Mutka and H. M. Järvinen, "Assessment process for programming assignments", in *IEEE Conference on Advanced Learning Technologies*, 2004, pp. 181–185. DOI: 10.1109/ICALT.2004.1357399.

[8] K. Ala-Mutka, T. Uimonen, and H.-M. Jarvinen, "Supporting students in C++ programming courses with automatic program style assessment", *Journal of Information Technology Education: Research*, vol. 3, no. 1, pp. 245–262, 2004. DOI: 10.28945/300.

[9] V. Aleven, B. Mclaren, J. Sewall, and K. Koedinger, “A New Paradigm for Intelligent Tutoring Systems: Example-Tracing Tutors”, *International Journal of Artificial Intelligence in Education*, vol. 19, pp. 105–154, 2009.

[10] D. Allemang, “Using functional models in automatic debugging”, *IEEE Expert*, vol. 6, no. 6, pp. 13–18, 1991. DOI: `10.1016/S0399-8320(09)73148-3`.

[11] M. Alshayeb, “Empirical investigation of refactoring effect on software quality”, *Information and software technology*, vol. 51, no. 9, pp. 1319–1326, 2009. DOI: `10.1016/j.infsof.2009.04.002`.

[12] A. Altadmri and N. C. C. Brown, “37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data”, in *SIGCSE Technical Symposium on Computer Science Education*, 2015, pp. 522–527. DOI: `10.1145/2676723.2677258`.

[13] J. R. Anderson, *The architecture of cognition.* 1983.

[14] J. R. Anderson and E. Skwarecki, “The automated tutoring of introductory computer programming”, *Communications of the ACM*, vol. 29, no. 9, pp. 842–849, 1986. DOI: `10.1145/6592.6593`.

[15] P. Antonucci, “AutoTeach: Incremental Hints For Programming Exercises”, Master’s thesis, ETH Zurich, 2014.

[16] P. Antonucci, C. Estler, Đ. Nikolić, M. Piccioni, and B. Meyer, “An Incremental Hint System For Automated Programming Assignments”, in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2015, pp. 320–325. DOI: `10.1145/2729094.2742607`.

[17] E. Araujo, D. Serey, and J. Figueiredo, “Qualitative aspects of students’ programs: Can we make them measurable?”, in *Frontiers in Education Conference*, 2016, pp. 1–8. DOI: `10.1109/fie.2016.7757725`.

[18] D. Arnow and O. Barshay, “WebToTeach: an interactive focused programming exercise system”, in *Frontiers in Education Conference*, vol. 1, 1999, pp. 39–44. DOI: `10.1109/FIE.1999.839303`.

[19] A. Barr and M. Beard, “An instructional interpreter for Basic”, *ACM SIGCSE Bulletin*, vol. 8, no. 1, pp. 325–334, 1976. DOI: `10.1145/952989.803494`.

[20] A. Barr, M. Beard, and R. C. Atkinson, "A rationale and description of a CAI program to teach the BASIC programming language", *Instructional Science*, vol. 4, no. 1, pp. 1–31, 1975. DOI: 10.1007/BF00157068.

[21] A. Barr, M. Beard, and R. C. Atkinson, "The computer as a tutorial laboratory: the Stanford BIP project", *International Journal of Man-Machine Studies*, vol. 8, no. 5, pp. 567–596, 1976. DOI: 10.1016/S0020-7373(76)80021-1.

[22] M. L. Barrón-Estrada, R. Zatarain-Cabada, F. G. Hernández, R. O. Bustillos, and C. A. Reyes-García, "An Affective and Cognitive Tutoring System for Learning Programming", in *Advances in Artificial Intelligence and Its Applications*, vol. 9414 LNCS, 2015, pp. 171–182. DOI: 10.1007/978-3-319-27101-9_12.

[23] L. N. de Barros and K. V. Delgado, "Model Based Diagnosis of Student Programs", in *Monet Workshop on Model-Based System at ECAI*, 2006.

[24] B. A. Becker, P. Denny, R. Pettit, D. Bouchard, D. J. Bouvier, B. Harrington, A. Kamil, A. Karkare, C. McDonald, P.-M. Osera, *et al.*, "Compiler error messages considered unhelpful: The landscape of text-based programming error message research", in *ITiCSE, Working Group Reports*, 2019, pp. 177–210. DOI: 10.1145/3344429.3372508.

[25] C. Beierle, M. Kulaš, and M. Widera, "Automatic analysis of programming assignments", in *DeLFI: Die 1. e-Learning Fachtagung Informatik*, 2003, pp. 144–153.

[26] C. Beierle, M. Kulaš, and M. Widera, "Partial Specifications of Program Properties", in *International Workshop on Teaching Logic Programming*, 2004, pp. 18–34.

[27] S. Benford, E. Burke, and E. Foxley, "Learning to construct quality software with the Ceilidh system", *Software Quality Journal*, vol. 2, no. 3, pp. 177–197, 1993. DOI: 10.1007/BF00402268.

[28] S. Benford, E. Burke, E. Foxley, N. Gutteridge, and A. M. Zin, "Early experiences of computer-aided assessment and administration when teaching computer programming", *Research in Learning Technology*, vol. 1, no. 2, pp. 55–70, 1993. DOI: 10.3402/rlt.v1i2.9481.

[29] S. Benford, E. Burke, E. Foxley, and C. Higgins, “The Ceilidh system for the automatic grading of students on programming courses”, in *ACM Southeast Conference*, 1995, pp. 176–182. DOI: 10.1145/1122018.1122050.

[30] J. Bennedsen and M. E. Caspersen, “Failure rates in introductory programming”, *ACM SIGCSE Bulletin*, vol. 39, no. 2, pp. 32–36, 2007. DOI: 10.1145/1272848.1272879.

[31] J. Bennedsen and M. E. Caspersen, “Failure rates in introductory programming: 12 years later”, *ACM Inroads*, vol. 10, no. 2, pp. 30–36, 2019. DOI: 10.1145/3324888.

[32] H. Blau and J. E. B. Moss, “FrenchPress Gives Students Automated Feedback on Java Program Flaws”, in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2015, pp. 15–20. DOI: 10.1145/2729094.2742622.

[33] M. Blumenstein, S. Green, S. Fogelman, A. Nguyen, and V. Muthukkumarasamy, “Performance analysis of GAME: A generic automated marking environment”, *Computers & Education*, vol. 50, pp. 1203–1216, 2008. DOI: 10.1016/j.compedu.2006.11.006.

[34] M. Blumenstein, S. Green, A. Nguyen, and V. Muthukkumarasamy, “GAME: A generic automated marking environment for programming assessment”, in *Conference on Information Technology: Coding and Computing*, vol. 1, 2004, pp. 212–216. DOI: 10.1109/ITCC.2004.1286454.

[35] J. G. Bonar and R. Cunningham, “Bridge: Intelligent Tutoring with Intermediate Representations”, Carnegie Mellon University, University of Pittsburgh, Tech. Rep., 1988.

[36] J. Börstler, M. Nordström, and J. H. Paterson, “On the quality of examples in introductory Java textbooks”, *ACM Transactions on Computing Education (TOCE)*, vol. 11, no. 1, pp. 1–21, 2011. DOI: 10.1145/1921607.1921610.

[37] J. Börstler, H. Störrle, D. Toll, J. van Assema, R. Duran, S. Hooshangi, J. Jeuring, H. Keuning, C. Kleiner, and B. MacKellar, “"I know it when I see it" Perceptions of Code Quality”, in *ITiCSE, Working Group Reports*, 2017, pp. 70–85. DOI: 10.1145/3174781.3174785.

[38] D. Boud and E. Molloy, Eds., *Feedback in higher and professional education: understanding it and doing it well.* Routledge, 2012. DOI: 10.4324/9780203074336.

[39] D. Breuker, J. Derriks, and J. Brunekreef, "Measuring static quality of student code", in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2011, pp. 13–17. DOI: 10.1145/1999747.1999754.

[40] F. Brooks Jr, *The mythical man-month.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[41] N. C. C. Brown, M. Kölling, D. McCall, and I. Utting, "Blackbox: A Large Scale Repository of Novice Programmers' Activity", in *SIGCSE Technical Symposium on Computer Science Education*, 2014, pp. 223–228. DOI: 10.1145/2538862.2538924.

[42] N. C. Brown and A. Altadmri, "Novice Java programming mistakes: Large-scale data vs. educator beliefs", *ACM Transactions on Computing Education (TOCE)*, vol. 17, no. 2, p. 7, 2017. DOI: 10.1145/2994154.

[43] P. Brusilovsky, "Intelligent Tutor, Environment and Manual for Introductory Programming", *Innovations in Education & Training International*, vol. 29, no. 1, pp. 26–34, 1992. DOI: 10.1080/0954730920290104.

[44] P. Brusilovsky, S. Edwards, A. Kumar, L. Malmi, L. Benotti, D. Buck, P. Ihantola, R. Prince, T. Sirkiä, S. Sosnovsky, *et al.*, "Increasing adoption of smart learning content for computer science education", in *ITiCSE, Working Group Reports*, 2014, pp. 31–57. DOI: 10.1145/2713609.2713611.

[45] P. Brusilovsky and G. Weber, "Collaborative Example Selection in an Intelligent Example-Based Programming Environment", in *Conference on Learning Sciences*, 1996, pp. 357–362.

[46] J. C. Caiza and J. M. Del Alamo, "Programming assignments automatic grading: review of tools and implementations", in *International Technology, Education and Development Conference*, 2013, pp. 5691–5700.

[47] J. Carter, K. Ala-Mutka, U. Fuller, M. Dick, J. English, W. Fone, and J. Sheard, "How shall we assess this?", in *ITiCSE, Working Group Reports*, 2003, pp. 107–123. DOI: 10.1145/960875.960539.

[48] M. E. Caspersen and J. Bennedsen, "Instructional design of a programming course: A learning theoretic approach", in *Workshop on Computing Education Research*, 2007, pp. 111–122. DOI: 10.1145/1288580.1288595.

[49] K. E. Chang, B. C. Chiao, S. W. Chen, and R. S. Hsiao, "A programming learning system for beginners - A completion strategy approach 2000", *IEEE Transactions on Education*, vol. 43, no. 2, pp. 211–220, 2000. DOI: 10.1109/13.848075.

[50] B. Cheang, A. Kurnia, A. Lim, and W.-C. Oon, "On automated grading of programming assignments in an academic institution", *Computers & Education*, vol. 41, no. 2, pp. 121–131, 2003. DOI: 10.1016/S0360-1315(03)00030-7.

[51] Y. S. Chee, "Cognitive apprenticeship and its application to the teaching of Smalltalk in a multimedia interactive learning environment", *Instructional Science*, vol. 23, no. 1-3, pp. 133–161, 1995. DOI: 10.1007/BF00890449.

[52] R. R. Choudhury, H. Yin, and A. Fox, "Scale-Driven Automatic Hint Generation for Coding Style", in *Intelligent Tutoring Systems*, 1, vol. 9684 LNCS, 2016, pp. 122–132. DOI: 10.1007/978-3-319-39583-8_12.

[53] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs", *ACM SIGPLAN Notices*, vol. 46, no. 4, pp. 53–64, 2011. DOI: 10.1145/357766.351266.

[54] A. T. Corbett and J. R. Anderson, "Student modeling in an intelligent programming tutor", in *Cognitive Models and Intelligent Environments for Learning Programming*, vol. 111, 1993, pp. 135–144. DOI: 10.1007/978-3-662-11334-9_13.

[55] A. T. Corbett and J. R. Anderson, "Knowledge tracing: Modeling the acquisition of procedural knowledge", *User Modelling and User-Adapted Interaction*, vol. 4, no. 4, pp. 253–278, 1994. DOI: 10.1007/BF01099821.

[56] A. T. Corbett and J. R. Anderson, "Locus of Feedback Control in Computer-Based Tutoring: Impact on Learning Rate, Achievement and Attitudes", in *SIGCHI Conference on Human factors in computing systems*, 2001, pp. 245–252. DOI: 10.1145/365024.365111.

[57] A. T. Corbett, J. R. Anderson, and E. J. Patterson, "Student Modeling and Tutoring Flexibility in the Lisp Intelligent Tutoring System", in *Intelligent Tutoring Systems*, 1990, pp. 83–106.

[58] T. Crow, A. Luxton-Reilly, and B. Wuensche, "Intelligent tutoring systems for programming education: A systematic review", in *Australasian Computing Education Conf.*, 2018. DOI: `10.1145/3160489.3160492`.

[59] T. Dadic, "Intelligent Tutoring System for Learning Programming", in *Intelligent Tutoring Systems in E-Learning Environments*, IGI Global, 2011, pp. 166–186. DOI: `10.4018/978-1-61692-008-1.ch009`.

[60] T. Dadic, S. Stankov, and M. Rosic, "Meaningful Learning in the Tutoring System for Programming", in *Conference on Information Technology Interfaces*, 2008, pp. 483–488. DOI: `10.1109/iti.2008.4588458`.

[61] R. L. Danielson, "Pattie: An automated tutor for top-down programming", PhD thesis, University of Illinois at Urbana-Champaign, 1975.

[62] G. De Ruvo, E. Tempero, A. Luxton-Reilly, G. B. Rowe, and N. Giacaman, "Understanding semantic style by analysing student code", in *Australasian Computing Education Conference*, 2018, pp. 73–82. DOI: `10.1145/3160489.3160500`.

[63] D. M. De Souza, J. C. Maldonado, and E. F. Barbosa, "ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities", in *IEEE Conference on Software Engineering Education and Training*, 2011, pp. 1–10. DOI: `10.1109/CSEET.2011.5876088`.

[64] D. M. De Souza, B. H. Oliveira, J. C. Maldonado, S. R. S. Souza, and E. F. Barbosa, "Towards the use of an automatic assessment system in the teaching of software testing", in *Frontiers in Education Conference*, 2014, pp. 1–8. DOI: `10.1109/fie.2014.7044375`.

[65] D. M. De Souza, S. Isotani, and E. F. Barbosa, "Teaching novice programmers using ProgTest", *Int. Journal of Knowledge and Learning*, vol. 10, no. 1, pp. 60–77, 2015. DOI: `10.1504/ijkl.2015.071054`.

[66] F. P. Deek, K.-W. Ho, and H. Ramadhan, "A critical analysis and evaluation of web-based environments for program development", *The Internet and Higher Education*, vol. 3, no. 4, pp. 223–269, 2000. DOI: `10.1016/s1096-7516(01)00038-0`.

[67] F. P. Deek and J. A. McHugh, “A survey and critical analysis of tools for learning programming”, *Computer Science Education*, vol. 8, no. 2, pp. 130–178, 1998. DOI: 10.1076/csed.8.2.130.3820.

[68] S. Demeyer, F. Van Rysselberghe, T. Girba, J. Ratzinger, R. Marinescu, T. Mens, B. Du Bois, D. Janssens, S. Ducasse, M. Lanza, *et al.*, “The lan-simulation: A refactoring teaching example”, in *International Workshop on Principles of Software Evolution*, IEEE, 2005, pp. 123–131. DOI: 10.1109/IWPSE.2005.30.

[69] T. Dingsøyr and C. Lassenius, “Emerging themes in agile software development: Introduction to the special section on continuous value delivery”, *Information and Software Technology*, vol. 77, pp. 56–60, 2016. DOI: 10.1016/j.infsof.2016.04.018.

[70] C. Douce, D. Livingstone, and J. Orwell, “Automatic test-based assessment of programming: A review”, *Journal on Educational Resources in Computing*, vol. 5, no. 3, 2005. DOI: 10.1145/1163405.1163409.

[71] B. Du Boulay, “Some difficulties of learning to program”, *Journal of Educational Computing Research*, vol. 2, no. 1, pp. 57–73, 1986. DOI: 10.2190/3LFX-9RRF-67T8-UVK9.

[72] S. Edwards, N. Kandru, and M. Rajagopal, “Investigating static analysis errors in student Java programs”, in *ICER Conference on International Computing Education Research*, 2017, pp. 65–73. DOI: 10.1145/3105726.3106182.

[73] S. Edwards, J. Spacco, and D. Hovemeyer, “Can industrial-strength static analysis be used to help students who are struggling to complete programming activities?”, in *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 2019, pp. 7825–7834. DOI: 10.24251/HICSS.2019.941.

[74] S. H. Edwards, “Improving student performance by evaluating how well students test their own programs”, *Journal on Educational Resources in Computing*, vol. 3, no. 3, pp. 1–24, 2003. DOI: 10.1145/1029994.1029995.

[75] S. H. Edwards and M. A. Pérez-Quiñones, “Experiences using test-driven development with an automated grader”, *Journal of Computing Sciences in Colleges*, vol. 22, no. 3, pp. 44–50, 2007.

[76] J. English and T. English, “Experiences of using automated assessment in computer science courses”, *Journal of Information Technology Education: Innovations in Practice*, vol. 14, pp. 237–254, 2015. DOI: 10.28945/2304.

[77] N. L. Ensmenger, *The computer boys take over: Computers, programmers, and the politics of technical expertise.* Mit Press, 2012.

[78] A. Estey, H. Keuning, and Y. Coady, “Automatically classifying students in need of support by detecting changes in programming behaviour”, in *SIGCSE Technical Symposium on Computer Science Education*, ACM, 2017, pp. 189–194. DOI: 10.1145/3017680.3017790.

[79] W. Feurzeig, S. Papert, M. Bloom, R. Grant, and C. Solomon, “Programming languages as a conceptual framework for teaching mathematics”, *ACM SIGCUE Outlook*, vol. 4, no. 2, pp. 13–17, 1970. DOI: 10.1145/965754.965757.

[80] S. Fincher and A. Robins, Eds., *The Cambridge handbook of computing education research.* Cambridge University Press, 2019.

[81] S. Fincher, J. Tenenberg, B. Dorn, C. Hundhausen, R. McCartney, and L. Murphy, “Computing education research today”, in *The Cambridge handbook of computing education research*, Cambridge University Press, 2019, ch. 12. DOI: 10.1017/9781108654555.003.

[82] G. Fischer and J. W. von Gudenberg, “Improving the quality of programming education by online assessment”, in *Symposium on Principles and practice of programming in Java*, 2006, pp. 208–211. DOI: 10.1145/1168054.1168085.

[83] J. Fitzpatrick, “Applying the ABC Metric to C, C++, and Java”, Tech. Rep., 1997. [Online]. Available: https://www.softwarerenovation.com/ABCMetric.pdf.

[84] M. Fowler, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

[85] E. Foxley and C. A. Higgins, “The CourseMaster CBA System: Improvements over Ceilidh Improvements over Ceilidh”, in *CAA Conference, Loughborough*, Loughborough University, 2001.

[86] S. Garner, P. Haden, and A. Robins, “My program is correct but it doesn’t run: A preliminary investigation of novice programmers’ problems”, in *Australasian Conference on Computing Education*, 2005, pp. 173–180.

[87] T. S. Gegg-Harrison, “Exploiting Program Schemata in a Prolog Tutoring System”, PhD thesis, 1993.

[88] A. Gerdes, B. Heeren, J. Jeuring, and L. T. van Binsbergen, “Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback”, *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 65–100, 2017. DOI: `10.1007/s40593-015-0080-x`.

[89] A. Gerdes, J. Jeuring, and B. Heeren, “Using strategies for assessment of programming exercises”, in *SIGCSE Technical Symposium on Computer Science Education*, 2010, pp. 441–445. DOI: `10.1145/1734263.1734412`.

[90] A. Gerdes, J. Jeuring, and B. Heeren, “An interactive functional programming tutor”, in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, ACM, 2012, pp. 250–255. DOI: `10.1145/2325296.2325356`.

[91] M. Ghosh, B. K. Verma, and A. T. Nguyen, “An Automatic Assessment Marking And Plagiarism Detection”, in *Conference on Information Technology and Applications*, 2002, ISBN: 5006951416568.

[92] M. Goedicke, M. Striewe, and M. Balz, “Computer aided assessments and programming exercises with JACK”, Tech. Rep. 28, 2008.

[93] M. Gómez-Albarrán, “The Teaching and Learning of Programming: A Survey of Supporting Software Tools”, *The Computer Journal*, vol. 48, no. 2, pp. 130–144, 2005. DOI: `10.1093/comjnl/bxh080`.

[94] O. Gotel, C. Scharff, and A. Wildenberg, “Teaching software quality assurance by encouraging student contributions to an open source web-based system for the assessment of programming assignments”, *ACM SIGCSE Bulletin*, vol. 40, no. 3, pp. 214–218, 2008. DOI: `10.1145/1597849.1384329`.

[95] O. Gotel, C. Scharff, A. Wildenberg, M. Bousso, C. Bunthoeurn, P. Des, V. Kulkarni, S. P. N. Ayudhya, C. Sarr, and T. Sunetnanta, “Global perceptions on the use of WeBWorK as an online tutor for computer science”, in *Frontiers in Education Conference*, 2008, pp. 5–10. DOI: `10.1109/FIE.2008.4720331`.

[96] P. Gross and K. Powers, "Evaluating assessments of novice programming environments", in *International Workshop on Computing Education Research*, 2005, pp. 99–110. DOI: 10.1145/1089786.1089796.

[97] S. Gross, B. Mokbel, B. Hammer, and N. Pinkwart, "Learning Feedback in Intelligent Tutoring Systems", *Künstliche Intelligenz*, vol. 29, no. 4, pp. 413–418, 2015. DOI: 10.1007/s13218-015-0367-y.

[98] S. Gross, B. Mokbel, B. Paassen, B. Hammer, and N. Pinkwart, "Example-based Feedback Provision using Structured Solution Spaces", *International Journal of Learning Technology*, vol. 9, no. 3, pp. 248–280, 2014. DOI: 10.1504/IJLT.2014.065752.

[99] S. Gross and N. Pinkwart, "Towards an Integrative Learning Environment for Java Programming", in *IEEE Conference on Advanced Learning Technologies*, 2015, pp. 24–28. DOI: 10.1109/ICALT.2015.75.

[100] S. Gulwani, I. Radiček, and F. Zuleger, "Feedback generation for performance problems in introductory programming assignments", in *SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 41–51. DOI: 10.1145/2635868.2635912.

[101] P. J. Guo, "Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education", in *SIGCSE Technical Symposium on Computer Science Education*, 2013, 579–584. DOI: 10.1145/2445196.2445368.

[102] M. Guzdial, "Programming environments for novices", in *Computer Science Education Research*, S. Fincher and M. Petre, Eds., 2004, pp. 127–154. DOI: 10.1201/9781482287325-20.

[103] M. Guzdial and B. du Boulay, "The history of computing education research", in *The Cambridge handbook of computing education research*, S. Fincher and A. Robins, Eds., Cambridge University Press, 2019, ch. 1. DOI: 10.1017/9781108654555.002.

[104] B. Hartanto, "Incorporating Anchored Learning in a C# Intelligent Tutoring System", PhD thesis, Queensland University of Technology, 2014, ISBN: 9786028040730.

[105] B. Hartanto and J. Reye, "CSTutor: An Intelligent Tutoring System that Supports Natural Learning", in *Conference on Computer Science Education Innovation and Technology*, 2013, pp. 19–26. DOI: 10.5176/2251-2195_CSEIT13.09.

[106] H. M. Hasan, "Assessment of Student Programming Assignments in COBOL", *Education and Computing*, vol. 4, pp. 99–107, 1988. DOI: 10.1016/s0167-9287(88)90574-2.

[107] J. Hattie and H. Timperley, "The power of feedback", *Review of Educational Research*, vol. 77, no. 1, pp. 81–112, 2007. DOI: 10.3102/003465430298487.

[108] Y. He, M. Ikeda, and R. Mizoguchi, "Helping novice programmers bridge the conceptual gap", in *Conference on Expert Systems for Development*, IEEE, 1994, pp. 192–197. DOI: 10.1109/ICESD.1994.302282.

[109] B. Heeren and J. Jeuring, "Feedback services for stepwise exercises", *Science of Computer Programming*, vol. 88, pp. 110–129, 2014. DOI: 10.1016/j.scico.2014.02.021.

[110] B. Heeren and J. Jeuring, "Automated feedback for mathematical learning environments", in *ICTMT International Conference on Technology in Mathematics Teaching, to appear*, 2019.

[111] M. T. Helmick, "Interface-based programming assignments and automatic grading of Java programs", *ACM SIGCSE Bulletin*, vol. 39, no. 3, pp. 63–67, 2007. DOI: 10.1145/1269900.1268805.

[112] C. A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas, "Automated assessment and experiences of teaching programming", *Journal on Educational Resources in Computing*, vol. 5, no. 3, 2005. DOI: 10.1145/1163405.1163410.

[113] C. A. Higgins and F. Z. Mansouri, "PRAM: A Courseware System for the Automatic Assessment of AI Programs", in *Innovative Teaching and Learning*, vol. 1, 2000, pp. 311–329. DOI: 10.1007/978-3-7908-1868-0_10.

[114] C. A. Higgins, P. Symeonidis, and A. Tsintsifas, "The marking system for CourseMaster", *ACM SIGCSE Bulletin*, vol. 34, no. 3, pp. 46–50, 2002. DOI: 10.1145/637610.544431.

[115] J. Holland, A. Mitrovic, and B. Martin, "J-LATTE: a constraint-based tutor for Java", in *Conference on Computers in Education*, 2009, pp. 142–146.

[116] J. Hong, "Guided programming and automated error analysis in an intelligent Prolog tutor", vol. 61, no. 4, pp. 505–534, 2004. DOI: 10.1016/j.ijhcs.2004.02.001.

[117] D. Hovemeyer and W. Pugh, “Finding bugs is easy”, *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004. DOI: 10.1145/1052883.1052895.

[118] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury, “Refactoring based program repair applied to programming assignments”, in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, 388–398. DOI: 10.1109/ASE.2019.00044.

[119] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, “Review of recent systems for automatic assessment of programming assignments”, in *Koli Calling International Conference on Computing Education Research*, 2010, pp. 86–93. DOI: 10.1145/1930464.1930480.

[120] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers, M. A. Rubio, J. Sheard, B. Skupas, J. Spacco, C. Szabo, and D. Toll, “Educational data mining and learning analytics in programming: Literature review and case studies”, in *ITiCSE, Working Group Reports*, 2015, pp. 41–63. DOI: 10.1145/2858796.2858798.

[121] C. Innocenti, C. Massucco, D. Persico, and L. Sarti, “Ugo: An intelligent tutoring system for Prolog”, in *PEG Conference on Knowledge Based Environments for Teaching and Learning*, 1991, pp. 322–329.

[122] D. Jackson, “A software system for grading student computer programs”, *Computers & Education*, vol. 27, no. 3-4, pp. 171–180, 1996. DOI: 10.1016/S0360-1315(96)00025-5.

[123] D. Jackson, “A semi-automated approach to online assessment”, *ACM SIGCSE Bulletin*, vol. 32, no. 3, pp. 164–167, 2000. DOI: 10.1145/353519.343160.

[124] D. Jackson and M. Usher, “Grading student programs using ASSYST”, *ACM SIGCSE Bulletin*, vol. 29, no. 1, pp. 335–339, 1997. DOI: 10.1145/268085.268210.

[125] J. Jansen, A. Oprescu, and M. Bruntink, “The impact of automated code quality feedback in programming education”, in *Post-proceedings of the Tenth Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE)*, 2017.

[126] J. Jeuring, L. T. van Binsbergen, A. Gerdes, and B. Heeren, "Model solutions and properties for diagnosing student programs in Ask-Elle", in *Computer Science Education Research Conference*, 2014, pp. 31–40. DOI: 10.1145/2691352.2691355.

[127] J. Jeuring, F. Grosfeld, B. Heeren, M. Hulsbergen, R. IJntema, V. Jonker, N. Mastenbroek, M. van der Smagt, F. Wijmans, M. Wolters, *et al.*, "Communicate!—a serious game for communication skills—", in *Design for teaching and learning in a networked world*, 2015, pp. 513–517. DOI: 10.1007/978-3-319-24258-3_49.

[128] W. Jin, T. Barnes, and J. Stamper, "Program representation for automatic hint generation for a data-driven novice programming tutor", in *Intelligent Tutoring Systems*, 2012, pp. 304–309. DOI: 10.1007/978-3-642-30950-2_40.

[129] W. Jin, A. Corbett, W. Lloyd, L. Baumstark, and C. Rolka, "Evaluation of Guided-Planning and Assisted-Coding with Task Relevant Dynamic Hinting", in *Intelligent Tutoring Systems*, 2014, pp. 318–328. DOI: 10.1007/978-3-319-07221-0_40.

[130] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?", in *International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 672–681. DOI: 10.1109/ICSE.2013.6606613.

[131] W. L. Johnson, "Understanding and debugging novice programs", *Artificial Intelligence*, vol. 42, no. 1, pp. 51–97, 1990. DOI: 10.1016/0004-3702(90)90094-G.

[132] W. L. Johnson and E. Soloway, "Intention-based diagnosis of novice programming errors", in *AAAI conference*, 1984, pp. 162–168. DOI: 10.1109/mex.1987.4307101.

[133] W. L. Johnson and E. Soloway, "PROUST: Knowledge-based program understanding", *IEEE Transactions on Software Engineering*, vol. 11, no. 3, pp. 267–275, 1985. DOI: 10.1109/tse.1985.232210.

[134] Joint Task Force on Computing Curricula, ACM and IEEE Computer Society, *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. 2013. DOI: 10.1145/2534860.

[135] F. Jurado, M. Redondo, and M. Ortega, "Using fuzzy logic applied to software metrics and test cases to assess programming assignments and give advice", *Journal of Network and Computer Applications*, vol. 35, no. 2, pp. 695–712, 2012. DOI: 10.1016/j.jnca.2011.11.002.

[136] F. Jurado, M. Redondo, and M. Ortega, "eLearning standards and automatic assessment in a distributed eclipse based environment for learning computer programming", *Computer Applications in Engineering Education*, vol. 22, no. 4, pp. 774–787, 2014. DOI: 10.1002/cae.21569.

[137] S. Karkalas and S. Gutierrez-Santos, "Enhanced JavaScript learning using code quality tools and a rule-based system in the FLIP exploratory learning environment", in *IEEE International Conference on Advanced Learning Technologies*, 2014, pp. 84–88. DOI: 10.1109/ICALT.2014.35.

[138] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers", *ACM Computing Surveys*, vol. 37, no. 2, pp. 83–137, 2005.

[139] H. Keuning, "Strategy-based feedback for imperative programming exercises", Master's thesis, Open University of The Netherlands, 2014. [Online]. Available: http://www.hkeuning.nl/Thesis%20Hieke%20Keuning%20final.pdf.

[140] H. Keuning, B. Heeren, and J. Jeuring, "Strategy-based feedback in a programming tutor", in *Computer Science Education Research Conference*, 2014, pp. 43–54. DOI: 10.1145/2691352.2691356.

[141] H. Keuning, B. Heeren, and J. Jeuring, "Code Quality Issues in Student Programs", in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2017, pp. 110–115. DOI: 10.1145/3059009.3059061.

[142] H. Keuning, B. Heeren, and J. Jeuring, "How teachers would help students to improve their code", in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2019, pp. 119–125. DOI: 10.1145/3304221.3319780.

[143] H. Keuning, B. Heeren, and J. Jeuring, "A tutoring system to learn code refactoring", In submission, n.d.

[144] H. Keuning, B. Heeren, and J. Jeuring, "Student refactoring behaviour in a programming tutor", In submission, n.d.

[145] H. Keuning, J. Jeuring, and B. Heeren, "Towards a Systematic Review of Automated Feedback Generation for Programming Exercises", in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2016, pp. 41–46. DOI: `10.1145/2899415.2899422`.

[146] H. Keuning, J. Jeuring, and B. Heeren, "Towards a systematic review of automated feedback generation for programming exercises – extended version", Tech. Rep. UU-CS-2016-001, 2016. [Online]. Available: `https://dspace.library.uu.nl/bitstream/handle/1874/346321/Extended.pdf`.

[147] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises", *ACM Transactions on Computing Education (TOCE)*, vol. 19, no. 1, p. 3, 2018. DOI: `10.1145/3231711`.

[148] S.-M. Kim and J. H. Kim, "A hybrid approach for program understanding based on graph parsing and expectation-driven analysis", *Applied Artificial Intelligence*, vol. 12, no. 6, pp. 521–546, 1998. DOI: `10.1080/088395198117659`.

[149] D. Kirk, T. Crow, A. Luxton-Reilly, and E. Tempero, "On assuring learning about code quality", in *Australasian Conf. on Computing Education*, 2020, pp. 86–94. DOI: `10.1145/3373165.3373175`.

[150] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering", Tech. Rep. EBSE-2007-01, 2007.

[151] H. Koh and D. M.-J. Wu, "Goal-directed semantic tutor", in *Conference on industrial and engineering applications of artificial intelligence and expert systems*, 1988, pp. 171–176. DOI: `10.1145/51909.51930`.

[152] C. Köllmann and M. Goedicke, "Automation of Java Code Analysis for Programming Exercises", in *Workshop on Graph Based Tools, Electronic Communications of the EASST*, vol. 1, 2006, pp. 1–12. DOI: `10.14279/tuj.eceasst.1.78`.

[153] C. Köllmann and M. Goedicke, "A Specification Language for Static Analysis of Student Exercises", in *Conference on Automated Software Engineering*, 2008, pp. 355–358. DOI: `10.1109/ASE.2008.47`.

[154] U. Kose and O. Deperlioglu, "Intelligent Learning Environments Within Blended Learning for Ensuring Effective C Programming Course", *International Journal of Artificial Intelligence & Applications*, vol. 3, no. 1, pp. 105–124, 2012. DOI: `10.5121/ijaia.2012.3109`.

[155] A. Kyrilov and D. C. Noelle, "Binary Instant Feedback on Programming Exercises Can Reduce Student Engagement and Promote Cheating", in *Koli Calling International Conference on Computing Education Research*, 2015, pp. 122–126. DOI: `10.1145/2828959.2828968`.

[156] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers", *ACM SIGCSE Bulletin*, vol. 37, no. 3, pp. 14–18, 2005. DOI: `10.1145/1067445.1067453`.

[157] H. C. Lane and K. VanLehn, "Teaching the tacit knowledge of programming to novices with natural language tutoring", *Computer Science Education*, pp. 183–201, 2005. DOI: `10.1080/08993400500224286`.

[158] T. Lazar and I. Bratko, "Data-Driven Program Synthesis for Hint Generation in Programming Tutors", in *Intelligent Tutoring Systems*, 2014, pp. 306–311. DOI: `10.1007/978-3-319-07221-0_38`.

[159] N.-T. Le, "A classification of adaptive feedback in educational systems for programming", *Systems*, 2016. DOI: `10.3390/systems4020022`.

[160] N.-T. Le and W. Menzel, "Problem Solving Process oriented Diagnosis in Logic Programming", in *Conference on Computers in Education*, 2006, pp. 63–70.

[161] N.-T. Le, W. Menzel, and N. Pinkwart, "Evaluation of a Constraint-Based Homework Assistance System for Logic Programming", in *Conference on Computers in Education*, 2009, pp. 51–58, ISBN: 9789868473539.

[162] N.-T. Le and N. Pinkwart, "Adding weights to constraints in intelligent tutoring systems: Does it improve the error diagnosis?", in *Towards Ubiquitous Learning*, vol. LNCS 6964, 2011, pp. 233–247. DOI: `10.1007/978-3-642-23985-4_19`.

[163] N.-T. Le and N. Pinkwart, "INCOM: A Web-based Homework Coaching System For Logic Programming", in *Conference on Cognition and Exploratory Learning in Digital Age*, 2011, pp. 43–50.

[164] N.-T. Le and N. Pinkwart, "Towards a classification for programming exercises", in *Workshop on AI-supported Education for Computer Science*, 2014, pp. 51–60.

[165] N.-T. Le, S. Strickroth, S. Gross, and N. Pinkwart, “A review of ai-supported tutoring approaches for learning programming”, in *Advanced Computational Methods for Knowledge Engineering*, 2013, pp. 267–279. DOI: 10.1007/978-3-319-00293-4_20.

[166] C. Lewis, “Attitudes and beliefs about computer science among students and faculty”, *ACM SIGCSE Bulletin*, vol. 39, no. 2, pp. 37–41, 2007. DOI: 10.1145/1272848.1272880.

[167] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, *et al.*, “A multinational study of reading and tracing skills in novice programmers”, *ACM SIGCSE Bulletin*, vol. 36, no. 4, pp. 119–150, 2004. DOI: 10.1145/1044550.1041673.

[168] J. Lodder, B. Heeren, and J. Jeuring, “A comparison of elaborated and restricted feedback in logex, a tool for teaching rewriting logical formulae”, *Journal of Computer Assisted Learning*, vol. 35, no. 5, pp. 620–632, 2019. DOI: 10.1111/jcal.12365.

[169] C.-K. Looi, “Automatic debugging of Prolog programs in a Prolog Intelligent Tutoring System”, *Instructional Science*, vol. 20, no. 2-3, pp. 215–263, 1991. DOI: 10.1007/BF00120883.

[170] S. Lowes, “Online teaching and classroom change: The impact of virtual high school on its teachers and their schools”, Columbia University, Institute for Learning Technologies, Tech. Rep., 2007.

[171] Y. Lu, X. Mao, T. Wang, G. Yin, and Z. Li, “Improving students’ programming quality with the continuous inspection process: A social coding perspective”, *Frontiers of Computer Science*, vol. 14, no. 5, 2019. DOI: 10.1007/s11704-019-9023-2.

[172] A. Luxton-Reilly, “Learning to program is easy”, in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2016, pp. 284–289. DOI: 10.1145/2899415.2899432.

[173] A. Luxton-Reilly, I. Albluwi, B. A. Becker, M. Giannakos, A. N. Kumar, L. Ott, J. Paterson, M. J. Scott, J. Sheard, C. Szabo, *et al.*, “Introductory programming: A systematic literature review”, in *ITiCSE, Working Group Reports*, 2018, pp. 55–106. DOI: 10.1145/3293881.3295779.

[174] A. Luxton-Reilly, P. Denny, D. Kirk, E. Tempero, and S.-Y. Yu, "On the differences between correct student solutions", in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2013, pp. 177–182. DOI: 10.1145/2462476.2462505.

[175] C. MacNish, "Java Facilities for Automating Analysis, Feedback and Assessment of Laboratory Work", *Computer Science Education*, vol. 10, no. 2, pp. 147–163, 2000. DOI: 10.1076/0899-3408(200008)10:2;1-C;FT147.

[176] C. MacNish, "Machine Learning and Visualisation Techniques for Inferring Logical Errors in Student Code Submissions", in *IEEE Conference on Advanced Learning Technologies*, 2002, pp. 317–321.

[177] T. A. Majchrzak and C. A. Usener, "Evaluating the Synergies of Integrating E-Assessment and Software Testing", in *Information Systems Development*, 2013, pp. 179–193. DOI: 10.1007/978-1-4614-4951-5_15.

[178] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti, "Visual algorithm simulation exercise system with automatic assessment: Trakla2", *Informatics in education*, vol. 3, no. 2, p. 267, 2004.

[179] A. K. Mandal, C. Mandal, and C. Reade, "A System for Automatic Evaluation of Programs for Correctness and Performance", in *Conferences Web Information Systems and Technologies 2005 and 2006*, 2007, pp. 367–380. DOI: 10.1007/978-3-540-74063-6_29.

[180] L. Mannila and M. de Raadt, "An objective comparison of languages for teaching introductory programming", in *Koli Calling International Conference on Computing Education Research*, 2006, pp. 32–37. DOI: 10.1145/1315803.1315811.

[181] F. Z. Mansouri, C. A. Gibbon, and C. A. Higgins, "PRAM: prolog automatic marker", in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 1998, pp. 166–170. DOI: 10.1145/282991.283108.

[182] V. J. Marin and C. R. Rivero, "Clustering recurrent and semantically cohesive program statements in introductory programming assignments", in *ACM International Conference on Information and Knowledge Management*, 2019, 911–920. DOI: 10.1145/3357384.3357960.

[183] S. Marwan, J. Jay Williams, and T. Price, "An evaluation of the impact of automated programming hints on performance and learning", in *ICER Conference on International Computing Education Research*, 2019, pp. 61–70. DOI: 10.1145/3291279.3339420.

[184] R. Matloobi, M. Blumenstein, and S. Green, "An Enhanced Generic Automated Marking Environment: GAME-2", *IEEE Multidisciplinary Engineering Education Magazine*, vol. 2, no. 2, pp. 55–60, 2007.

[185] R. Matloobi, M. Blumenstein, and S. Green, "Extensions to Generic Automated Marking Environment: Game-2+", in *Interactive Computer Aided Learning Conference*, vol. 1, 2009, pp. 1069–1076.

[186] G. McCalla, R. Bunt, and J. Harms, "The design of the SCENT automated advisor", *Computational Intelligence*, vol. 2, no. 1, pp. 76–92, 1986. DOI: 10.1111/j.1467-8640.1986.tb00073.x.

[187] G. McCalla, J. Greer, B. Barrie, and P. Pospisil, "Granularity Hierarchies", *Computers & Mathematics with Applications*, vol. 23, no. 2–5, pp. 363–375, 1992. DOI: 10.1016/0898-1221(92)90148-b.

[188] S. McConnell, *Code Complete: A Practical Handbook of Software Construction, Second Edition.* Microsoft Press, 2004.

[189] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students", in *ITiCSE, Working Group Reports*, 2001, pp. 125–180. DOI: 10.1145/572133.572137.

[190] J. McKendree, B. Radlinski, and M. E. Atwood, "The Grace Tutor: A qualified success", in *Intelligent Tutoring Systems*, 1992, pp. 677–684. DOI: 10.1007/3-540-55606-0_78.

[191] D. C. Merrill, B. J. Reiser, M. Ranney, and J. G. Trafton, "Effective tutoring techniques: A comparison of human tutors and intelligent tutoring systems", *Journal of the Learning Sciences*, vol. 2, no. 3, pp. 277–305, 1992. DOI: 10.1207/s15327809jls0203_2.

[192] A. Mitrovic, K. Koedinger, and B. Martin, "A Comparative Analysis of Cognitive Tutoring and Constraint-Based Modeling", in *User Modeling*, 2003, pp. 313–322. DOI: 10.1007/3-540-44963-9_42.

[193] J. Moghadam, R. R. Choudhury, H. Yin, and A. Fox, "AutoStyle: Toward Coding Style Feedback At Scale", in *ACM Conference on Learning @ Scale*, 2015, pp. 261–266. DOI: 10.1145/2818052.2874315.

[194] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?", in *International Conference on Software Reuse*, 2006, pp. 287–297. DOI: 10.1007/11763864_21.

[195] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it", *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011. DOI: 10.1109/TSE.2011.41.

[196] W. R. Murray, "Automatic program debugging for intelligent tutoring systems", *Computational Intelligence*, vol. 3, no. 1, pp. 1–16, 1987. DOI: 10.1111/j.1467-8640.1987.tb00169.x.

[197] S. Narciss, "Feedback strategies for interactive learning tasks", *Handbook of research on educational communications and technology*, pp. 125–144, 2008.

[198] P. Naur, "Automatic Grading of student's ALGOL Programming", *BIT Numerical Mathematics*, vol. 4, no. 3, pp. 177–188, 1964. DOI: 10.1016/S0360-1315(03)00030-7.

[199] G. L. Nelson, B. Xie, and A. J. Ko, "Comprehension first: Evaluating a novel pedagogy and tutoring system for program tracing in cs1", in *ICER Conference on International Computing Education Research*, 2017, pp. 2–11. DOI: 10.1145/3105726.3106178.

[200] A. Nguyen, C. Piech, J. Huang, and L. Guibas, "Codewebs: Scalable Homework Search for Massive Open Online Programming Courses", in *Conference on World wide web*, 2014, pp. 491–502. DOI: 10.1145/2566486.2568023.

[201] S. Nutbrown and C. Higgins, "Static analysis of programming exercises: Fairness, usefulness and a method for application", *Computer Science Education*, vol. 26, no. 2-3, pp. 104–128, 2016. DOI: 10.1080/08993408.2016.1179865.

[202] E. Odekirk-Hash and J. L. Zachary, "Automated feedback on programs means students need less help from teachers", *ACM SIGCSE Bulletin*, vol. 33, no. 1, pp. 55–59, 2001. DOI: 10.1145/366413.364537.

[203] C. Ott, A. Robins, and K. Shephard, “Translating Principles of Effective Feedback for Students into the CS1 Context”, *ACM Transactions on Computing Education (TOCE)*, vol. 16, no. 1, pp. 1–27, 2016. DOI: 10.1145/2737596.

[204] D. Parsons and P. Haden, “Parson’s programming puzzles: A fun and effective learning tool for first programming courses”, in *Australasian Conference on Computing Education*, 2006, pp. 157–163.

[205] M. Pärtel, M. Luukkainen, A. Vihavainen, and T. Vikberg, “Test My Code”, *International Journal of Technology Enhanced Learning*, vol. 5, no. 3-4, pp. 271–283, 2013. DOI: 10.1504/ijtel.2013.059495.

[206] E. Patitsas, J. Berlin, M. Craig, and S. Easterbrook, “Evidence that computer science grades are not bimodal”, *Communications of the ACM*, vol. 63, no. 1, pp. 91–98, 2019. DOI: 10.1145/3372161.

[207] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson, “A survey of literature on the teaching of introductory programming”, *ACM SIGCSE Bulletin*, vol. 39, no. 4, pp. 204–223, 2007. DOI: 10.1145/1345375.1345441.

[208] D. Perelman, J. Bishop, S. Gulwani, and D. Grossman, “Automated Feedback and Recognition through Data Mining in Code Hunt, MSR-TR-2015-57”, Microsoft Research, Tech. Rep., 2015.

[209] D. Perelman, S. Gulwani, and D. Grossman, “Test-Driven Synthesis for Automated Feedback for Introductory Computer Science Assignments”, in *Workshop on Data Mining for Educational Assessment and Feedback*, 2014.

[210] R. Pettit, J. Homer, R. Gee, S. Mengel, and A. Starbuck, “An Empirical Study of Iterative Improvement in Programming Assignments”, in *SIGCSE Technical Symposium on Computer Science Education*, 2015, pp. 410–415. DOI: 10.1145/2676723.2677279.

[211] R. Pettit and J. Prather, “Automated assessment tools: Too many cooks, not enough collaboration”, *Journal of Computing Sciences in Colleges*, vol. 32, no. 4, pp. 113–121, 2017.

[212] C. Peylo, T. Thelen, C. Rollinger, and H. Gust, “A Web-based intelligent educational system for PROLOG”, in *Workshop on Adaptive and Intelligent Web-Based Education Systems, ITS*, 2000, pp. 70–80.

[213] N. Pillay, "Developing intelligent programming tutors for novice programmers", *ACM SIGCSE Bulletin*, vol. 35, no. 2, pp. 78–82, 2003. DOI: 10.1145/782941.782986.

[214] J. C. Rodríguez-del Pino, E. Rubio-Royo, and Z. Hernández-Figueroa, "A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features", in *Conference on e-Learning, e-Business, Entreprise Information Systems, & e-Government*, 2012.

[215] Y. Pisan, D. Richards, A. Sloane, H. Koncek, and S. Mitchell, "Submit! A Web-Based System for Automatic Program Critiquing", in *Australasian Conference on Computing Education*, 2002, pp. 59–68.

[216] L. Porter, D. Bouvier, Q. Cutts, S. Grissom, C. Lee, R. McCartney, D. Zingaro, and B. Simon, "A multi-institutional study of peer instruction in introductory computing", *ACM Inroads*, vol. 7, no. 2, 76–81, 2016. DOI: 10.1145/2938142.

[217] I. Pribela, M. Ivanović, and Z. Budimac, "System for testing different kinds of students' programming assignments", in *Conference on Information Technology*, 2011.

[218] T. W. Price, Y. Dong, R. Zhi, B. Paaßen, N. Lytle, V. Cateté, and T. Barnes, "A comparison of the quality of data-driven programming hint generation algorithms", *International Journal of Artificial Intelligence in Education*, vol. 29, no. 3, pp. 368–395, 2019. DOI: 10.1007/s40593-019-00177-z.

[219] Y. Qian and J. Lehman, "Students' misconceptions and other difficulties in introductory programming: A literature review", *ACM Transactions on Computing Education (TOCE)*, vol. 18, no. 1, p. 1, 2017. DOI: 10.1145/3077618.

[220] L. Qiu and C. Riesbeck, "An incremental model for developing educational critiquing systems: experiences with the Java Critiquer", *Journal of Interactive Learning Research*, vol. 19, no. 1, pp. 119–145, 2008.

[221] R. Radlinski and J. McKendree, "Grace meets the real world: tutoring COBOL as a second language", in *SIGCHI Conference on Human factors in computing systems*, 1992, pp. 343–350. DOI: 10.1145/142750.142829.

[222] K. A. Rahman and M. J. Nordin, "A review on the static analysis approach in the automated programming assessment systems", in *National conference on programming*, 2007.

[223] H. A. Ramadhan, F. Deek, and K. Shihab, "Incorporating software visualization in the design of intelligent diagnosis systems for user programming", *Artificial Intelligence Review*, vol. 16, no. 1, pp. 61–84, 2001. DOI: `10.1023/A:1011078011415`.

[224] K. Rivers, "Automated data-driven hint generation for learning programming", 2017. [Online]. Available: `http://krivers.net/thesis.pdf`.

[225] K. Rivers and K. R. Koedinger, "Data-driven hint generation in vast solution spaces: a self-improving Python programming tutor", *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 37–64, 2017. DOI: `10.1007/s40593-015-0070-z`.

[226] A. Robins, "Learning edge momentum: A new account of outcomes in cs1", *Computer Science Education*, vol. 20, no. 1, pp. 37–71, 2010. DOI: `10.1080/08993401003612167`.

[227] A. Robins, "Novice programmers", in *The Cambridge handbook of computing education research*, S. Fincher and A. Robins, Eds., Cambridge University Press, 2019, ch. 12.

[228] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion", *Computer Science Education*, 2003. DOI: `10.1076/csed.13.2.137.14200`.

[229] R. Romli, S. Sulaiman, and K. Z. Zamli, "Automatic programming assessment and test data generation, A review on its approaches", in *International Symposium in Information Technology*, 2010, pp. 1186–1192.

[230] T. Rosenthal, P. Suppes, and N. Ben-Zvi, "Automated evaluation methods with attention to individual differences - A study of a computer-based course in C", in *Frontiers in Education Conference*, vol. 1, 2002, pp. 7–12. DOI: `10.1109/fie.2002.1157893`.

[231] G. R. Ruth, "Intelligent program analysis", *Artificial Intelligence*, vol. 7, pp. 65–85, 1976. DOI: `10.1016/0004-3702(76)90022-9`.

[232] W. Sack, "Knowledge base compilation and the language design game", in *Intelligent Tutoring Systems*, 1992, pp. 225–233. DOI: `10.1007/3-540-55606-0_29`.

[233] W. Sack and E. Soloway, “From PROUST to CHIRON: ITS Design as Iterative Engineering; Intermediate Results are Important!”, *Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches*, pp. 239–274, 1992.

[234] R. Saikkonen, L. Malmi, and A. Korhonen, “Fully automatic assessment of programming exercises”, vol. 33, no. 3, pp. 133 –136, 2001. DOI: 10.1145/507758.377666.

[235] J. A. Sant, “"Mailing it in": email-centric automated assessment”, *ACM SIGCSE Bulletin*, vol. 41, no. 3, pp. 308–312, 2009. DOI: 10.1145/1562877.1562971.

[236] G. M. Schneider, “The introductory programming course in computer science: Ten principles”, *ACM SIGCSE Bulletin*, pp. 107–114, 1978. DOI: 10.1145/990654.990598.

[237] S. C. Shaffer, “Ludwig: An Online Programming Tutoring and Assessment System”, *ACM SIGCSE Bulletin*, vol. 37, no. 2, pp. 56–60, 2005. DOI: 10.1145/1083431.1083464.

[238] G. Shimic and A. Jevremovic, “Problem-based learning in formal and informal learning environments”, *Interactive Learning Environments*, vol. 20, no. 4, pp. 351–367, 2012. DOI: 10.1080/10494820.2010.486685.

[239] V. J. Shute, “Focus on formative feedback”, *Review of Educational Research*, vol. 78, pp. 153–189, 2008. DOI: 10.3102/0034654307313795.

[240] Simon, A. Luxton-Reilly, V. V. Ajanovski, E. Fouh, C. Gonsalvez, J. Leinonen, J. Parkinson, M. Poole, and N. Thota, “Pass rates in introductory programming and in other stem disciplines”, in *ITiCSE, Working Group Reports*, 2019, 53–71. DOI: 10.1145/3344429.3372502.

[241] R. Singh, S. Gulwani, and A. Solar-Lezama, “Automated feedback generation for introductory programming assignments”, *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 15–26, 2013. DOI: 10.1145/2499370.2462195.

[242] M. Z. Smith and J. J. Ekstrom, “String of Perls: Using Perl to Teach Perl”, in *ASEE Annual Conference and Exposition*, 2004.

[243] S. Smith, S. Stoecklin, and C. Serino, “An innovative approach to teaching refactoring”, in *SIGCSE Technical Symposium on Computer Science Education*, Houston, Texas, USA, 2006, pp. 349–353. DOI: 10.1145/1121341.1121451.

[244] E. Soloway, E. Rubin, B. Woolf, J. Bonar, and W. L. Johnson, “Meno-II: An AI-based programming tutor”, *Journal of Computer-Based Instruction*, vol. 10, no. 1, 1983.

[245] J. S. Song, S. H. Hahn, K. Y. Tak, and J. H. Kim, “An intelligent tutoring system for introductory C language course”, *Computers & Education*, vol. 28, no. 2, pp. 93–102, 1997. doi: 10.1016/s0360-1315(97)00003-1.

[246] J. Sorva, V. Karavirta, and L. Malmi, “A Review of Generic Program Visualization Systems for Introductory Programming Education”, *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, pp. 1–64, 2013. doi: 10.1145/2490822.

[247] J. Sorva and T. Sirkiä, “Uuhistle: A software tool for visual program simulation”, in *Koli Calling International Conference on Computing Education Research*, 2010, pp. 49–54. doi: 10.1145/1930464.1930471.

[248] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez, “Experiences with marmoset: designing and using an advanced submission and testing system for programming courses”, *ACM SIGCSE Bulletin*, vol. 38, no. 3, pp. 13–17, 2006. doi: 10.1145/1140124.1140131.

[249] M. Stegeman, E. Barendsen, and S. Smetsers, “Designing a rubric for feedback on code quality in programming courses”, in *Koli Calling International Conference on Computing Education Research*, Koli, Finland, 2016, pp. 160–164. doi: doi.org/10.1145/2999541.2999555.

[250] S. Stoecklin, S. Smith, and C. Serino, “Teaching students to build well formed object-oriented methods through refactoring”, in *SIGCSE Technical Symposium on Computer Science Education*, 2007, pp. 145–149. doi: 10.1145/1227310.1227364.

[251] M. Striewe, M. Balz, and M. Goedicke, “A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking”, *Conference on Computer Supported Education*, vol. 2, pp. 54–61, 2009.

[252] M. Striewe and M. Goedicke, “Using Run Time Traces in Automated Programming Tutoring”, in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2011, pp. 303–307. doi: 10.1145/1999747.1999832.

[253] M. Striewe and M. Goedicke, “A review of static analysis approaches for programming exercises”, in *Computer Assisted Assessment. Research into E-Assessment*, 2014, pp. 100–113. DOI: 10.1007/978-3-319-08657-6_10.

[254] R. Suzuki, G. Soares, E. Glassman, A. Head, L. D’Antoni, and B. Hartmann, “Exploring the Design Space of Automatically Synthesized Hints for Introductory Programming Assignments”, in *SIGCHI Conference Extended Abstracts on Human factors in computing systems*, 2017. DOI: 10.1145/3027063.3053187.

[255] E. Sykes, “Qualitative Evaluation of the Java Intelligent Tutoring System”, *Journal of Systemics, Cybernetics and Informatics*, vol. 3, no. 5, pp. 49–60, 2005.

[256] E. Sykes, “Design, development and evaluation of the Java Intelligent Tutoring System”, *Technology, Instruction, Cognition & Learning*, vol. 8, no. 1, pp. 25–65, 2010.

[257] S. Tacoma, B. Heeren, J. Jeuring, and P. Drijvers, “Automated feedback on the structure of hypothesis tests”, in *International Conference on Artificial Intelligence in Education*, 2019, pp. 281–285. DOI: 10.1007/978-3-030-23207-8_52.

[258] G. Thorburn and G. Rowe, “PASS: An automated system for program assessment”, *Computers & Education*, vol. 29, no. 4, pp. 195–206, 1997. DOI: 10.1016/S0360-1315(97)00021-3.

[259] N. Tillmann, J. de Halleux, T. Xie, S. Gulwani, and J. Bishop, “Teaching and learning programming and software engineering via interactive gaming”, in *Conference on Software Engineering*, IEEE, 2013, pp. 1117–1126. DOI: 10.1109/ICSE.2013.6606662.

[260] N. Truong, P. Bancroft, and P. Roe, “Learning to program through the web”, *ACM SIGCSE Bulletin*, vol. 37, no. 3, pp. 9–13, 2005. DOI: 10.1145/1151954.1067452.

[261] N. Truong, P. Roe, and P. Bancroft, “Static analysis of students’ Java programs”, in *Australasian Conference on Computing Education*, vol. 30, 2004, pp. 317–325.

[262] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad", in *IEEE International Conference on Software Engineering*, 2015, pp. 403–414. DOI: 10.1109/icse.2015.59.

[263] H. Ueno, "A generalized knowledge-based approach to comprehend Pascal and C programs", *IEICE Transactions on Information and Systems*, vol. 83, no. 4, pp. 591–598, 2000.

[264] M. Ulloa, "Teaching and learning computer programming: A survey of student problems, teaching methods, and automated instructional tools", *ACM SIGCSE Bulletin*, vol. 12, no. 2, pp. 48–64, 1980. DOI: 10.1145/989253.989263.

[265] L. C. Ureel II and C. Wallace, "Automated critique of early programming antipatterns", in *SIGCSE Technical Symposium on Computer Science Education*, 2019, 738–744. DOI: 10.1145/3287324.3287463.

[266] A. K. Vail and K. E. Boyer, "Identifying effective moves in tutoring: On the refinement of dialogue act annotation schemes.", in *Intelligent Tutoring Systems*, 2014, pp. 199–209. DOI: 10.1007/978-3-319-07221-0_24.

[267] J. Van Merriënboer and M. De Croock, "Strategies for computer-based programming instruction: Program completion vs. program generation", *Journal of Educational Computing Research*, vol. 8, no. 3, pp. 365–94, 1992. DOI: 10.2190/mjdx-9pp4-kfmt-09pm.

[268] K. VanLehn, "The Behavior of Tutoring Systems", *International Journal of Artificial Intelligence in Education*, vol. 16, no. 3, pp. 227–265, 2006.

[269] K. VanLehn, "The Relative Effectiveness of Human Tutoring, Intelligent Tutoring Systems, and Other Tutoring Systems", *Educational Psychologist*, vol. 46, no. 4, pp. 197–221, 2011. DOI: 10.1080/00461520.2011.611369.

[270] P. Vanneste, K. Bertels, and B. Decker de, "The use of reverse engineering to analyse student computer programs", *Instructional Science*, vol. 24, pp. 197–221, 1996. DOI: 10.1007/bf00119977.

[271] A. Venables and L. Haywood, "Programming Students NEED Instant Feedback!", in *Australasian Conference on Computing Education*, vol. 20, 2003, pp. 267–272, ISBN: 0909925984.

[272] A. Vihavainen, J. Airaksinen, and C. Watson, "A systematic review of approaches for teaching introductory programming and their influence on success", in *ICER Conference on International Computing Education Research*, ACM, 2014, pp. 19–26. DOI: 10.1145/2632320.2632349.

[273] A. Vihavainen, M. Luukkainen, and P. Ihantola, "Analysis of source code snapshot granularity levels", in *SIGITE Conference on Information Technology Education*, 2014, pp. 21–26. DOI: 10.1145/2656450.2656473.

[274] A. Vihavainen, T. Vikberg, M. Luukkainen, and M. Pärtel, "Scaffolding students' learning using test my code", in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2013, pp. 117–122. DOI: 10.1145/2462476.2462501.

[275] A. Vizcaíno, "A Simulated Student Can Improve Collaborative Learning", *International Journal of Artificial Intelligence in Education*, vol. 15, pp. 3–40, 2005.

[276] A. Vizcaíno, J. Contreras, J. Favela, and M. Prieto, "An adaptive, collaborative environment to develop good habits in programming", in *Intelligent Tutoring Systems*, vol. LNCS 1839, 2000, pp. 262–271. DOI: 10.1007/3-540-45108-0_30.

[277] M. Vujošević-Janičić, M. Nikolić, D. Tošić, and V. Kuncak, "Software verification and graph similarity for automated evaluation of students' assignments", *Information and Software Technology*, pp. 1004 –1016, 2013. DOI: 10.1016/j.infsof.2012.12.005.

[278] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises", in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, 481–495. DOI: 10.1145/3192366.3192384.

[279] T. Wang, X. Su, P. Ma, Y. Wang, and K. Wang, "Ability-training-oriented automated assessment in introductory programming course", *Computers & Education*, vol. 56, no. 1, pp. 220–226, 2011. DOI: 10.1016/j.compedu.2010.08.003.

[280] C. Watson and F. W. Li, "Failure rates in introductory programming revisited", in *ITiCSE Conference on Innovation and Technology in Computer Science Education*, 2014, pp. 39–44. DOI: 10.1145/2591708.2591749.

[281] G. Weber, “Episodic learner modeling”, *Cognitive Science*, vol. 20, no. 2, pp. 195–236, 1996. DOI: 10.1016/S0364-0213(99)80006-8.

[282] G. Weber and P. Brusilovsky, “ELM-ART: An Adaptive Versatile System for Web-based Instruction”, *International Journal of Artificial Intelligence in Education*, vol. 12, pp. 351–384, 2001.

[283] G. Weber and M. Specht, “User Modeling and Adaptive Navigation Support in WWW-Based Tutoring Systems”, in *Conference on User Modeling*, 1997, pp. 289–300. DOI: 10.1007/978-3-7091-2670-7_30.

[284] G. M. Weinberg, *The psychology of computer programming*. Van Nostrand Reinhold New York, 1971, vol. 29.

[285] D. Weragama, “Intelligent Tutoring System for Learning PHP”, PhD thesis, Queensland University of Technology, 2013. DOI: 10.4018/978-1-61692-008-1.ch009.

[286] D. Weragama and J. Reye, “Analysing student programs in the PHP intelligent tutoring system”, *International Journal of Artificial Intelligence in Education*, vol. 24, no. 2, pp. 162–188, 2014. DOI: 10.1007/s40593-014-0014-z.

[287] R. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014. DOI: 10.1007/978-3-662-43839-8.

[288] E. S. Wiese, M. Yen, A. Chen, L. A. Santos, and A. Fox, “Teaching Students to Recognize and Implement Good Coding Style”, in *ACM Conference on Learning @ Scale*, 2017, pp. 41–50. DOI: 10.1145/3051457.3051469.

[289] W. Wu, G. Li, Y. Sun, J. Wang, and T. Lai, “AnalyseC: A framework for assessing students’ programs at structural and semantic level”, in *Conference on Control and Automation*, 2007, pp. 742–747. DOI: 10.1109/ICCA.2007.4376454.

[290] S. Xu and Y. S. Chee, “Transformation-based diagnosis of student programs for programming tutoring systems”, *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 360–384, 2003. DOI: 10.1109/TSE.2003.1191799.

[291] C. Yongqing, H. Qing, and Y. Jingyu, “An expert system for education: IPTS”, in *International Conference on Systems, Man, and Cybernetics*, 1988, pp. 930–933.

Open Universiteit
www.ou.nl