Empir Software Eng (2018) 23:2795–2828
https://doi.org/10.1007/s10664-017-9579-0

Does syntax highlighting help programming novices?

Christoph Hannebauer[1] · **Marc Hesenius**[1] · **Volker Gruhn**[1]

Published online: 28 February 2018

Abstract Program comprehension is an important skill for programmers – extending and debugging existing source code is part of the daily routine. Syntax highlighting is one of the most common tools used to support developers in understanding algorithms. However, most research in this area originates from a time when programmers used a completely different tool chain. We examined the influence of syntax highlighting on novices' ability to comprehend source code. Additional analyses cover the influence of task type and programming experience on the code comprehension ability itself and its relation to syntax highlighting. We conducted a controlled experiment with 390 undergraduate students in an introductory Java programming course. We measured the correctness with which they solved small coding tasks. Each test subject received some tasks with syntax highlighting and some without. The data provided no evidence that syntax highlighting improves novices' ability to comprehend source code. There are very few similar experiments and it is unclear as of yet which factors impact the effectiveness of syntax highlighting. One major limitation may be the types of tasks chosen for this experiment. The results suggest that syntax highlighting squanders a feedback channel from the IDE to the programmer that can be used more effectively.

Keywords Syntax highlighting · Source code typography · Code colouring · IDE interface · Program comprehension

Communicated by: Sven Apel

✉ Christoph Hannebauer
christoph@hannebauer.name

Marc Hesenius
marc.hesenius@uni-due.de

Volker Gruhn
volker.gruhn@uni-due.de

[1] paluno – The Ruhr Institute for Software Technology, University of Duisburg-Essen, Essen, Germany

1 Introduction

Software developers spend a considerable amount of time extending or debugging existing systems (Lientz et al. 1978) – tasks requiring sound program comprehension skills. Unfortunately, source code can be complicated and even professional programmers need time to process and understand what a given algorithm is actually doing (Tiarks 2011). Typical programming tasks often involve code written by others, forcing developers to work with unfamiliar variable names, unexpected code structures, and peculiar comments.

Different strategies have been developed to support program comprehension. *Coding guidelines* are often used, so code written in the same project or the same company is supposed to have an identical look and feel. Style guides and coding conventions are also issued by language maintainers, e.g. for Java (Oracle Technology Network 1999) or C# (Microsoft Developer Network 2015). Another strategy, *code indentation*, is used to indicate blocks belonging together. Additionally, *syntax highlighting* is used to visually distinguish source code elements.

Syntax highlighting is a common tool for software developers. Every major Integrated Development Environment (IDE) ships with its own colouring scheme that is activated by default but can often be customised to fit the developer's needs and personal taste. In a nutshell, syntax highlighting uses typographic signalling on parts of source code to provide visual cues about language elements and programme structure. For example, it sets keywords apart from variables or makes static method calls immediately distinguishable from regular method calls. A variety of markups is used and a mix of colouring and different text formatting styles such as italics or boldface fonts is common. That colouring plays an important role in syntax highlighting is no surprise as it builds on an important human skill. Colour vision serves two main purposes: perceptual segregation and signalling (Goldstein 1995). Both are used when working with highlighted source code. Programmers need to distinguish between different keywords, token types, predefined language elements (that typically cannot be redefined for individual purposes), and custom programming constructs, e.g. variable names. According to Goldstein (1995), colours provide a clear model of the perceived concept. In case of programming, colours clearly signal differences between various language elements and give clues to the programmer.

Source code works in two ways. On the one hand, it is a set of instructions for the computer, telling it specifically what to do. On the other hand, it serves as an explanation to human readers, describing the technical implementation of an algorithm in detail (Knuth 1984). Understanding the computer's interpretation of the source code is therefore a valuable information for developers. One way for IDEs to connect both perspectives is syntax highlighting. Gruhn and Hannebauer (2012).

However, little is known about the actual effect of syntax highlighting on programmers. Some older research on the influence of different semantical code highlighting schemes on program comprehension exists (Baecker and Marcus 1989; van Laar 1989), but only a few recent studies investigated the effects of syntax highlighting commonly used in today's IDEs (Beelders and du Plessis 2015; Sarkar 2015) and found contradicting results. We do not know whether programmers understand source code with highlighted syntactical structures better or if it is actually harmful to their program comprehension. Especially beginners might suspect that the colours and font styles have some additional meaning, but without enough experience, their purpose remains hidden or – even worse – might be misinterpreted. Despite the lack of evidence for its usefulness, different forms of syntax highlighting are omnipresent in professional as well as educational IDEs. Different cognitive processes are

at work here, and whilst it would be interesting to investigate these and the influence of syntax highlighting in detail, we will focus on the results, i.e. whether syntax highlighting supports programming novices, and leave questions regarding the how and why to future research.

We conducted an extensive study on syntax highlighting with 390 mostly first-year students enrolled in computer science or related fields. Test subjects were presented with a set of different Java code snippets in two flavours – using Syntax Highlighting (SH) versus plain Black & White (BW) text – and had to solve simple program comprehension tasks on these code snippets. We then performed statistical analyses on their results in order to examine the impact of syntax highlighting on their ability to solve the tasks correctly. The time needed to solve the tasks was not measured as previous research suggests that code highlighting has an impact on correctness (Baecker and Marcus 1989), whereas there is no statistically significant impact on speed even in experiments that showed correctness improvements through code highlighting (Oman and Cook 1990).

Contrary to our expectations and given that syntax highlighting is omnipresent in modern IDEs, the results do not support our hypotheses. Earlier research on code highlighting based on semantical code properties showed stronger positive effects on Program comprehension with at least four times smaller datasets (Baecker and Marcus 1989; Oman and Cook 1990; Rambally 1986; Tapp and Kazman 1994). We conclude that, for the types of tasks used in the experiment, programmers do not benefit from the Eclipse code highlighting used in the experiment, which is based on syntactical structures. Our findings indicate that current IDEs possibly waste a feedback channel to the developer with an ineffective code highlighting scheme. This feedback channel could convey more meaningful information, for example the font colour could encode the type of function in terms of its namespace.

Jedlitschka et al. (2008) proposed a structure for experiment reports in software engineering that this experiment report adheres to. Thus, we will first describe related technology and studies in Section 2 and then describe our experiment design in Section 3. In Section 4, we discuss the coding tasks used for the experiments and the results obtained for the individual task categories. Result analysis follows in Section 5. We discuss all results in Section 6 and look at possible threats to validity in Section 7, before concluding the paper in Section 8.

2 Background

Recently, the effect of syntax highlighting on programming performance received some research attention. Before this, a somewhat larger number of studies was conducted in the 80s and 90s – a time when programmers used a different set of tools and languages to write software. Furthermore, most of these studies concentrate on different typographic signalling methods (e.g. indentation or font size). We identified three different uses of highlighting: indicating the token type, conveying run-time information, and grouping segments of code into logical units. Also, in most older studies, source code was printed out on paper and handed to the test subjects – an unusual experimental setup from today's perspective. Working with printed source code has become very uncommon.[1]

[1]Microsoft Visual Studio and Eclipse still can print out source code, though. Interestingly, Eclipse retains syntax highlighting in the print-out and Visual Studio removes it.

2.1 Syntax highlighting

Hakala et al. (2006) performed an experiment with 16 third and fourth year computer science students and compare three different colouring schemes for Java:

- a *control scheme* without highlighting,
- a *block scheme* with blue comments, red method headers and declarations, and the remaining code in black and white, and
- a *token scheme* using the syntax highlighting for Java from the code editor vim.

There were twelve tasks, in each of which participants had to search for a target within a code block of 76 lines of code. The colouring schemes had neither an effect on the speed with which participants found the target nor an effect on the correctness with which they found the correct target. Whilst there are some threats to the validity of this experiment, there were three different types of targets and the type of target did have a statistically significant effect on speed and correctness. Thus, code colouring is at most a small help for the tasks tested in the experiment. Nevertheless, the majority of participants subjectively found colours useful for their task.

Sarkar (2015) had 10 graduate computer science students read the source code of six python programmes and determine the output for a given input value. The six python programmes came in three pairs of equal difficulty, and each participant saw one random programme of each pair with a common syntax highlighting scheme, whilst the other had no typographic highlighting. Tasks were solved significantly faster with SH and eye-tracking data showed that SH significantly reduced the number of context switches. The advantage of SH becomes significantly less with growing experience of the participants. SH had no effects on fixation counts and durations. However, the small number of participants threatens the result's validity, as the statistical significance may be an effect of possibly unfavourable task assignment.

Dimitri (2015) evaluated the effect of typographic highlighting in Sonic Pi, a simple programming language for music and subset of Ruby. In an experiment with 10 computer science students, the author showed that syntax highlighting reduces the time needed to write as well as debug a programme in Sonic Pi. The results also indicate that this effect becomes weaker when participants have more experience in programming. It is unclear how these results apply to general programming as the used programming tasks were much simpler than typical programming tasks, e.g. the code in the experiment used no conditions, method invocations, or variables.

Beelders and du Plessis (2015) performed an experiment with 34 students, in which they used eye-tracking to compare the difficulty of reading a code snippet in C# with and without syntax highlighting. The participants had to find the output value of two code snippets with given input values. The figures in the study indicate that they used the standard syntax highlighting scheme from Visual Studio for the experiment. The authors do not find significant benefits of syntax highlighting for participant's performance, measured via the number of eye fixation on different parts of the code, fixation durations, and required regressions, i.e. looking back to something already looked at. However, this may be due to a too small number of participants. The correctness or speed of the answer was not taken into account. At least with significance level $\alpha = 0.1$, there are some conditions with statistical significance where the participants subjectively found the syntax-highlighted code more readable and aesthetically more pleasing.

2.2 Highlighting schemes different to modern IDEs

This section reviews older related studies using highlighting schemes that are uncommon today. These forms of code highlighting were not necessarily based on token type.

Rambally (1986) provided test subjects (44 intermediate-level and 35 senior-level programming students) with source code formatted according to three different schemes: (a) with coloured blocks, denoting loops for example; (b) with colours for different statements and functions, e.g. input/output (I/O) or variable binding; (c) black and white plain text. He hypothesised that only scheme (b) would lead to improvements in program comprehension. The results were in favour of the hypothesis, so he concluded that the colouring supported test subjects in understanding the source code. Interestingly, he added that test subjects favoured scheme (a), although the comprehension results clearly indicated that scheme (b) was more efficient. This experiment used two different highlighting variants: scheme (a) highlights units and scheme (b) highlights different token types – an approach pretty close to syntax highlighting, but Rambally – in contrast to modern IDEs – seemed to take further information into account, e.g. by specifically colouring functions concerned with I/O. Furthermore, it is not clear what exact colouring strategy was used, it could be either background or font colouring. In any case, Rambally (1986) shows that colouring is a way to improve program comprehension and that the rules used to determine the colouring scheme impact its effectiveness.

Crosby and Stelovsky (1990) investigated how programmers read algorithms. They presented an algorithm's source code written in Pascal without highlighting and a graphical representation to novices as well as experienced software developers and evaluated whether these two groups would use different reading strategies in order to understand the programme. The authors recorded the test subjects' eye movements and tried to detect patterns whilst they examined the algorithm. They found that keywords are not the primarily investigated parts of a programme, although keywords are typically emphasised through the use of bold font. Comments and comparisons attract most of the readers' attention. This observation held true for experienced as well as novice programmers. The authors stated that keywords lack semantic information and are too predictable to be important enough to understand the algorithm, hence the little interest from readers. Highlighting keywords is an approach to indicate token types. Unfortunately, Crosby and Stelovsky (1990) did not investigate whether the bold font suggested a keyword and therefore a part of the text that does not need that much attention from readers or if the code's structure simply made a keyword necessary and obvious.

Feigenspan et al. (2013) show that changing the background colour of preprocessor directive blocks makes it easier for developers to identify code belonging to the same feature. They showed that background colouring can improve program comprehension, is favoured by programmers, and can scale to very large programmes. However, their approach is just an alternative way of marking related source code parts. For this specific use, their approach is useful, but it does not answer any question about syntax highlighting itself. They highlight units of code with background colouring.

van Laar (1989) tested the effect of coloured structures in source code. He presented Pascal programmes to 16 test subjects using a mix of coloured and non-coloured as well as indented and non-indented source code. He found significant differences in how well the subjects processed different variations and identified the coloured and indented one as the most effective. In contrast to syntax highlighting schemes of modern IDEs, van Laar (1989)

used colours in a similar way to indentations: highlighting lines of code belonging to the same code block instead of employing a syntactical colouring scheme. van Laar (1989) thus demonstrated how colours can be used, but his approach differs from the colouring scheme we focus on in our study. Again, this was an attempt to highlight units of code using different background or font colours.

Reijers et al. (2011) described their approach to transfer the notion of syntax highlighting into the field of Business Process Modelling (BPM). Modellers and programmers share a common fate: they have to constantly understand existing conceptualisations (models or source code) and align them with their own mental model. Reijers et al. (2011) therefore developed a highlighting scheme for workflow nets and conducted an experiment with 62 expert (from industry and academia) and 42 novice (students on graduate level) modellers. The experiment showed that the highlighting scheme significantly increased the accuracy of novices, helping them to read and understand the models. Of course, this study does not deal with source code, but their approach is close to highlighting token types. Nevertheless, it shows that novices can benefit from coloured elements.

Tapp and Kazman (1994) performed a 3×2 experiment with 39 test subjects. The experiment compared three different code formatting flavours for two different programming tasks. One flavour coloured the background of the code, but not its font. The second flavour highlighted code via different font sizes. The third flavour acted as a control group and involved no special formatting. The background colours or font sizes for certain lines of code provided additional information depending on the task at hand. The experiment showed significantly improved performance in one aspect for one of the tasks when using the colouring flavour. Test subjects also favoured colouring over font size. However, Tapp and Kazman (1994) did not test *syntax* highlighting. Instead, they tried to convey information about the programme's behaviour at run-time via background colouring or font size.

Gellenbeck and Cook (1991) presented a rather small experiment with only eight test subjects. In contrast to other studies, they focused on professional programmers having more than five years of experience. They marked module names and header comments with typographic signalling in the form of a larger and boldface font compared to the rest of the source code, which was then printed out and handed to the subjects. Besides the signalling, they tested for the usefulness of mnemonic names and header comments. They conclude that mnemonic names and header comments are more important to program comprehension than signalling and also that signalling does not aid programmers in locating information within a programme. Although they highlighted different token types with different font formatting, their experiment was restricted to two types of tokens and did not cover the whole source code.

Baecker and Marcus (1989) presented various highlighting schemes for programmes written in C. They discussed several aspects of source code visualisation such as font types and sizes, bar positions and sizes, whitespace distribution, brackets and boxes for structuring, and background highlighting. Font colouring is only a minor topic. They also discussed which source code properties should determine the visualisation parameters. Whilst they focused on source code print-outs, they also explained possible adaptations of the print-out format to on-screen displays. Baecker and Marcus (1989) validated their proposed format in an experiment with 44 third-year students. They found improvements of program comprehension through the proposed format, and tested them with statistically significant result, although this depended on the specific programme. This was an extensive study with different alternatives, all involving different highlighting for token types. However, they mainly used font formatting for signalling purposes and, in contrast to recent IDEs, did not take colouring into account.

Oman and Cook (1990) analysed the influence of macro- and micro-typography on program comprehension. They proposed a book-like arrangement of source code, including e.g. a table of contents, and called this part the macro-typography. In addition, they propose micro-typographic changes to the presentation of the source code: This primarily prescribes how to use indentation and empty lines, but also boldface fonts for function calls. They validate the proposed format in three different experiments; the one relevant for our study tests the micro-typographic enhancements. 36 intermediate and 44 advanced computer science students participated and C as well as Pascal were used. The results showed statistically significant benefits when using the proposed micro-typographic format. Oman and Cook (1990) mixed unit and token type highlighting in the micro-typography and focused on unit highlighting in the macro-typography.

All these studies have in common that they tested different typographic signalling mechanisms and aimed for highlighting different information. Some of them found significant evidence that the proposed highlighting schemes were useful to experts as well as novices. However, the implementation of syntax highlighting in recent IDEs differs from the schemes described above – syntax highlighting typically only indicates different token types, without providing any additional or more sophisticated information. Eclipse in particular uses a rather minimalistic colouring scheme by default compared to other IDEs.

2.3 Summary

In summary, research gives some hints that syntax highlighting in its current form may be useful at least for some programming tasks, but the results are still inconclusive. Positive outcomes are based on small scale experiments and sometimes treat special cases not representative for general programming tasks. Other experiments could not show that syntax highlighting helps with any programming tasks. Even in experiments where syntax highlighting was of no help, programmers mostly liked the use of colours in code editors. In particular, Rambally (1986) showed that programmers favour suboptimal highlighting schemes sometimes.

3 Experiment planning

We describe our experimental set up in the following subsections. We take a closer look at the test group, examine the different tasks they had to solve, and what environment was used.

3.1 Experiment goals

With the results from the experiments, we wanted to analyse four research questions (henceforth referred to as RQ 1-4). The first targets the general effect of syntax highlighting:

Research Question 1 Does syntax highlighting support program comprehension and therefore have a positive effect on the programming ability of novice programmers?

Syntax highlighting colourises tokens based on their token type. There are various data sources other than the token type on which the typography of the displayed source code may depend, like programme run-time information (Tapp and Kazman 1994) or unit types (Rambally 1986; Oman and Cook 1990). Section 2 discusses this in more detail. The type of information most helpful to programmers depends on what they want to understand when

reading source code. For example, they may want to know the output of a given code snippet, its algorithmic complexity or which other modules the given code snippet depends on. This yields the next research question:

Research Question 2 How does the extent of the positive effect of syntax highlighting depend on the type of task to be solved?

Long experience with a certain Integrated Development Environment (IDE) makes developers accustomed to its way of syntax highlighting. Consequently, using the accustomed syntax highlighting scheme on source code may result in better and faster program comprehension. However, experienced developers may easily recognise different token types without syntax highlighting, thus the benefits of syntax highlighting are lost. We test which of these two forces has a stronger effect and ask:

Research Question 3 Does the positive effect on their programming ability increase when programmers familiarise themselves with the meanings of their Integrated Development Environment (IDE)'s syntax highlighting rules?

Grant and Sackman (1967) as well as Sackman et al. (1968) found high performance differences of 25 to 1 between skilled and unskilled programmers. However, Prechelt (1999) showed that the original figures of 25 to 1 were based on methodological problems and the interpersonal variation is in fact only between 2 to 1 and 4 to 1. Still, there are differences and existing research analysed what factors determine these differences (Bergin and Reilly 2005; Holden and Weeden 2003). The statistical impact of self-reported programming experience and programming performance yielded some contradictory results (Bergin and Reilly 2005; Kleinschmager and Hanenberg 2011). This leads to another research question, this time not related to syntax highlighting:

Research Question 4 Does previous programming experience increase the ability of students to correctly solve short programming-related tasks?

RQ 4 allowed us to evaluate the validity of the self-reported data. Furthermore it let us evaluate whether the tasks we used in our experiments actually require programming skills to solve them correctly.

3.2 Experiment participation

Test subjects of our study were students enrolled in computer science and related fields attending a course on *Programming in Java*. All students in the course could participate in the experiment as a preparation for their final course exams. They were informed that this preparation is also an experiment, but not about the specific nature of the experiment to make it a blind experiment. They also gave their consent for the scientific usage of the results. The test group consisted mainly of first year undergraduate students with little or no programming experience as well as several more experienced students. The experiment comprised three points of measurement, each of which involved a different set of tasks. In total, 390 students participated in at least one point of measurement.

The study data were drawn from three preparatory pre-exam tests interspersed throughout the semester, henceforth referred to as Point of Measurement *A, B,* and *C*. The timeline and exact number of test subjects per point of measurement is presented in Table 1. 79 test subjects participated in only one point of measurement, 209 test subjects participated

Table 1 Distribution of tasks and test subjects per point of measurement

Point of measurement	A	B	C
Number of tasks	5	8	7
Number of participants	376	319	107
Date of measurement	2013-05-14	2013-06-04	2013-07-09

in two points of measurement, and 102 test subjects participated in all three points of measurement.

The experiments were conducted in form of one-hour-long electronic examinations using the LPLUS examination system (LPLUS GmbH 2014) that were automatically evaluated. Students got the opportunity to get acquainted with the examination system during a separate pilot session before performing any real experiment. The pilot session also provided us with some feedback on task design as well as examination system handling.

During experiments, test subjects were located at isolated PCs and presented with tasks on a computer screen with mouse and keyboard available. Tasks were either knowledge questions or source code tasks. The knowledge questions asked for definitions or tested understanding of object-oriented concepts and were not factored into the study results as they are not affected by syntax highlighting. These knowledge questions were included for didactical reasons, as the experiments also served as preparation for the course exams. Source code tasks asked the test subjects to solve problems based on source code snippets.

3.3 Programming experience

243 students also completed a form about their programming experience. This form did not work in Point of Measurement A due to a technical issue, therefore data from this form are not available for all participants.

Figure 1 shows the distribution of answers regarding programming experience. We asked for experience with programming in general and for experience with Java in particular, since we used Java in the experiment. We operationalised programming experience via the self-reported time since the participants started programming in general or with Java. We chose this measure for its objectiveness, and only later found out that other measures might have been more suitable (Siegmund et al. 2014).

Other demographic data like the participants' gender and age were not collected.

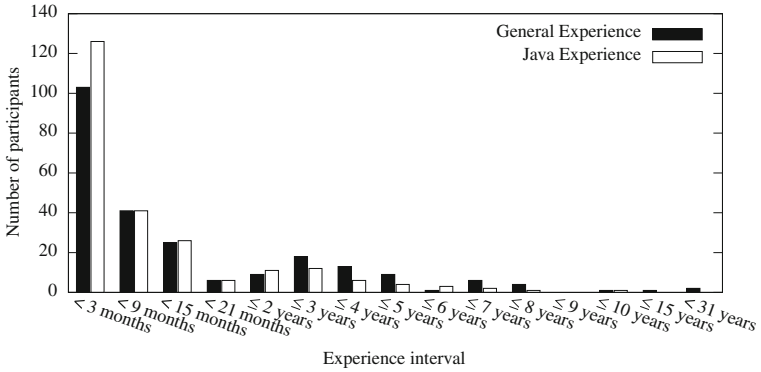


Fig. 1 Histogram showing the programming experience of the participants

3.4 Source code tasks

Every source code task existed in two flavours of typographic style: either with syntax highlighting using Eclipse's basic Java scheme (SH) or merely in plain black & white text (BW). There was no difference in terms of content. Both flavours used the Courier New font, the default in the Eclipse Integrated Development Environment (IDE) on Windows. Test subjects worked on each task either in SH flavour or BW flavour, but never on both. However, different tasks could have different flavours for the same test subject.

The order in which the tasks were presented was randomised for each test subject. For single and multiple choice tasks, the order of possible answers was also randomised for each test subject. This ensured that results were not biassed by the order in which the tasks or answers were presented.

All 20 source code tasks were used to conduct our study. The distribution over points of measurement is presented in Table 1. A task from Point of Measurement A was reused in Point of Measurement C, but treated as separate (henceforth denoted as tasks 3 and 3b).

The tasks presented to test subjects belonged to different categories summarised in Table 2, each of which was processed in a different way. Didactical considerations influenced which tasks occurred in an experiment, therefore some task categories occur more often than others. Table 2 also lists how many Lines of Code (LOC) the tasks in each category had at least and at most. A detailed analysis of the task categories follows in Section 4.

The tasks are numbered, using the following scheme: Tasks 1 (OUTP), 2 (FTG), 3 (UML), 6 (FSE), and 7 (FTGUML) belong to Point of Measurement A. Point of Measurement B comprises the Tasks 8 (OUTP), 9 (OUTP), 10 (UML), 11 (ARR), 12 (FTGUML), 13 (OUTP), 14 (FSE), and 15 (FTGUML). The remaining tasks are Tasks 16 (OUTP), 17 (ARR), 3b (UML), 18 (FTGUML), 19 (FTGUML), 21 (ARR), and 22 (FSE) and belong to Point of Measurement C.

3.5 Collected variables

In the experiment, we collected the variables shown in Table 3 for each test subject. TASK-SCORE-N, TASK-CORRECT-N, and TASK-FLAVOUR-N are sets of variables, they exist

Table 2 Task categories

Code	Task	Description	No. of tasks	(LOC)
OUTP	Determine Output	Determine the output of a given code snippet	5	8-25
FSE	Find syntactical errors	Name all lines of code that cannot be compiled	3	27-37
FTG	Fill in the gaps	Fill the gaps in code snippets with given answers	1	23
UML	Find coding mistakes	Find all mismatches of code snippets with given UML diagrams	3	27-29
FTGUML	Fill in the gaps (UML)	Fill the gaps in code snippets with given answers to match a given UML diagram	5	11-37
ARR	Arrange code snippets	Arrange given code snippets in correct sequence	3	11-20

Table 3 Collected variables

Name	Description	Value range
TASK-SCORE-N	For each task N that the test subjected worked on, the degree to which the test subject solved the task correctly	[0; 1]
TASK-CORRECT-N	For each task N that the test subjected worked on, a boolean value indicating whether TASK-SCORE-N reaches a task-specific threshold so that the task is considered correctly solved	true or false
TASK-FLAVOUR-N	For each task N that the test subject worked on, the flavour in which the task was presented to the test subject	SH or BW
TOTAL-SCORE	The arithmetic mean of all task scores	[0; 1]
DR	A difference in proportions, measuring the impact of SH on the participant. The sign shows whether SH had positive or negative impact, whilst the absolute value indicates its strength.	[−1; 1]
JAVA-XP	Months of experience programming in Java	Non-negative rational number
GENERAL-XP	Months of experience programming in any programming language	Non-negative rational number

for every combination of test subject and task that the test subject worked on. The other variables exist only once for each test subject.

For the analysis of RQ 1, we use the DR and, as a means of method triangulation, an accumulation analysis based on TASK-CORRECT-N and TASK-FLAVOUR-N. TASK-CORRECT-N and TASK-FLAVOUR-N are the foundation for the analysis of RQ 2. For RQ 3, we use DR, JAVA-XP, and GENERAL-XP. Finally, RQ 4 uses TOTAL-SCORE, JAVA-XP, and GENERAL-XP.

The variables JAVA-XP and GENERAL-XP are described in Section 3.3. The variables TASK-CORRECT-N and DR will now be defined more precisely.

3.5.1 TASK-CORRECT-N: Correctly solved tasks

Some tasks can only be solved correctly or incorrectly. Other tasks can be partially correct, but only with a few discrete values between completely correct and completely incorrect. To treat tasks uniformly and because we know of no statistical method with enough statistical power to handle partial values when analysing individual tasks, we use a threshold to separate correct from incorrect solutions for tasks that have partially correct solutions. For example, a trivial threshold would treat all partially correct solutions as correct and only solutions with no correct parts as incorrect. Ideally, the number of correct and incorrect answers should be balanced: If there are very few correct or incorrect answers, the results are more prone to measurement errors. Therefore, every task that allows partially correct solutions has an individual threshold that depends on the difficulty of the task. With respect

to this threshold, all tasks will be treated as having solutions that are distinctly correct or incorrect.

3.5.2 DR: A measure of sensitivity to syntax highlighting

We assigned a score from 0 to 1 for each task and test subject that worked on this task (TASK-SCORE-N). We used the results for tests with higher statistical power, although they cannot be used for individual tasks or task categories.

The following paragraphs define a possible measure for the effect of syntax highlighting on a test subject – the difference in proportions (DR) – and explain its rationale. The *DR* is adapted from similar types of statistical experiments, where proportions are called risks (Ludbrook 2008). First, we will present a simple and straightforward definition of the *DR*. Later, we refine this definition to avoid a specific bias induced by the way the experiment was designed.

A simple definition $\tilde{DR}$ of the *DR* is just the difference between the fraction of correct solutions of tasks with SH and tasks with BW. For example, if a test subject solved three of four tasks with SH completely correct and failed at the fourth task, this test subject would have solved 75% of the SH tasks correctly. If the test subject likewise solved 30% of tasks in BW correctly, the $\tilde{DR}$ for this test subject would be $0.75 - 0.30 = 0.45$, indicating that SH had a positive influence on the test subject.

The experiment design guarantees that every test subject faces at least one task with SH and at least one task in BW. Thus, the $\tilde{DR}$ is well-defined for every test subject. A $\tilde{DR}$ close to 0 indicates that the test subject is insensitive to SH, whilst absolute values for $\left|\tilde{DR}\right|$ close to 1 indicate that a test subject is very sensitive to SH. The sign of $\tilde{DR}$ shows whether the effect of SH is positive or negative.

The tasks' difficulties vary independently of the presence of SH. For example, assume that for one test subject, the set of tasks in the BW flavour are on average more difficult than the ones in the SH flavour. Even if the test subject was completely insensitive to SH, the probability of a $\tilde{DR} > 0$ would be greater than the probability of a $\tilde{DR} < 0$. Since every task had a different fraction of test subjects with SH, this error may accumulate to a systematic bias for the $\tilde{DR}$ when looking at greater numbers of test subjects. To counter this systematic bias, we used a measure *DR* derived from $\tilde{DR}$, that contains an additional correction factor.

For the calculation of the *DR*, tasks gain different weights, depending on the relation of number of test subjects working on the task in the SH flavour to the number of test subjects working on the task in the BW flavour. These weights shall ensure that both flavours receive the same share of influence from every task.

Let T be the set of tasks and S be the set of test subjects. For every test subject $s \in S$, let ${}_sT_{SH}, {}_sT_{BW} \subset T$ be the set of tasks that s worked on in the SH and BW flavour, respectively. For every task $t \in T$, the weight $f_t \in (0; 1)$ shall be defined as

$$f_t := \frac{|\{s \in S | t \in {}_sT_{SH}\}|}{|\{s \in S | t \in {}_sT_{BW}\}|}$$

For example 223 test subjects were presented Task 1 with SH, whilst 153 test subjects worked on Task 1 in the BW flavour. Thus, we have $f_{Task1} = \frac{223}{153} = 1.4575$. For those test subjects who worked on Task 1 in BW, the task will count as 1.4575 tasks for the calculation of the *DR*, so the (unknown) difficulty of Task 1 has the same impact on SH tasks as it has on BW tasks.

Let ${}_sc : T \to [0; 1]$ be the function that says how much test subject s solved of each task correctly. For example, ${}_sc(\text{Task 1}) = 1$ means that s solved Task 1 absolutely correct, whilst ${}_sc(\text{Task 1}) = 0.5$ means that s reached only half of the score of Task 1. Based on the definition of $\tilde{DR}$ and the weight f_t, the DR for the test subject s, denoted as ${}_sDR$, shall be defined as

$$ {}_sDR := \frac{\sum_{t \in {}_sT_{SH}} {}_sc\,(t)}{|{}_sT_{SH}|} - \frac{\sum_{t \in {}_sT_{BW}} f_t \cdot {}_sc\,(t)}{\sum_{t \in {}_sT_{BW}} f_t} \tag{1} $$

3.6 Experiment design

Test subjects participating in a particular point of measurement worked on all tasks of that point of measurement. Each test subject was presented with some tasks in SH and some others in BW. The flavour was assigned randomly for each test subject and task, but every test subject received the same number of tasks in SH as all other test subjects of the same point of measurement. Therefore all test subjects also worked on the same number of tasks in BW. For example, in Point of Measurement A, all participants worked on the same five tasks, of which three were in SH and two in BW for each test subject, but which ones were in SH varied randomly between test subjects. In the first two points of measurement, test subjects would receive slightly more SH than BW tasks.

3.6.1 Analysis per task type: RQ 2

The analyses for RQ 2 are separate for each task. They compare test subjects with the task in SH flavour against those with BW flavour. Thus, this is a series of post-test two-group randomised experiments.

A task can be modelled with two probabilities $\pi_{SH}, \pi_{BW} \in [0; 1]$. π_{SH} and π_{BW} represent the probability that a randomly chosen test subject will solve the task correctly if the source code is presented with SH or in BW, respectively. This is the probability that TASK-CORRECT-N is true depending on TASK-FLAVOUR-N. The probabilities π_{SH} and π_{BW} depend only on the task but not on a specific test subject, as test subjects are selected randomly within a given set.

If the answer to RQ 1 is positive, test subjects should perform better in a task if the source code is presented with SH as opposed to presentation in BW. Expressed as a formula, this means $\pi_{SH} > \pi_{BW}$. Accordingly, the null hypothesis in this case is $\pi_{SH} \leq \pi_{BW}$.

Every individual task represents an experiment for RQ 1 in itself. The solutions for each flavour can be modelled as binomial distributions and the results can be displayed in a contingency table such as Table 4.

Given a contingency table of this type, the following three statistical tests decide whether the null hypothesis can be rejected: Pearson's χ^2-statistic, Barnard's Exact Test, and Fisher's Exact Test. Fisher's Exact Test does not apply to the type of experiment in this paper, although it is often incorrectly used for these types of statistical problems. In order to use

Table 4 2×2 contingency table showing the solutions of one task

	SH	BW
Number of correct solutions	a	b
Number of incorrect solutions	c	d

Fisher's Exact Test, $a + b$ and $c + d$ would have to be fixed. The popular Pearson's χ^2-statistic is only an approximation and therefore less accurate than Barnard's Exact Test. Hence, we use Barnard's Exact Test to test whether the null hypothesis can be rejected. Ludbrook (2008).

For the calculation of Barnard's Exact Test, we adapted the Comprehensive R Archive Network (CRAN) package Barnard (Erguler 2013) in the statistical software R. The adaptation (Hannebauer et al. 2017) allowed differentiation between left-tailed and right-tailed tests. The original package automatically chooses whether to use a left-tailed or right-tailed test, which sometimes yields unintended results.

First, every task in all three points of measurements was analysed separately. Thus, every task defined its own contingency table. Using Barnard's Exact Test, every task is assigned a p-value. This p-value equals the probability of seeing such a result, given that the null hypothesis is true, i. e. SH was no benefit for the test subjects working on this task.

3.6.2 Calculating combined p-values

Given the number of tables involved, only very low p-values are evidence of effects for a specific task. The actual p-values calculated for the contingency tables are not low enough to allow this deduction. Additionally, the individual contingency tables do not map directly to one of the four research questions formulated in Section 1. Instead, the p-values should be treated as intermediate calculation steps that need further treatment to answer the research questions.

However, difficulty may differ even for tasks in the same category. A task's difficulty is hard to predict in advance, thus finding the right balance is tricky. This prevents the use of a contingency table adding up solutions for multiple tasks. Instead, multiple tasks must be seen as multiple independent experiments. Therefore we combined the p-values for the individual tasks using Wallis's method of calculating the combined p-value for a set of experiments (Wallis 1942).

4 Results

The following sections for the individual task categories contain the contingency tables for all respective tasks along with the corresponding p-values. For reasons of brevity, the sums are omitted. The tasks use an internal numbering scheme invisible to test subjects, since task order was randomised. Tasks 1 to 7 belong to Point of Measurement A, Tasks 8 to 15 to Point of Measurement B, and Tasks 16 to 22 and 3b to Point of Measurement C. Detailed data are part of the downloadable lab package (Hannebauer et al. 2017).

4.1 Task category OUTP ('determine output')

The *OUTP* ('determine output') category presented code snippets for which test subjects had to determine the correct output. The answers were either selectable via multiple choice or to be entered in free text fields. Figure 2 shows a screenshot of Task 13 as an example. Test subjects could take notes to find a solution.

Understanding code occurs on different layers of abstraction. The lowest layer is understanding how a piece of code transforms an input into its output. Programmers may perform something like a symbolic execution in their minds to get an abstract idea of this transformation. This task does not ask for the abstract transformation, but for the

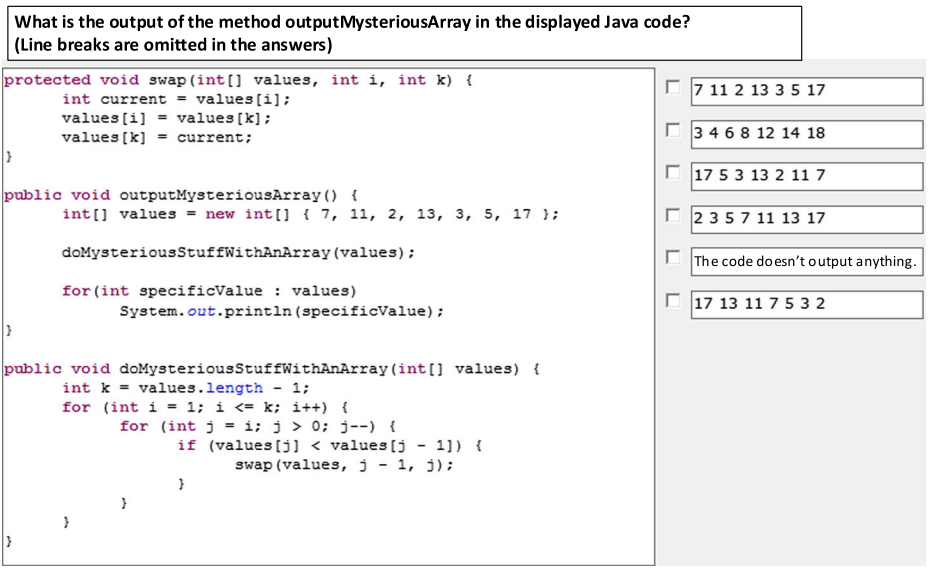


Fig. 2 Screenshot of Task 13 with SH, with English instruction and answer replacing the German original used in the experiment

transformation of a specific input. This can be easier for some tasks, but the example of Task 13 as displayed in Fig. 2 shows that understanding the abstract algorithm is sometimes easier: The code bubble-sorts an array of seven elements, which is tedious to do with pen and paper, but the output is obvious once a test subject understands that `doMysteriousStuffWithAnArray()` implements a sorting algorithm. Therefore we believe that this task category represents a common program comprehension task of a low level of abstraction. This type of task may also occur in debugging contexts where a programmer has to find out why a piece of code has an incorrect output for some input value.

How could syntax highlighting help participants solve this task correctly? Syntax highlighting may help the programmer to find the important parts of the code where the algorithm is implemented and distinguish it from the scaffolding that the programming language requires for a syntactically correct programme. Colours help with this perceptual segregation (Goldstein 1995). Participants facing the SH variant may focus their attention on determining the output of the programme, whilst participants with the BW variant may need to spend some of their mental resources on the mere comprehension of the source code structure whilst determining the output.

Table 5 lists the contingency tables for the five tasks in the category OUTP. The one-sided Barnard's tests do not show statistically significant advantages of the SH variant, except for Task 13, where it is slightly statistically significant. The combined p-value is 0.2814, so this task category generally seems not be influenced by the presence or absence of SH.

4.2 Task category FSE ('find syntactical errors')

For tasks in the category *FSE* ('find syntactical errors'), test subjects had to locate syntactical errors in a given code snippet that would prevent the code from compiling. There was

Table 5 Contingency tables of the tasks in category OUTP

Task 1	SH	BW
Correct	194	133
Incorrect	29	20
$p = 0.5311$		
Task 8	SH	BW
Correct	16	13
Incorrect	198	92
$p = 1$		
Task 9	SH	BW
Correct	28	12
Incorrect	185	94
$p = 0.3956$		
Task 13	SH	BW
Correct	112	48
Incorrect	99	60
$p = 0.0808$		
Task 16	SH	BW
Correct	18	12
Incorrect	37	40
$p = 0.142$		

no need to state the exact error – rather, test subjects were asked for the line numbers of the defective lines. Figure 3 shows Task 14 as an example of FSE.

This task category asks the test subjects to solve a common programming error (Seo et al. 2014): syntactic defects typically detected by a compiler. In the example shown in Fig. 3, the compiler would complain about the use of an undefined variable 'i' in line 7 and an unexpected 'do' in line 21. Syntax highlighting visualises the syntactic representation the compiler has about the code, so tasks in this category are the most likely to benefit from syntax highlighting.

Table 6 shows the contingency tables for the three tasks in this category. Only for one task, SH has a slightly statistically significant advantage. The combined p-value is 0.0925, which is also slightly statistically significant. There are seven task categories, so finding a result significant at the significance level $\alpha = 0.1$ for one of them might also be a statistical artefact. Also, considering the high number of participants, the effect of SH must be quite weak if it exists.

Although SH might bring a weak benefit for this kind of task, it is still an intricate problem to search for syntax errors manually. Running the compiler would yield exactly the lines with syntax errors that are searched for in this type of task in fractions of a second. Thus, this specific kind of task might be unnecessary in real programming.

4.3 Task category FTG ('fill in the gaps')

FTG ('fill in the gaps') tasks presented test subjects with code snippets containing different gaps. The aim was to fill the gaps with given answers, leading to a useful programme.

```
1  public class AdaptiveInsertionSort extends InsertionSort {
2       public void sort(char[] values) {
3
4            int high = values.length - 1;
5
6            // place smallest element at index 0
7            for (i = high; i >= 1; i--) {
8                 if (values[i] < values[i - 1]) {
9                      this.swap(values, i - 1, i);
10                }
11           }
12
13           // optimized Insertion Sort
14           for (int k = 2; k <= high; k++) {
15                this.shift(values, k);
16           }
17      }
18
19      private void shift(char[] values, int i) {
20           char x = values[i];
21           do (values[i - 1] > x) {
22                values[i] = values[i - 1];
23                i--;
24           }
25           values[i] = x;
26      }
27 }
```

Fig. 3 Example code snippet containing two syntactical errors of FSE task 14 in SH flavour. Test subjects were provided with a text box for each of the two syntactical errors

Given answer snippets could be dragged around with the mouse and dropped into the gaps. Figure 4 shows a screenshot of the only task in this category in the BW variant.

This type of task asked for a high-level understanding of class inheritance and polymorphism. Syntax highlighting could help to find the code parts relevant for the programme – strings are highlighted and the question asks for an output of 'B'. However, as the contingency table in Table 7 shows, the results are not even in favour of syntax highlighting. Possibly, this was due to the small size of the programme snippets.

Table 6 Contingency tables of the tasks in category FSE

Task 6	SH	BW
Correct	123	68
Incorrect	106	79
$p = 0.091$		
Task 14	SH	BW
Correct	139	69
Incorrect	72	39
$p = 0.4324$		
Task 22	SH	BW
Correct	34	25
Incorrect	21	27
$p = 0.1109$		

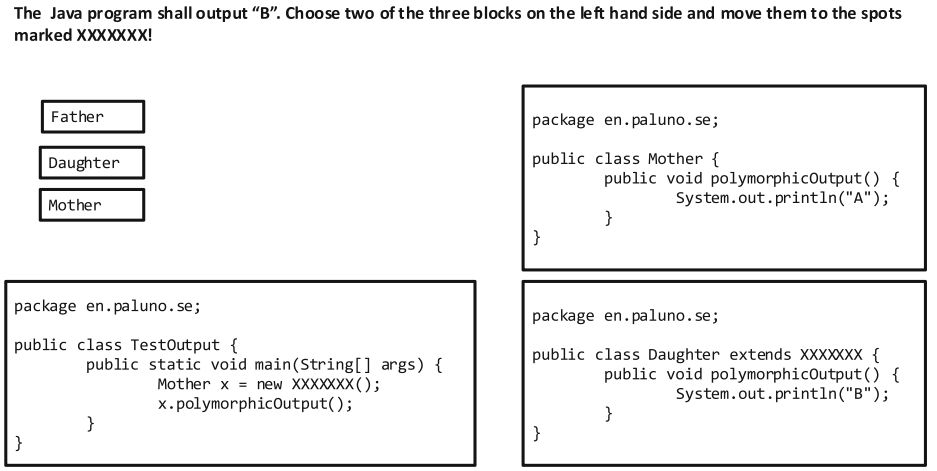


Fig. 4 Reproduction of Task 2 in BW in English. The original uses German identifiers and instructions

4.4 Task category UML ('find coding mistakes')

In *UML* ('find coding mistakes') tasks, a source code snippet was accompanied by a UML class diagram. The aim was to identify how the code snippet differed from the diagram. The diagram was always to be considered correct, so test subjects had to find mismatches within the code. Figure 5 shows a screenshot of Task 3, where test subjects could select the true statements from the following list (statements were originally in German and translated for this list):

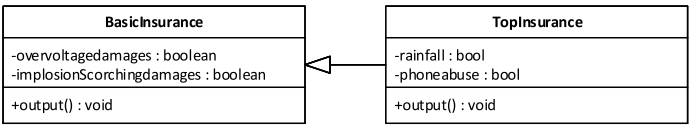
- The order of inheritance must be reversed. `TopInsurance` must inherit from `BasicInsurance` and not the other way around.
- Class attributes, for example `overvoltagedamages` in `BasicInsurance`, must be declared public, as access from outside is not possible otherwise.
- Attributes like `overvoltagedamages` in method `output()` must be converted to strings with the method `toString()` before output.
- Method `output()` must be marked public.
- Attributes `rainfall` and `phoneabuse` must be redefined in class `BasicInsurance`.
- Method `output()` must have return type `String`.

The different elements of the UML class diagram correspond to elements in the Java code. Programmers may know which kind of token they are searching for. In this case, they may restrict their search to those elements having the correct highlighting for the searched token type if they work on the SH variant. For example, the UML diagram specifies whether

Table 7 Contingency table of the task in category FTG

Task 2	SH	BW
Correct	145	102
Incorrect	76	53
$p = 0.9985$		

The UML diagram below and its implementation in Java are given. The Java implementation contains two content mistakes, though, that contradict the UML specification. Please select the statements that describe a mistake in the Java program in the sense of differences to the UML class diagram.




```
package en.paluno.se.training4.task1;

public class BasicInsurance extends TopInsurance {
    private boolean overvoltagedamages = true;
    private boolean implosionScorchingdamages = true;

    private void output() {
        System.out.println("Features of Basic Insurance:");
        System.out.println("Damages from overvoltage: " + this.overvoltagedamages);
        System.out.println("Damages from implosions and scorchings: " + this.implosionScorchingdamages);
        super.output();
        System.out.println("");
    }
}
```

```
package en.paluno.se.training4.task1;

public class TopInsurance {
    private boolean rainfall = true;
    private boolean phoneabuse = true;

    private void output() {
        System.out.println("Features of Top Insurance:");
        System.out.println("Rainfall: " + this.rainfall);
        System.out.println("Phone abuse: " + this.phoneabuse);
        System.out.println("");
    }
}
```

Fig. 5 Reproduction of Task 3 in SH. The original version had German instructions and identifiers

attributes and methods should be declared public or private. This is specified via keywords in the Java code and so a programmer may search for keywords when checking whether accessibility had been defined correctly in the code.

Table 8 shows the contingency tables for the three tasks in this category. Task 3 originally appeared in point of measurement A and was reused in point of measurement C as Task 3b. None of these tasks showed a statistically significant advantage of SH over BW. The combined p-value of 0.3339 is also not statistically significant.

4.5 Task category FTGUML (‘fill in the gaps (UML)’)

The *FTGUML* (‘fill in the gaps (UML)’) category combines the previous tasks FTG and UML. Test subjects were again provided with a code snippet and a UML class diagram and had to match both by filling in the gaps with given answers. The gaps were either within the code or the diagram, depending on the task. Figure 6 shows a screenshot of Task 15 with SH.

The code to be inserted consisted of single words each. The words did not differ in terms of highlighting, so a possible advantage of SH would be limited to the main body of code. Here again, the highlighting might make it easier to map elements of the UML diagram to parts of the code. For example, the interface keyword may show up as UML stereotype, other keywords use different symbols, whilst class identifiers show up as the same text in the diagram. Table 9 shows the contingency tables for the five tasks in category

Table 8 Contingency tables of the tasks in category UML

Task 3	SH	BW
Correct	114	67
Incorrect	114	81
$p = 0.2463$		
Task 10	SH	BW
Correct	84	51
Incorrect	130	54
$p = 1$		
Task 3b	SH	BW
Correct	33	25
Incorrect	22	27
$p = 0.1315$		

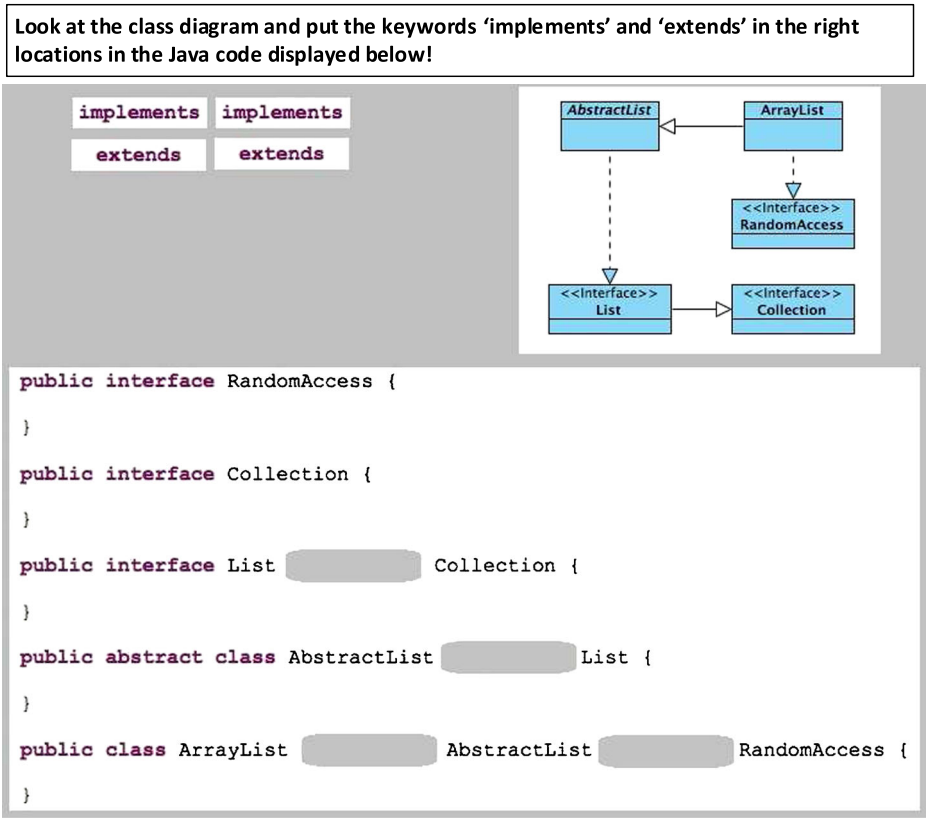


Fig. 6 Screenshot of Task 15 in SH with English instructions translated from German. Test subjects could drag and drop the four keywords into the boxes in the code

FTGUML, but no statistically significant effect in favour of SH results from Barnard's tests of the contingency tables. The combined p-value of 0.9412 by Wallis's method is also not statistically significant.

4.6 Task category ARR ('arrange code snippets')

Finally, the *ARR* ('arrange code snippets') category provided test subjects with different lines of code and an algorithm description. The given snippets had to be dragged and dropped into the right order to produce a meaningful programme according to the given description. Figure 7 shows a screenshot of Task 21, which was the only task in category ARR.

This type of task may require test subjects to look back and forth between the task description and the code to compare which line of code corresponds to which part of the task description. Beelders and du Plessis (2015) characterised these regressions as an indicator for the difficulty to comprehend code. Syntax Highlighting might help to find the right spot in the code that maps to a part of the task instruction as opposed to keywords that do not map directly to task instructions, but are merely scaffolding. Thus, syntax highlighted code might reduce the cognitive effort for this mapping and therefore ease the task of comparison.

Table 10 shows the contingency tables for the three ARR tasks. SH did not show a positive effect for our test subjects. Consequently, the combined p-value of 1 for the three tasks is not statistically significant.

Table 9 Contingency tables of the tasks in category FTGUML

Task 7	SH	BW
Correct	139	91
Incorrect	88	58
$p = 0.5011$		
Task 12	SH	BW
Correct	94	57
Incorrect	117	51
$p = 1$		
Task 15	SH	BW
Correct	141	83
Incorrect	72	23
$p = 1$		
Task 18	SH	BW
Correct	36	29
Incorrect	19	23
$p = 0.253$		
Task 19	SH	BW
Correct	25	29
Incorrect	27	26
$p = 0.9995$		

The fast exponentiation method *Square & Multiply* reduces the number of required multiplications. For example, $y = x^4$ can be calculated as $y = x \cdot x \cdot x \cdot x$ (3 multiplications) or as $z = x \cdot x$ and $y = z \cdot z$ (2 multiplications), thus $y = \left(x^2\right)^2$. The method founds on the observation that for all exponents $n \geq 0$, the following holds true:

$$x^n = \begin{cases} 1 & \text{if } n = 0, \\ x & \text{if } n = 1, \\ x \cdot \left(x^2\right)^{(n-1)/2} & \text{if } n \text{ is odd, and} \\ \left(x^2\right)^{n/2} & \text{if } n \text{ is even.} \end{cases}$$

This can be implemented as a recursive function. Insert the following code lines in the code shown at the bottom.

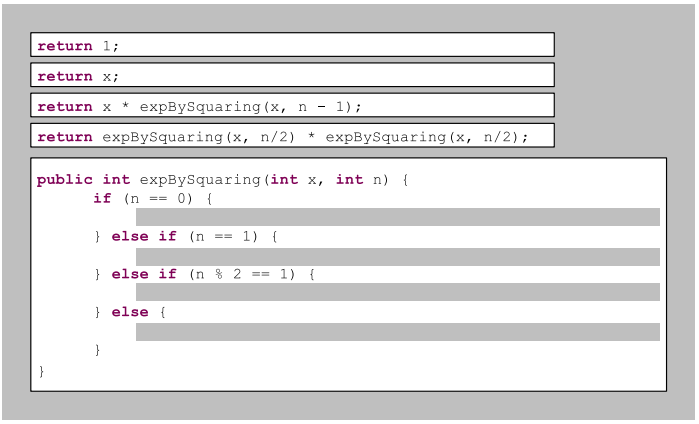


Fig. 7 Screenshot of Task 21 in SH and English reproduction of the originally German task instructions

5 Analysis

This section describes the outcomes of the statistical tests used for the four research questions.

Table 10 Contingency tables of the tasks in category ARR

Task 11	SH	BW
Correct	35	79
Incorrect	73	132
$p = 0.9827$		
Task 17	SH	BW
Correct	41	45
Incorrect	11	10
$p = 1$		
Task 21	SH	BW
Correct	33	40
Incorrect	19	15
$p = 1$		

5.1 RQ 1: overall effect

The *DR* is a measure of a test subject's sensitivity to syntax highlighting, where positive values indicate a positive effect of SH and negative values indicate a negative effect. The distribution of *DR* values for the 390 test subjects is shown in Fig. 8.

If SH has a positive effect on the results, the mean *DR* should be positive. Indeed, the mean is 0.00558, but as this is very close to 0.0, the effect can only be very weak. Figure 8 shows that the distribution looks like a normal distribution, so a parametric test could show whether this difference in mean is statistically significant.

However, a Shapiro-Wilk Test finds statistically significant differences between the sample and the normal distribution ($p < 0.001$). As the Shapiro-Wilk test is very sensitive to sample sizes, these differences may not be a problem for using the t-test. High skewness is a major problem for t-tests (Chaffin and Rhiel 1993), so we analysed the skewness of the sample. The skewness of the DR distribution in the sample is 0.416. Extrapolating the data from Chaffin and Rhiel (1993) suggests that with the sample size of $n = 390$, the skewness is low enough to allow a one-tailed one-sample t-test.

A one-tailed one-sample t-test does not show that the *DR* is statistically significantly greater than 0 ($p = 0.3458$). Hence the dataset provides no evidence that RQ 1 has a positive answer. If there is an effect, it is too small to be recognised in the dataset.

As Dragicevic (2016) suggests, confidence intervals are a method to find out whether the effect is just very weak or the test power is just too low to measure it. In order to find an effect size for SH, we calculated the 95% confidence interval for the mean, which is the DR interval $[-0.022; 0.033]$. Divided by the sample standard deviation, the 95% confidence interval of the mean covers the standard deviations $[-0.079\sigma; 0.120\sigma]$. As previous research did not calculate effect sizes, we cannot compare the effect size in our setting with those of previous studies.

However, it is still possible that there is an effect for individual experience levels or individual task types that becomes visible when looking only at these specific subsets.

5.2 RQ 2: differences between task types

As explained in Section 3.4, the tasks differ in the type of problem the test subjects had to solve. This allowed to evaluate RQ 2. Using Wallis' method, tasks belonging to the same category were combined and one overall p-value was calculated per category. Table 11

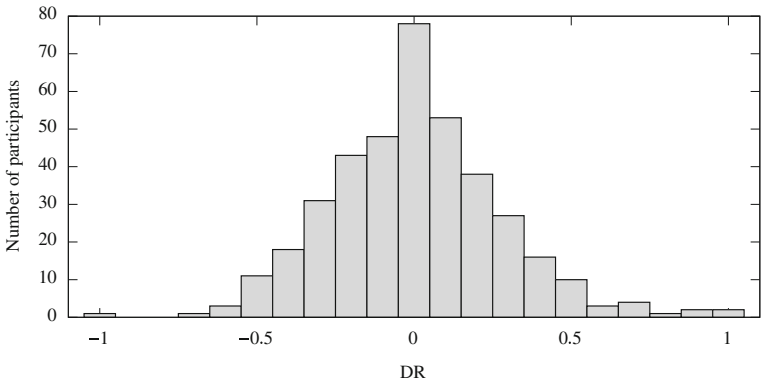


Fig. 8 Histogram of the DR distribution

Table 11 p-values per task category

Category	OUTP	FSE	FTG	UML	FTGUML	ARR
p-value	0.2814	0.0925	0.9985	0.3339	0.9412	1

shows the results of these calculations again for all task categories. No category showed significantly better results for tasks in the SH flavour opposed to tasks in the BW flavour. FSE showed slightly significant differences, with $p < 0.1$, but this might be a statistical artefact, given the large sample size.

Combining the results of all tasks provides another statistical test for SH's general effect according to RQ 1. For this test, results from all tasks must be combined. Using Wallis' method as described above, the tasks' p-values were combined to calculate an overall p-value of 0.74. This is much higher than any meaningful significance level, hence this method also shows no evidence for a positive answer to RQ 1 in the dataset.

5.3 RQ 3: influence of programming experience

243 test subjects also filled out a questionnaire about their programming experience, described in Section 3.3. The questionnaire yielded the general programming experience in months for 239 survey participants and the Java programming experience in months for 239 survey participants (the sets are not equal, as eight participants answered only one question). Figure 9 shows a histogram of the *DR*s grouped by years of experience of the test subjects.

We conducted a regression analysis to find out whether there is a correlation between programming experience and sensitivity to syntax highlighting. The regression analysis excludes the five-cent-fractile with the longest experience. These outliers have more than six years of experience with programming in general or more than four years of programming experience with Java. There are two reasons for this exclusion: First, after a whilst, knowledge about the SH scheme is saturated and additional Java experience does not increase sensitivity to SH anymore. In fact, Eclipse's SH scheme is quite simple and consists only of four colours and one variation in font type. The five-cent-fractile still leaves a very gentle learning curve of one colour per year. Second, the experience values rely on the honesty of the survey participants and their comprehension of the questionnaire. Although no specific evidence came up to cast doubt on the honesty and comprehension of the survey

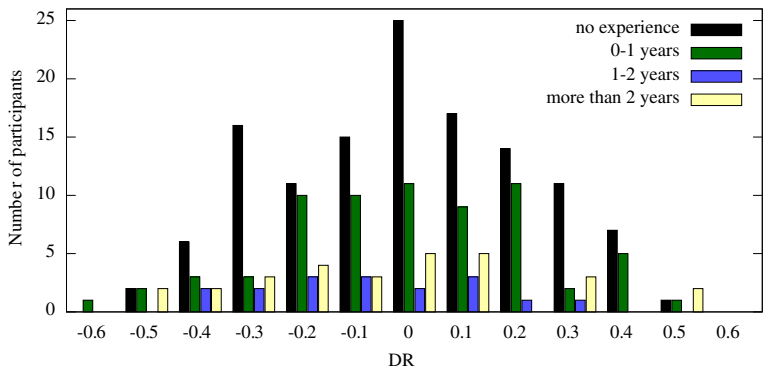


Fig. 9 Histogram of DRs grouped by Java programming experience (bin width 0.1)

Table 12 Results from the two (OLS) regression analyses with *DR* as dependent variable (all non-significant); all experiences are in months

Explanatory variable	Coefficient	F-test	p-value(F)
General programming experience	-0.0002	$F(1, 225) = 0.05$	0.8238
Java programming experience	-0.0005	$F(1, 226) = 0.16$	0.6903

participants, bogus values are more likely to be outliers. In an ordinary least squares (OLS) regression analysis, outliers impact the regression by the square of their deviation from the mean. Thus, extreme outliers have a strong impact on the regression. Even few bogus values of this kind could invalidate the analysis.

In the following (OLS) regression analysis, both the general programming experience in months as well as the Java programming experience in months are explanatory variables. General programming experience and Java experience correlate quite closely with each other. Thus, the analysis comprises two (OLS) regressions, one for each of the explanatory variables. The *DR* serves as dependant variable. Both regressions are insignificant. The results are listed in Table 12.

5.4 RQ 4: programming performance and experience

In order to cheque the answers of the questionnaire for plausibility, another regression analysis calculates the correlation between experience and the test subjects' score for all coding tasks, irrespective of their highlighting flavour. Programming experience in general correlates significantly with the test subject's score ($p = 0.0321$). Java experience correlates even highly significant with the test subject's score ($p = 0.0010$). Table 13 lists detailed results for these regression analyses. As shown, every month of Java programming experience increases the expected test score by 0.3146 percentage points. Figure 10 visualises this relationship. General programming experience has a much weaker effect, as every month of General programming experience increases the score by only 0.0594 percentage points.

Figure 11 highlights this relationship as a histogram, where the test subjects who answered about their Java experience are divided into four groups: no prior experience, at most one year of Java experience, at most two years of Java experience, and more than two years of Java experience. The histogram is normalised to relative frequencies, so each bar shows only the fraction of test subjects within each of the four groups that reach a score in the interval indicated on the x-axis. The histogram visually shows that the score distribution shifts to a greater average when the sample has more Java experience, measured as years since the test subject has started to work with Java.

Table 13 Results from the two (OLS) regression analyses with score in percentage as dependent variable; all experiences are in months

Explanatory variable	Coefficient	F-test	p-value(F)
General programming experience	0.0594	$F(1, 225) = 4.65$	0.0321*
Java programming experience	0.3146	$F(1, 226) = 11.18$	0.0010***

* significant at $\alpha = 0.05$

*** significant at $\alpha = 0.001$

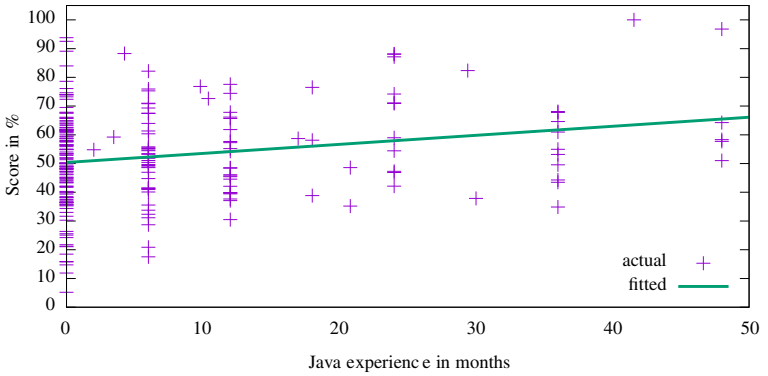


Fig. 10 Scatter plot showing actual test scores by Java experience and the fitted curve resulting from the regression analysis

The arithmetic mean score in the group with no prior Java experience is 50.37% and the sample standard deviation in this group is $\sigma = 16.64$ percentage points. Together with the information shown above, each year of Java programming experience increases the expected score by $12 \cdot 0.3146 = 0.2268\sigma$. Future research should find out which factors influence this effect size.

These results answer RQ 4 positively: Self-reported programming experience correlates positively with actual performance in the type of short programming-related tasks used in our study. Programming experience with the specific, object-oriented language used in the tasks has stronger influence on the score than general programming experience. This positive result is also a hint that the test subjects correctly reported their programming experience and that solving the short tasks used in our experiment requires programming skills.

6 Discussion

This section discusses the results and compares them to related studies. We draw implications from the data and ask questions for further research.

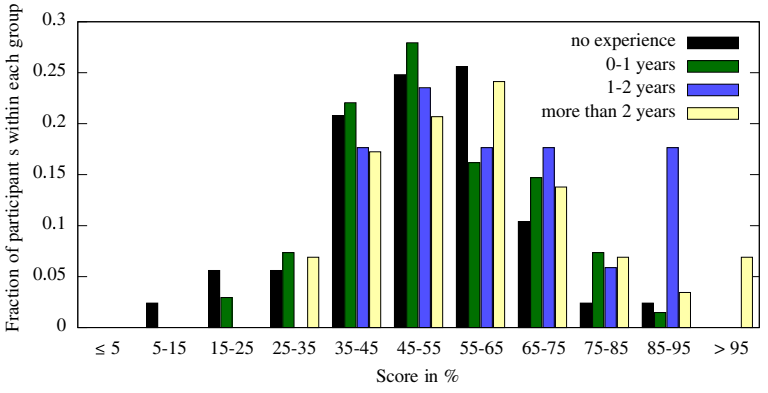


Fig. 11 Histogram of Scores grouped by Java programming experience (bin width 0.1), normalised to relative frequencies

6.1 RQ 1: syntax highlighting effect

With regard to RQ 1, we expected syntax highlighting to support novices in understanding source code. A task's SH flavour should thus have yielded more correct results from students than its corresponding BW flavour. This, however, was not the case in general, as the analyses in Sections 5.1 and 5.2 have shown – a surprising result, as previous experiments showed much stronger statistical effects given a smaller dataset (Baecker and Marcus 1989; Oman and Cook 1990; Rambally 1986; Tapp and Kazman 1994). In contrast to these earlier studies examined in Section 2.2, our experiment used a modern Integrated Development Environment (IDE)'s (Eclipse) syntax highlighting scheme and therefore presented test subjects with a program comprehension mechanism used in professional environments. This difference may account for the negative results.

As described in Section 2.1, a few studies also analysed syntax higlighting as used in in modern IDEs. Two previous studies with smaller data sets found similar results (Hakala et al. 2006; Beelders and du Plessis 2015). Sarkar (2015) found seemingly contradicting results. The reason for these differences are unclear as of yet and could be the python highlighting scheme, a different participant culture, or simply a statistical artefact.

Not only professional IDEs, even less sophisticated text editors provide syntax highlighting as a must-have feature. For example, the text editing library Scintilla (Scintilla Project 2014) devotes 52,4% of their source code to syntax highlighting.[2] Apparently, the tested syntax highlighting scheme is of no specific use for novices, so questions about the teaching approach to programming languages arise. Is it useful for novices to use professional tools? Could novices benefit from a different highlighting scheme? Can we train better developers by changing the learning environments? An interesting follow-up experiment could be a replication of our scenario with a syntax highlighting scheme used by educational IDEs (e.g. Allen et al. 2002, Grey and Flatt 2003).

Furthermore, with syntax highlighting not being as supportive as expected, ways to employ additional mechanisms to program comprehension as tested in related studies are of interest. One example of a more efficient colouring scheme follows Rambally's colouring of statements based on their function (Rambally 1986): Instead of colouring the source code based on the token type, built-in language constructs like classes and methods might be assigned colours based on their namespace. In object-oriented programming languages like Java and C#, namespaces are commonly used and e.g. accompanying frameworks categorise pre-defined classes in namespaces already, so the only effort necessary would be on the Integrated Development Environment (IDE)'s side.

Rambally (1986) also showed that there are cases where programmers favour a less efficient highlighting scheme over a more efficient one. More generally, Mehta and Zhu (2009) have shown that people favour colouring schemes that are considerably less suited for the task at hand. Hakala et al. (2006) and Beelders and du Plessis (2015) both explained that their participants found syntax highlighting useful, although they had not measured any objective benefit of syntax highlighting. This may explain why the current form of syntax highlighting is so common, even if its benefits were negligible, as in the context of our experiment. This paradox may also prevent the introduction of better highlighting schemes.

[2]The *.cxx files in lexer and lexlib contain 38,482 and 908 (LOC), respectively, whereas the whole project comprises 75,182 (LOC) in version 225.

6.2 RQ 2: task type dependency

Regarding the different task types, only the 'find syntactical errors' (FSE) category showed at least a slightly significant effect of syntax highlighting on the test subjects' results. For this task type, the test subjects achieved more correct results with the SH than the BW flavour. This supports the idea underlying RQ 2 that the task at hand is an important factor for the supportive effect of syntax highlighting. Indeed, that syntax highlighting would have a stronger effect on tasks in the category *FSE* than in other categories was expectable – it was the only category directly targeting syntactical problems: Test subjects had to hunt for syntax errors in code snippets. Besides this, it remains surprising that no other task type statistically shows a benefit from the use of syntax highlighting.

As in the analysis of RQ 1, this result begs for an even more critical look at the use of syntax highlighting nowadays. Regarding novices, only the search for syntactical errors could be shown to benefit from the current use and application of syntax highlighting. The use of a programming language's syntax is definitely problematic for novices. However, the tasks in the category FSE challenged the test subjects only to enter the line numbers of syntax errors a compiler would show them anyway. Thus, in a real programming scenario, programmers already receive a more elaborate explanation of syntax errors if they just try to compile the programme instead of skimming through the syntax-highlighted source code. On the other hand, maybe syntax highlighting teaches novices about the different types of tokens and therefore plays an important role in the learning phase of becoming a programmer. In this case, a less minimalistic colouring scheme than Eclipse's might be more beneficial.

6.3 RQ 3: familiarisation

According to RQ 3, the more experience programmers have with a specific type of SH, the easier it is for them to understand the message underlying the colouring of the text. Thus, experienced programmers should benefit more from SH. In fact, the need to learn the language of SH might then explain the lack of evidence for RQ 1, as the test subjects were mostly novices.

However, the results show no evidence for a positive answer to RQ 3. As discussed in the previous part of this section, there is also a theoretical explanation for a possibly negative answer to RQ 3: Maybe only novices benefit from SH, as only novices still have to learn the differences between different types of tokens. Yet, if such a relation were present, the regression analysis described in Section 5.3 should have detected it: The relation would have shown a negative coefficient for the experience and a low p-value. Thus, if a relation existed in either direction, it was too weak to be detected with our data and methods.

6.4 RQ 4: programming experience

The regression analysis in Section 5.4 provides an answer to RQ 4: Programmers with longer programming experience perform better – they solve code comprehension tasks involving short snippets of source code more correctly. We measured programming experience as the self-reported length of time for which a test subject had been programming in general and especially in Java. Siegmund et al. (2014) found years of programming experience a good predictor for the ability to solve short programming task that correspond to our category OUTP. Since they measured the time needed to solve a task and we measured

correctness, the results are not directly comparable, though. In contrast to our study, Dieste et al. (2017) found years of programming experience to have no effect on programming performance.

7 Threats to validity

In this section, we discuss possible limitations to our study. We analyse the experimental setup and try to identify threats to the internal, external, and general validity (Kitchenham et al. 2002).

7.1 General validity

As is the case in many studies in university environments, our test group consisted of students, which might reduce the credibility for practical application due to the controlled learning environment. Of course, this does not affect our experiment's validity as statements about novice programmers: first-year students are representatives of this target group.

Another possible threat to the general validity might result from wrong experiment design, e.g. the tasks might not measure program comprehension but rather a completely different cognitive process. We tried to avoid this pitfall commonly known as construct validity by using different types of tasks, so it becomes more unlikely that all tasks measure something different than program comprehension.

Tests on continuous values like parametric tests have a higher statistical power than tests on binary variables. Especially, Barnard's Tests has a lower statistical power than a t-test. Thus, results for the individual task types based on Barnard's Test may simply not have shown an effect because the statistical power was too low. However, the experiment's sample size was comparatively high, so the effect of syntax highlighting cannot be very strong if it is still not detected with Barnard's Test. Furthermore, we analysed RQ 1 with a t-test and also conducted an analysis of effect size and showed that the effect of syntax highlighting was small in the experiment setting.

Test subjects may have the opinion that SH improves their programming performance and therefore the presence of SH might have influenced their motivation and self-perception of the task difficulty, which in turn might have influenced the results. We used a blind experiment, so the test subjects did not know that each task existed in two flavours, BW and SH, to reduce this threat to validity.

7.2 Internal validity

Also, our participants had to use an unfamiliar software system and could not work with their accustomed Integrated Development Environment (IDE) or individual syntax highlighting settings. The experiments took place in a special room for electronic examinations and it is unclear how and if this artificial atmosphere and the technical equipment might have influenced the individual results. We tried to mitigate this risk by providing a trial session one week before the first experiment.

Furthermore, our tasks consisted of given code snippets. It is unclear how the snippets' size has influenced the results – it may be that code size is an important factor determining the effectiveness of using syntax highlighting. However, for example Rambally (1986) measured significant effects of highlighting using a programme of only 107 LOC.

7.3 External validity

The experiments only included tasks in which the test subjects had to read and understand, but not write source code. This implies the threat that the lack of interaction reduced the effect of syntax highlighting below statistical significance. However, even if syntax highlighting helped only in writing source code, it would do so by improved comprehensibility of the written code, because syntax highlighting itself has obviously no interactive component. This indicates that an effect of syntax highlighting on code comprehensibility would be a pre-condition for a positive effect on the ability to write source code. Almost all related research presented in Section 2 supports this line of argument, as they let the test subjects only read and not write code, and still showed positive effects on code comprehensibility for other types of highlighting than syntax highlighting (Baecker and Marcus 1989; Oman and Cook 1990; Rambally 1986; Tapp and Kazman 1994).

We measured whether test subjects solved the tasks correctly or not. The benefit of syntax highlighting however might not be a higher chance to solve a task correctly, but faster. As we did not measure the time needed to solve individual tasks, the analysis neither implies nor denies a relation between syntax highlighting and the speed of solving a task. However, as presented in Section 2, no previous study analysed the effect of highlighting on both speed and correctness and found an effect only on speed but not on correctness. But for example Oman and Cook (1990) showed that some forms of highlighting show significant effects on correctness, but not so much on speed. Even if there is an effect of syntax highlighting on speed, this does not invalidate the result that the effect on correctness is weak: As described in Section 5.1, with a confidence of 95%, syntax highlighting helps to solve at most 3.3% of the tasks used in this experiment correctly, which is 0.120 standard deviations.

8 Conclusion

We could not find evidence in our data that syntax highlighting as used in Eclipse has a beneficial effect on program comprehension for programming novices. If there is a beneficial effect, the effect size must be quite small. This result is surprising given that modern IDEs – in professional and educational incarnations – extensively use syntax highlighting in similar ways.

Previous research suggests that programmers sometimes prefer ineffective colouring schemes. Future research should explore the causes and investigate the optimal colouring scheme to please users and provide useful information.

Another explanation for the popularity of syntax highlighting – despite its lack of effect at least under the circumstances of the experiment described in this paper – may be that it is useful for learning a language but not for using it. This is an interesting research question worth exploring that might also yield insights for the preparation of learning materials.

As discussed in Section 7, we did not analyse whether syntax highlighting helps to solve a task quicker. This nevertheless interesting question is therefore still open to future research as well. Our experiments had constraints whose role should be explored in future experiments: Are there additional types of tasks, for example with a larger volume of source code, that might increase the usefulness of syntax highlighting? Is a syntax highlighting scheme different to Eclipse's more beneficial?

Furthermore, we tested program comprehension with existing code, hence we can only make assumptions about *reading* but not about *writing* code. For future research in this area, more interesting questions arise: Can syntax highlighting support novices in making less

syntactical errors whilst typing? Is an opposite approach – highlighting syntactical errors instead of tokens – more effective?

Contrary to Eclipse's syntax highlighting scheme, previous research demonstrates that other types of highlighting have beneficial effects on program comprehension (Baecker and Marcus 1989; Oman and Cook 1990; Rambally 1986; Tapp and Kazman 1994). Given that the effect of syntax highlighting for program comprehension is so small, current IDEs may use code colouring as a more effective feedback channel by providing alternative information like aforementioned researchers suggest. Other interesting options for code colouring are:

- The colouring of a class or function call could depend on the namespace the class or function is defined in. The colour would show which kind of classes the code uses, and so a glance over a piece of code would reveal already which part is responsible for e.g. file I/O, networking, etc. Varicoloured code might indicate cross-functional code, whilst single-coloured pieces of code might indicate a proper separation of concerns.
- Colour might as well indicate the number of edits to the code. If code has been edited very often, it might be better to rewrite the whole code section instead of applying yet another patch.
- Authorship information from versioning systems might be another interesting colouring option. Each colour of the source code represents one author. This allows developers to quickly identify sections edited by multiple authors. Research suggests that these sections are more prone to errors (Bird et al. 2011).

Acknowledgments We thank all students enrolled in the course *Programming in Java* for their participation in our experiment. We also thank Matthias Book, Tobias Brückmann, and Tobias Griebe for useful comments on earlier draughts of this paper. We further thank Florian Stefan and again Matthias Book for their help setting up and supervising the experiment. We thank Stefan Hanenberg for his feedback on the paper and for the discussions about statistics. We furthermore thank the anonymous reviewers for their valuable feedback.

References

Allen E, Cartwright R, Stoler B (2002) Drjava: a lightweight pedagogic environment for java. ACM SIGCSE Bullet. 34(1):137. https://doi.org/10.1145/563517.563395

Baecker RM, Marcus A (1989) Human factors and typography for more readable programs. ACM, New York

Beelders TR, du Plessis JPL (2015) Syntax highlighting as an influencing factor when reading and comprehending source code. J Eye Mov Res 9(1). https://doi.org/10.16910/jemr.9.1.1

Bergin S, Reilly R (2005) Programming: Factors that influence success. SIGCSE Bull 37(1):411–415. https://doi.org/10.1145/1047124.1047480

Bird C, Nagappan N, Murphy B, Gall H, Devanbu P (2011) Don't touch my code!: Examining the effects of ownership on software quality. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM, New York, ESEC/FSE '11, pp 4–14. https://doi.org/10.1145/2025113.2025119

Chaffin WW, Rhiel SG (1993) The effect of skewness and kurtosis on the one-sample t test and the impact of knowledge of the population standard deviation. J Stat Comput Simul 46(1-2):79–90. https://doi.org/10.1080/00949659308811494

Crosby M, Stelovsky J (1990) How do we read algorithms? a case study. Computer 23(1):25–35. https://doi.org/10.1109/2.48797

Dieste O, Aranda AM, Uyaguari F, Turhan B, Tosun A, Fucci D, Oivo M, Juristo N (2017) Empirical evaluation of the effects of experience on code quality and programmer productivity: an exploratory study. Empir Softw Eng 22:1–86. ISSN:1573-7616. https://doi.org/10.1007/s10664-016-9471-3

Dimitri GM (2015) The impact of syntax highlighting in sonic pi. In: Psychology of Programming Interest Group Annual Conference 2015. Proceedings of, pp 59–70

Dragicevic P (2016) Fair Statistical Communication in HCI. Springer International Publishing, Cham, pp 291–330. https://doi.org/10.1007/978-3-319-26633-6_13

Erguler K (2013) Barnard: Barnard's Unconditional Test. R package version 1.3. http://cran.r-project.org/package=Barnard, [accessed 2014-07-22]

Feigenspan J, Kästner C, Apel S, Liebig J, Schulze M, Dachselt R, Papendieck M, Leich T, Saake G (2013) Do background colors improve program comprehension in the #ifdef hell? Empir Softw Eng 18(4):699–745. https://doi.org/10.1007/s10664-012-9208-x

Gellenbeck EM, Cook CR (1991) Does signaling help professional programmers read and understand computer programs? In: Fourth Workshop on Empirical Studies of Programmers, Ablex Publishing Corporation, Norwood, pp 82-98

Goldstein EB (1995) Sensation and Perception. Wadsworth Publishing, Belmont

Grant EE, Sackman H (1967) An exploratory investigation of programmer performance under on-line and off-line conditions. IEEE Trans Hum Fact Electron HFE-8(1):33–48. https://doi.org/10.1109/THFE.1967.233303

Gray KE, Flatt M (2003) ProfessorJ: A gradual introduction to java through language levels. In: Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, New York, OOPSLA '03, pp 170–177. https://doi.org/10.1145/949344.949394

Gruhn V, Hannebauer C (2012) Using wikis as software development environments. In: Proceedings of the the 11th SoMeT_12, IOS Press, Amsterdam, Frontiers in Artificial Intelligence and Applications, vol 246, pp 3–20. https://doi.org/10.3233/978-1-61499-125-0-3

Hakala T, Nykyri P, Sajaniemi J (2006) An experiment on the effects of program code highlighting on visual search for local patterns. In: 18th Workshop of the Psychology of Programming Interest Group, Proceedings of, pp 38–52

Hannebauer C, Hesenius M, Gruhn V (2017) Lab package for the syntax highlighting experiment. https://www.uni-due.de/~hw0433/sh/

Holden E, Weeden E (2003) The impact of prior experience in an information technology programming course sequence. In: Proceedings of the 4th Conference on Information Technology Curriculum, ACM, New York, CITC4 '03, pp 41–46. https://doi.org/10.1145/947121.947131

Jedlitschka A, Ciolkowski M, Pfahl D (2008) Reporting experiments in software engineering. Springer, Berlin, pp 201–228

Kitchenham B, Pfleeger S, Pickard L, Jones P, Hoaglin D, El Emam K, Rosenberg J (2002) Preliminary guidelines for empirical research in software engineering. IEEE Trans Softw Eng 28(8):721–734. https://doi.org/10.1109/TSE.2002.1027796

Kleinschmager S, Hanenberg S (2011) How to rate programming skills in programming experiments?: A preliminary, exploratory, study based on university marks, pretests, and self-estimation. In: Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools, ACM, New York, PLATEAU '11, pp 15–24. https://doi.org/10.1145/2089155.2089161

Knuth DE (1984) Literate programming. Comput J 27(2):97–111. https://doi.org/10.1093/comjnl/27.2.97

van Laar D (1989) Evaluating a colour coding programming support tool. In: Sutcliffe A, Macaulay L (eds) Proceedings of the Fifth Conference of the British Computer Society Human-Computer Interaction Specialist Group, Cambridge University Press. British Computer Society Workshop Series, Cambridge, pp 217–230

Lientz BP, Swanson EB, Tompkins GE (1978) Characteristics of application software maintenance. Commun ACM 21(6):466–471. https://doi.org/10.1145/359511.359522

LPLUS GmbH (2014) LPLUS-System on-campus. http://lplus.de/lplus.de/index_9_19_2_...html, [accessed 2014-07-16]

Ludbrook J (2008) Analysis of 2 x 2 tables of frequencies: matching test to experimental design. Int J Epidemiol 37(6):1430–1435. https://doi.org/10.1093/ije/dyn162

Mehta R, Zhu RJ (2009) Blue or red? exploring the effect of color on cognitive task performances. Science 323(5918):1226–1229. https://doi.org/10.1126/science.1169144

Microsoft Developer Network (2015) C# coding conventions (c# programming guide). http://msdn.microsoft.com/en-us/library/ff926074.aspx, [accessed 2017-03-08]

Oman PW, Cook CR (1990) Typographic style is more than cosmetic, vol 33. https://doi.org/10.1145/78607.78611

Oracle Technology Network (1999) Code conventions for the java programming language. http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html, [accessed 2017-03-08]

Prechelt L (1999) The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really? Interner Bericht, Fakultät für Informatik. Universität Karlsruhe, Karlsruhe
Rambally GK (1986) The influence of color on program readability and comprehensibility. In: Proceedings of the Seventeenth SIGCSE Technical Symposium on Computer Science Education, ACM, New York, SIGCSE '86, pp 173–181. https://doi.org/10.1145/5600.5702
Reijers H, Freytag T, Mendling J, Eckleder A (2011) Syntax highlighting in business process models. Decis Support Syst 51(3):339 – 349. https://doi.org/10.1016/j.dss.2010.12.013
Sackman H, Erikson WJ, Grant EE (1968) Exploratory experimental studies comparing online and offline programming performance. Commun ACM 11(1):3–11. https://doi.org/10.1145/362851.362858
Sarkar A (2015) The impact of syntax colouring on program comprehension. In: Psychology of Programming Interest Group Annual Conference 2015, Proceedings of, pp 49–58
Scintilla Project (2014) Scintilla – a free source code editing component for win32, gtk+, and os x. http://www.scintilla.org, [accessed 2014-07-22]
Seo H, Sadowski C, Elbaum S, Aftandilian E, Bowdidge R (2014) Programmers' build errors: A case study (at google). In: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, ICSE 2014, pp 724–734. https://doi.org/10.1145/2568225.2568255
Siegmund J, Kästner C, Liebig J, Apel S, Hanenberg S (2014) Measuring and modeling programming experience. Empir Softw Eng 19(5):1299–1334. https://doi.org/10.1007/s10664-013-9286-4
Tapp R, Kazman R (1994) Determining the usefulness of colour and fonts in a programming task. In: Proceedings of the Third IEEE Workshop on Program Comprehension. IEEE Computer Society Press, Los Alamitos, pp 154–161. https://doi.org/10.1109/WPC.1994.341265
Tiarks R (2011) What programmers really do: an observational study. Softwaretechnik-Trends 31(2):36–37
Wallis WA (1942) Compounding probabilities from independent significance tests. Econometrica, J Econ Soc 10(3/4):229–248

Christoph Hannebauer is an IT consultant working for Glück & Kanja Consulting and specializes in security of cloud solutions. He received his doctoral degree from the University of Duisburg-Essen in 2016, where his research focused on human factors in software engineering and especially developers of Open Source software.

Marc Hesenius is a research associate at paluno – The Ruhr Institute for Software Technology and a PhD candidate at the University of Duisburg-Essen. He holds a master's degree in software engineering. Prior to his academic career, he worked several years as a consultant for web technologies and freelance software engineer. His research interests focus around the engineering aspects of human-computer interaction.

Volker Gruhn is full professor for Software Engineering at the University of Duisburg-Essen. His research focuses on mobile applications cyber-physical systems. He received his doctoral degree from the University of Dortmund. He also founded the software consulting and development company adesso in 1997 and is the chair of its board now.