Two N-of-1 self-trials on readability differences between anonymous inner classes (AICs) and lambda expressions (LEs) on Java code snippets

Stefan Hanenberg¹ · Nils Mehlhorn²

Accepted: 28 October 2021 / Published online: 20 December 2021
© The Author(s) 2021

Abstract

In Java, lambda expressions (LEs) were introduced at a time where the similar language construct anonymous inner class (AIC) already existed for years. But while LEs became quite popular in mainstream programming languages in general, their usability is hardly studied. From the Java perspective the need to study the relationship between LEs and AICs was and is quite obvious, because both language constructs co-exist. However, it is quite usual that new language constructs are introduced although they are not or hardly studied using scientific methods – and an often heard argument from programming language designers is that the effort or the costs for the application of the scientific method on language constructs is too high. The present paper contributes in two different ways. First, with respect to LEs in comparison to AICs, this paper presents two N-of-1 studies (i.e. randomized control trials executed on a single subject) where LEs and AICs are used as listeners in Java code. Both experiments had two similar and rather simple tasks (“count the number of parameters”, respectively “count the number of used parameters”) with the dependent variable being reaction time. The first experiment used the number of parameters, the second the number of used parameters as the controlled, independent variable (in addition to the technique LE and AIC). Other variables (LOC, etc.) were randomly generated within given boundaries. The main result of both experiments is that LEs without type annotations require less reading time ($p < .01$) with rather small effect sizes ($\eta_p^2 < .2$, reduction of reaction time of at most 35%). The results are based on 9,600 observations (one N-of-1 trial with eight replications). This gives evidence that the readability of LEs without type annotations improves the readability of code. However, the effect seems to be so small, that we do not expect this to have a larger impact on daily programming. Second, we see the contribution of this paper in the application of N-of-1 trials. Such experiments require relatively low effort in the data selection but still permit to analyze results in a non-subjective way using commonly accepted analysis techniques. Additionally, they permit to increase the number of selected data points in comparison to traditional multi-subject experiments. We think that researchers should take such experiments into account before planning and executing larger experiments.

Communicated by: Denys Poshyvanyk

✉ Stefan Hanenberg
stefan.hanenberg@uni-due.de

Extended author information available on the last page of the article.

Keywords Programming · Programming language · Readability · Lambda expressions

1 Introduction

It is quite common that programming languages evolve over time by introducing new or changing existing language features. For example, in the programming language Java we find the introduction of (anonymous) inner classes, generic types, or lambda expressions over the last three decades. While in some cases and for some languages the motivation for a new language feature might be the runtime performance of resulting programs or the provability of certain program characteristics (e.g. as motivation for type systems), most language features are probably introduced in order to improve development ergonomics, i.e. they focus on the readability, understandability, or writability of programs.

However, while topics such as readability and understandability are inherently human-centered topics, the number of human-centered studies in general is extremely low: according to a study by Kaijanaho (2015, p. 133), the number of human-centered studies using randomized controlled trials (RCTs) – a kind of study that other empirical disciplines such as medicine, psychology, etc. consider as the de facto standard for human-centered research – in that field up to 2012 was just 22. Such rather low number seems to be in line with the study by Buse et al. (2011, p. 649) who found that only a very small percentage of papers from the field of programming languages contains a user evaluations. While Kaijanaho's study could be considered as slightly outdated today (since only studies up to 2012 were considered), even slightly newer studies did not find large empirical evidence in the field of software development in general (see for example Ko et al. 2015; Vegas et al. 2016). As a consequence, one can conclude that there is extremely low evidence for or against certain programming language features. The consequence of this is that it is quite unclear whether certain language features have a positive or maybe a negative impact on software development – a situation that we consider as problematic taking into account the relevance of software construction on today's society. For example, although the introduction of generic types in Java was considered as a large step in the evolution of the language and probably required larger education and migration efforts, there is doubt whether this construct actually plays or played a major role after all.¹

This work focuses on the language constructs anonymous inner classes (AIC) and lambda expressions (LE). The reason for this is that LEs became more and more popular in modern object-oriented programming languages, although the background of LEs is in functional programming. This popularity can be seen from the introduction of newer APIs such as the Java Stream API which – according to the common code examples – promotes LEs over AICs, although the latter could technically be used as well. Taking into account that AICs are much more verbose than LEs, it is from our perspective plausible to study the readability of both language constructs. Furthermore, there seems to be a common assumption that LEs are more readable than AICs: the IntelliJ IDE even visualized AICs as LEs years before LEs were officially introduced in Java. And again, one should ask whether such a visualization was actually helpful or maybe even the opposite.

In order to study this possible effect, we designed two N-of-1 studies where simple reading tasks were given to a participant and where the code either used AICs or LEs. The AICs

¹ A study by Parnin et al. revealed that hardly generic types are written by developers (Parnin et al. 2013). On the other hand, it should be noted that generic types are the foundation for APIs such as the Stream API.

or LEs appeared in this code in the form of listeners (according to the observer design pattern (Gamma et al. 1994)) – a typical situation where AICs were originally leveraged. The result of the study is that LEs without type annotation improve the readability of the code in comparison to AICs, but the actual effect is small.

From our perspective, the contribution of the present paper consists of two different aspects. First, we provide a technical contribution meant to answer the question whether there are differences in the readability of AICs or LEs. The result of the study is that LEs without annotations do improve the readability – but although there is strong evidence that this effect exists, there is also evidence that the effect is rather small. Second, we see a contribution in the design and execution of N-of-1 trials – an experimental design that is quite uncommon in software science but extremely widespread in other empirical disciplines. We think that the application and acceptance of N-of-1 trials in software science could help to overcome the current problem that relatively few empirical results exist in the field in general and in particular in programming language design. While the evidence gathered from N-of-1 trials is less than from RTCs with multiple participants, they can be applied to get an impression of expected effect sizes as a first step towards larger trials since N-of-1 trials offer reduced efforts for data collection.

However, this benefit is not for free and requires a number of counteractions in design and analysis in order to overcome (or at least identify) typical problems in N-of-1 trials (respectively in so-called crossover trials in general). In this paper, we try to document as detailed as possible the different steps we took in order to satisfy these special demands of N-of-1 trials.

This paper starts in Section 2 with an introduction of the background which consists of the technical background (AICs and LEs) as well as a methodological background of N-of-1 trials. After discussing related work in Section 3, we introduce the design of the present experiment (Section 4) followed by the analysis (Section 5). Afterwards, we present the analysis of additional replications in Section 6 and discuss their relationship to the initial study. After discussing threats to validity in Section 7, we discuss the present work (Section 8). We intentionally discuss this work from two perspectives: first, from the technical perspective (comparison of AICs and LEs) and second from the methodological perspective (use of N-of-1 trials). Finally, we conclude this paper in Section 9.

2 Background

The present paper describes an N-of-1 study that compares LEs with AICs in the programming language Java. Consequently, we describe both topics in separate sections: LEs and AICS from the perspective of programming language design and programming, N-of-1 experiments from the perspective of experimental setups and layouts.

2.1 Inner Classes, Anonymous Inner Classes, and Lambda Expressions in Java

While the concept class is essential in object-oriented programming languages (at least in class-based languages²), it took some time before the concept of inner class, anonymous inner class or lambda expression became an essential part of object-oriented programming languages.

²See Dony et al. (1992) for the distinction between class-based and object-based, respectively prototype languages.

For example, the mainstream programming language Java originally did not contain inner classes when its language specification appeared in 1996 (Gosling et al. 1996). They were introduced in the second version of the language specification which also included anonymous classes (Gosling et al. 2000, pp. 135). This addition was guided by changes in the language API for graphical user interfaces, where the concept of listeners was applied – typically using anonymous inner classes.

An alternative to this kind of listener implementation emerged with Java 8 in 2013 (the language specification was published in 2014 (Gosling et al. 2014)) where lambda expressions were introduced (a language construct that is well-known in functional programming³). Lambda expressions are unnamed functions whose parameter types may either be explicitly written by the developer or inferred by the language. Additionally, Java 8 introduced method references that permit to pass a named method as a parameter.

Figure 1 illustrates how the different language constructs changed the implementation of listeners (registered at a `java.awt.Button`). Although technically comparable to AICs⁴, LEs are syntactically shorter, especially when the parameter type annotations are omitted (see lines 18–19 in comparison to lines 7–11). Furthermore, AICs and LEs in comparison to named classes can be provided as an in-line implementation, i.e., no separate constructs need to appear elsewhere in the code. As a consequence, developers do not navigate to a different place in the code in order to understand the semantics or the listener. In comparison to this, a developer who reviews the same logic implemented with a named class or a method reference needs to find the source of the class or method definition in order to understand the semantics (in case the semantics are not apparent from the name of the class or method). In the example, the use of a named class (line 4) requires navigation to the class definition (lines 30–34) in order to read what the listener actually does. If a method reference is used (line 22), the developer needs to navigate to the method definition (lines 25–27) in order to find the actions performed by the listener.

Today, Java promotes the use of LEs via APIs such as the Stream API.⁵ For example, the documentation recommends the use of LEs as filter criteria for streams although inner classes could be used as well. Consequently, it seems natural to us to ask whether one or the other construct has a benefit in readability.

Other main stream programming languages evolved in a similar way. For example, the programming language Python which was originally released under the version 0.9 (Van Rossum 1991) introduced LEs one version later (in version 1). In C++, LEs were introduced in version 11 (Stroustrup 2013), C# introduced LEs in version 3 (Microsoft C. 2007), PHP received them in version 5.3, etc.

Additional to the evolution of programming languages there were other movements towards LEs. For example, the IDE IntelliJ already visualized AICs as LEs in 2009 – about five years before lambdas became part of Java's language specification.⁶

³The theoretical background for functional programming is the lambda calculus that has lambda expressions as one of the core language features (see for example (Pierce 2002) for a general introduction into the lambda calculus and its application for modeling programming languages).

⁴One difference in anonymous inner classes and lambda expressions is the meaning of the keyword `this` which implies that the body of a function in an anonymous inner class cannot be directly used as a body of a lambda expression, but there are techniques to handle this situation, see for example <https://stackoverflow.com/questions/27762488/how-can-a-java-lambda-expression-reference-itself>. Another difference is that AICs can define multiple methods and states.

⁵<https://docs.oracle.com/javase/8/docs/api/overview-summary.html>

⁶<https://blog.jetbrains.com/idea/2009/03/closure-folding-in-intellij-idea-9-maia/>

```

1  // Listener construction and registration using
2
3  // (a) a named class (class definition at the bottom)
4      button.addActionListener(new MyActionListener());
5
6  // (b) an anonymous class
7      button.addActionListener(new ActionListener(){
8          public void actionPerformed(ActionEvent event) {
9              System.out.println("HelloWorld");
10         }
11     });
12
13 // (c) a lambda expression
14     button.addActionListener((ActionEvent event) ->
15         System.out.println("HelloWorld"));
16
17 // (d) a lambda expression without type annotation
18     button.addActionListener(
19         (event) -> System.out.println("HelloWorld"));
20
21 // (e) a method reference (method definition below)
22     button.addActionListener(this::myListener);
23
24 // Named method for method reference
25     void myListener(ActionEvent event) {
26         System.out.println("HelloWorld");
27     }
28
29 // Class definition for named class
30 class MyActionListener implements ActionListener {
31     public void actionPerformed(ActionEvent event) {
32         System.out.println("HelloWorld");
33     }
34 }

```

Fig. 1 Different implementations of an action listener using (a) a named class, (b) an anonymous class, (c) a lambda expression, (d) a lambda expression without type annotations, and (e) a method reference

However, despite of the wide adoption of LEs in programming languages and first indicators that LEs are also increasingly accepted by developers (see for example the study by Mazinanian et al. (2017)), it is unclear whether the application of LEs has a measurable effect on the usability of a programming language. A first study on the usability of lambda expressions came to the conclusion that they have rather a negative effect on the usability – at least if the LEs are used for iterating collections (in comparison to traditional loops and iterators) and if they are used by novices or junior professionals in the context of C++ (Uesbeck et al. 2016).

In the case of Java, it is questionable whether just the application of AICs would not have been sufficient. If LEs do not have a measurable (positive) effect on the programming language usability, their introduction could have been even harmful: a new language construct must be taught, learned and understood while additional tool support is required. Also, if two language constructs are available that essentially perform the same tasks and both constructs are applied in the same code base, the readability of this code base is probably reduced.

Consequently, there are two main questions that should be addressed in Java. The first and rather obvious one is whether there are measurable differences in usability between

LEs and AICs. The second (and more general) question is whether the ongoing spread of rather traditional functional programming language constructs in main stream languages do actually help developers – or the opposite.

The present paper addresses the first question with the aid of two N-of-1 self experiments with the focus on readability. Consequently, it is also necessary to discuss such kind of experiments.

2.2 N-of-1 Studies and Crossover Trials

N-of-1 studies are studies performed on a very small number of subjects (traditionally just a single subject – denoted by the number 1 in the term N-of-1). Such studies are specializations of so-called crossover trials where participating subjects are tested on multiple treatments. We start this section by showing that N-of-1 studies play an essential role in the scientific literature in general (Section 2.2.1), then we show in Section 2.2.2 that crossover trials are used relatively often in software science (despite the fact that experimentation is actually rather rare in software science). Then, we start discussing some general pros and cons of crossover trials in general and N-of-1 studies in particular.

2.2.1 On the Relevance of N-of-1 Studies in the Scientific Literature

N-of-1 studies, which are also called single-case studies, single-subject or single-patient studies (see Vieira et al. 2017), are quite common in empirical disciplines. The term N-of-1 study describes a (potentially randomized) control trial performed on a single subject. Such kind of studies have a long history in quantitative research starting (according to Mirza et al. 2017) probably in 1676. Tate et al. (2019) list a number of studies that illustrate the relevance of N-of-1 studies in behavioral sciences. For example, they cite the study by Shadish and Sullivan (2011), who examined 21 journals in behavioral sciences in 2008 and found that 44% of the published studies were N-of-1 studies, or they cite the study by Percides et al. (Percides et al. 2006) that states that 39% of the records in the PsycBITE evidence database were single-case studies.

Due to its relevance, N-of-1 trials are contained in widely accepted research standards such as CONSORT (see Vohra et al. 2015), APA JARS (see Appelbaum et al. 2018)), and even the very detailed and restrictive What Works Clearinghouse standard (WWC, see Kratochwill et al. 2010, Institute of E. S. 2020, p. 77) – research standards that are today mandatory for a large number of scientific journals in empirical disciplines or relevant for political decisions (the latter one holds for WWC). Consequently, N-of-1 trials are far from being exotic experimental setups in other disciplines, even the opposite: studying a topic using N-of-1 trials is a standard approach and consequently, this is appreciated by corresponding research standards.

Just recently the ACM Transaction of Computing Education accepted APA JARS as a research standard although the journal makes explicit that it “encourage[s], but do[es] not require, papers to use the APA Journal Article Reporting Standards (JARS)” (ACM Transactions on C. E. 2021). Taking into account that N-of-1 trials are an accepted study design in APA JARS, it means that TOCE implicitly accepts N-of-1 trials as valid study designs. This means that “being just an N-of-1 trial” is an invalid exclusion criterion for TOCE. And one should keep in mind that the given situation is not that one software science journal accepted one exotic research standard that accidentally contains N-of-1 trials. The

situation is, that today most journals in the field of software science do not make their research standard explicit.⁷

In contrast to that, it is quite usual to make the research standard explicit in other empirical disciplines. For example, one can take a look into the field of medicine and use, for example, the ranking by Jemielniak et al. (2019) as an indicator for how prestigious medical journals are. All journals from the top five of the list make their research standard explicit. This statement holds for The Cochrane Database of Systematic Reviews, The New England Journal of Medicine, PLOS One, The BMJ, and The Journal of the American Medical Association.⁸

2.2.2 Experimentation and Crossover Trials in Software Science

While the above said holds for traditional empirical disciplines, it is well-documented that controlled experiments in general hardly play a role in software science. For example, Kaijanaho (2015) documented that only 22 randomized controlled trials (RCTs) on programming language usability with human participants could be found up to 2012. Actually, this is in line with the study by Buse et al. (2011, p. 649) who found that papers from the field of programming languages contain the lowest percentage of user evaluations (below 5% of the found literature). Ko et al. (2015) analyzed publications at the four software engineering venues International Conference on Software Engineering (ICSE), ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), ACM Transactions on Software Engineering and Methodology (TOSEM) and IEEE Transactions on Software Engineering (TSE). These can be considered as part of the leading software engineering venues. Out of 1,701 submitted papers only 44 were identified as RCTs with human participants. Other authors found comparable low numbers for different venues (Tichy et al. 1995; Zelkowitz and Wallace 1998; Glass et al. 2002; Zelkowitz 2009). Just a more recent study by Vegas et al. from 2016 (Vegas et al. 2016) found slightly higher numbers in the journals and conferences TSE, EMSE, TOSEM, ICSE, ESEM and FSE in the years 2012–2014 (82 RTCs with human participants among 930 papers, where 45% of them appeared in 2014) .

Consequently, it is recurringly criticized in literature that a number of software science techniques today are applied but hardly empirically evaluated (see for example Tichy 1998; Hanenberg 2010; Stefik and Hanenberg 2014, 2017 among many others).

Taking such very low numbers of experiments into account it is not astonishing that the found RCTs follow rather traditional setups with a number of subjects divided into different treatment groups. So far, we are only aware of one experiment in the software science literature that is in principle a N-of-1 experiment: Hollmann and Hanenberg (2017) reported on an experiment series in software visualization where the first experiment was only performed on 5 subjects (while two of the subjects were the experimenters). Although this pre-experiment could be considered as an experiment with a very low sample size, the

⁷Up to the authors' knowledge, TOCE is up to today the only journal in software science that makes its research standard explicit. If we use for example the raking by Robert Feldt from <http://www.robertfeldt.net/advice/se.venues/> about the top journals in our field, the top five journals (IEEE Transaction of Software Engineering, Empirical Software Engineering, ACM Transactions on Software Engineering and Methodology, Automated Software Engineering, and Information and Software Technology) do not make explicit what research standards their authors should (or could) fulfill.

⁸According to <http://www.consort-statement.org/about-consort/endorsers> all journals endorse CONSORT except Cochrane that has its own standard.

authors report parts of the results of each participant individually, i.e. the experiment is reported as a N-of-1 trial with four repetitions.

While on the one hand N-of-1 experiments play hardly a role in software science, crossover trials are not uncommon: a crossover trial is an experiment layout on a sample of subjects where each individual subject is tested on multiple treatments (see Senn 2002 for a general introduction into crossover trials), i.e., an N-of-1 trial is a crossover trial on a very small number of subjects. Indicators that crossover studies are not uncommon can be easily found. For example, the previously mentioned experiment series by Hollmann and Hanenberg contains two randomized crossover trials (Hollmann and Hanenberg 2017), a different experiment series on type systems contains crossover trials as well (see for example Stuchlik 2011; Hoppe and Hanenberg 2013; Petersen et al. 2014; Fischer and Hanenberg 2015; Okon and Hanenberg 2016 among others) and among all RCTs identified by Vegas et al. more than $\frac{1}{3}$ of them were crossover trials (Vegas et al. 2016, p. 123). Therefore, testing subjects on multiple treatments is quite common.

2.2.3 Advantages and Disadvantages of Crossover and N-of-1 Trials

Crossover trials have a number of advantages but also a number of weaknesses for which they have been criticized in the literature (see for example Kitchenham et al. 2003). While some of those weaknesses exist in other experimental layouts as well, they are potentially higher in crossover designs and thus in N-of-1 trials.

Some of the general confounding factors in experimentation are, for example, learning effects (where measurements change over time because a subject learned within the experiment execution), novelty effects (where measurements change over time because a subject gets used to the experimental environment), or fatigue effects (where measurements change over time because a subject gets tired). The reason why such confounding factors probably become more problematic in crossover designs is that they depend on time, while an intrinsic characteristic of crossover designs is that dependent variables are measured at different points in time on the same subject.

For example, if a simple AB/BA layout is applied, it means for the first group that the dependent variable under the treatment B is tested after the dependent variable for the treatment A is measured. The comparison between A and B hence does not only contain the “pure data”. In case a learning effect exists, the measurement for B also contains this learning effect (in comparison to A). If a fatigue effect exists, it is contained in B as well. And if the novelty effect plays a role, this effect is contained in the measurement of A. Due to this, one does not only speak about the treatment effect but also about the period effect (see Madeyski and Kitchenham 2018, p. 1984) – and there are good reasons why one should be concerned about the period effect (see for example Wellek and Blettner 2012 among many others). Nevertheless, even in the presence of periodic effects crossover trials still have the ability to show the effect of one factor as long as the main effect is not small in comparison to the confounding factors (see a corresponding discussion by Stuchlik 2011, pp. 99–100).

Other disciplines such as medicine handle this problematic time-dependent effect in the experimental design: “a “washout” period is often built into the study design to separate two treatment periods to eliminate “carry-over” effects. A frequent recommendation is for the washout period to be at least 5 times the half-life of the treatment with the maximum half-life in the study.” (Evans 2010, p. 10)”. The underlying idea of such washout periods is that a drug, once given to a subject, lasts for some time in the subject’s body and causes an effect until it is washed out. However, in the field of software science it is relatively unclear

how long the effect of a certain treatment lasts and consequently, it is quite problematic to define a strategy for its reduction upfront.

While the above said holds for crossover trials in general, there are good reasons to assume that the period effect is even more problematic in N-of-1 trials. Traditional crossover trials operate on a set of subjects, i.e., there are multiple measurements for the AB group and multiple measurements for the BA group. As a consequence, it is possible to compare the results from the first period from one group with the results from the second period in the other group. A measured difference indicates a period effect. Furthermore, since there is typically more than one subject in the AB group and more than one subject in the BA group, possible (random) differences are (possibly) factored out. All this is not possible in N-of-1 trials, because if an N-of-1 trial is executed on just a single subject, there is no additional measurement for a given period that could be used for a comparison with the same period. Hence, while in crossover trials in general special care is needed for the period effect, even more care is necessary in N-of-1 trials.

Having said this, there are still good reasons for running crossover trials in general and to run N-of-1 trials in particular. If we assume that the often mentioned 10x problem (see for example (McConnell 2011))⁹ is real – or at least that there is a large deviation between participants in software science experiments (which might or might not make a difference of factor 10) – this problem is reduced by crossover trials because subjects are tested under different treatments. And an N-of-1 trial does not suffer from the problem at all: if there is just one participant, differences measured between treatments cannot be the result of deviations between participants. Furthermore, there is another very practical reason for running N-of-1 trials: the effort of data collection. While experimenters often complain about the large effort to collect data (such as organizing appointments for measurements, finding subjects, etc., see for example (Ko et al. 2015, pp. 115–120)), this is typically not a problem in N-of-1 trials: the organization of the trial just depends on a single participant. Hence, N-of-1 trials have the ability to show quickly whether a treatment has an effect: “Those who use the single-subject approach find it both a powerful and satisfying research method. One reason for this is that the method provides feedback quickly to the investigator about the effects of the treatment conditions. The experimenter knows relatively soon whether the treatment is working or not working.” (Lammers and Badia 2004, p. 14-4).

However, there is the potential problem of the participant bias in crossover trials in general and in N-of-1 trials in particular. In medicine it is quite common to design a blind study where participants are not aware of the treatments given to them (see for example Forbes 2013). Examples for such blinding are placebo studies (see for example Kam-Hansen et al. 2014)¹⁰ The goal of blinding is to reduce the potential influence of a participant’s bias on the experiment results. However, it is unclear up to today how different treatments can be hidden from the participant in software engineering. For example, when two programming languages should be studied in a crossover study, it remains unclear how the different treatments (the programming languages) can be hidden from the participant if he actually has to apply them. And in such situation, the personal bias of a subject might intentionally or unintentionally influence the experiment results. This problem is probably even larger in N-of-1 studies, because in case the participant is strongly biased, this bias will probably have

⁹But we need to mention that the evidence for the 10x problem is quite low (Bossavit 2015, pp. 36) – the factor 10 should rather be used as a metaphor that experimenters expect larger deviation among subjects in software engineering experiments.

¹⁰Although it is today debated whether a placebo is actually a mechanism for blinding or whether it has an effect itself (see Kaptchuk 1998).

a larger effect on the result than it would have in a multi-subject study. However, one also needs to take into account that this problem might also exist in traditional AB studies: if a subject has to use for example on programming language in an AB study and he has some bias against such language, there is still the danger that this bias has a measurable effect.

Another problem could occur if the only tested subject is one of the rare persons on whom the given treatment has no effect (although it might have an effect on the majority of people).

Hence, one can summarize that the reduced deviation among subjects is both an advantage and disadvantage of N-of-1 trials at the same time due to the high influence of a single subject. Having this in mind, one should be very careful with N-of-1 trials – and one should be especially careful when there are obvious biases of the participant.

In order to summarize the above said: N-of-1 trials have advantages and disadvantages. Advantages are the relatively low effort to run such experiments and the reduced deviation among subjects, disadvantages are the period effects (which are harder to detect) and the very high influence of a single participant on the experiment's results.

3 Related Work

While LEs and AICs have been compared in regard to runtime performance (Ortin et al. 2014), qualitative evaluation of either technique's effect on usability is rather scarce.

A study by Lucas et al. (2019) constitutes the only investigation specifically of Java LEs in direct comparison to AICs. The authors evaluated code snippets before and after the introduction of lambda expressions. First, they estimated readability based on LOC and cyclomatic complexity but ended up with rather inconclusive results where one model predicted significant negative effects ($p < .001$) while a similar one did not show any effect ($p = .668$). Further, they had 28 professional programmers rate the different code variants in regard to readability. This revealed a general favor for lambdas with about 50% agreeing on an improvement while the other half almost evenly shared a neutral or negative sentiment. However, some individual code migrations were predominantly perceived negatively, specifically when for-loops were replaced with lambda-based iterations. Meanwhile, consistent agreement on improved readability was found when AICs were replaced with LEs. These qualitative study results were further replicated with 15 undergraduate students.

Uesbeck et al. (2016) provided a controlled experiment on LEs. In it, they evaluated the impact of replacing while loops with LEs in C++. Three tasks were designed where iteration algorithms had to be implemented with either technique. The results from 54 subjects, a mix of students and professional developers, were analyzed regarding task completion, implementation time, and number of compilation errors while also looking at time spent in non-compiling development states and impact of experience level. The experiment was laid out in a between-subject design with one distinct subject group per technique, respectively for while-loops and lambda-based iteration. The study revealed significant results favoring loops. Task completion was significantly effected by the implementation technique ($p < .001$, $\eta_p^2 = .096$) as well as the experience level ($p < .001$, $\eta_p^2 = .401$). Regarding implementation time, programmers using lambdas ($M = 1503s$, $SD = 978s$) were on average roughly 50% slower than ones using iterators ($M = 1048s$, $SD = 887s$). This difference was found to be statistically significant ($p < .001$, $\eta_p^2 = .118$). An even bigger effect was observed for the experience level of participants ($p < .001$, $\eta_p^2 = .457$). Meanwhile, the respective task also had a small but significant influence on implementation time ($p = .016$, $\eta_p^2 = .038$).

Further, a significant difference in compilation errors was discovered between the two techniques ($p = .024$, $\eta_p^2 = .035$). Overall, the lambda group was outperformed in all aspects while experienced developers came on par with their counterparts from the loop group.

Mazinanian et al. (2017) investigated the use of LEs in a large-scale study of 241 open-source Java projects containing a total of 100,540 LEs. They analyzed how the projects evolved and applied static source code analysis while also questioning 97 developers from these projects in interviews. Specifically, Mazinanian et al. looked at which features of LEs were adopted as well as when, by whom, in which way, and for what reasons these were introduced. Among other things, they found that the functional programming concept of currying that is enabled by LEs was only sparingly applied. In approximately 90% of cases, LEs were passed as arguments to facilitate behavior parametrization. Further, significantly more LEs were found in test code with a factor of 1.15 in relation to production code ($p < .05$). Half of the analyzed projects showed a significant increase in adoption ($p < .05$) while core developers introduced more lambdas than others per LOC ($p < .05$). LEs were mostly used as a replacement for AICs or as a means to pass existing behavior to other methods while also serving as a substitute for loops and conditional structures. Meanwhile, migration to LEs mostly took place manually instead of leveraging tool support. Developers cited higher readability and terseness over AICs as a reason for choosing LEs. They also wanted to spare themselves from creating a new class for one-off behavior parametrization, e.g., in order to implement listeners. Other notable reasons were matters of consistency and migration to Java 8.

Another quantitative analysis of LEs was performed by Nielebock et al. (2019). They examined 2923 open-source projects developed with C#, C++ and Java in regard to the utilization context of LEs. Specifically, they assumed higher usage in concurrent code due to theoretical benefits of LEs in that context. However, their findings did not show that LEs are applied more frequently in concurrent contexts than in general, rather the opposite. Moreover, LEs were observed to predominantly capture surrounding variables – an approach that the authors deem as unfavorable, especially in concurrent contexts. Lastly, they investigated other use cases for LEs and found them to be used above average in user-interface and testing code as well as implementations of generic algorithms (e.g., sorting).

Additionally, efforts have been made to automatically transform Java code from using AICs towards LEs (Franklin et al. 2013; Tsantalís et al. 2017; Dantas et al. 2018). Although this shows a slightly reduced overall amount of code, such approaches generally lack empirical evidence for their necessity.

In order to summarize the previous mentioned studies, one can say that existing research is far from conclusive in regard to the effects of LEs. While Lucas et al. show a general preference for LEs specifically over AICs, their readability analysis remained inconclusive. This is consistent with various evidence deeming readability models inadequate (Scalabrino et al. 2017; Katzmarski and Koschke 2012; Fakhoury et al. 2019) and shows again the need for a human-centered rather than a theoretical approach. The only study in that vein by Uesbeck et al. indicates harmful effects of LEs. Their study makes lambdas seem like a less favorable option at least when code has to be written for iterating collections. However, that study focussed on writing new code – but it remains unclear whether the resulting code might be easier to understand. Additionally, the study by Uesbeck et al. focussed on iterating collections while the application of lambda expressions in the here described context is rather motivated by the observer design pattern, i.e. lambda expressions (in comparison to anonymous inner classes) that are passed to an object in order to be executed later in an application. From that perspective, the goal is not to compare lambdas as a new way to

integrate functional programming in Java with traditional iterative constructs, but just the possible effect of lambdas in comparison to traditional anonymous inner classes. Whether this effect actually exists or how large it is remains unclear from the literature.

4 Experiment Design

Before describing the experiment, we introduce a number of considerations that were taken into account in order to study the differences between LEs and AICs. In order to simplify the reading of this paper, we use in the following the abbreviation LE_T to describe LEs whose parameters have a declared type and LE_{NT} for lambda expressions that do not have such declarations.

4.1 Initial Considerations

The present experiment appeared in the context of studying the readability of code written with the Java Stream API. The Java Stream API permits to use LEs (with or without type annotations) or AICs for collection processing (e.g., filtering). One could argue that the Stream API should be studied with all different possible uses, but from our perspective, applying all possible treatments would increase the required effort for such an experiment too much: the more treatments, the higher the effort for a subject (for a crossover study), or the higher the required number of participants (for a between-subject RCT). Additionally, in the end we were just interested in the best way to use the Stream API: while the best technique should finally make it into the Stream API experiment, the inferior techniques should just be ignored.

Before designing an experiment, we took a number of initial considerations into account (in order to simplify the reading of the paper, we use the abbreviations C01–C09 to enumerate the considerations). We make these considerations explicit because we think it is a valuable information for readers as well as experimenters for possible follow-up studies to understand how we finally came to our experimental design.

Motivation for an N-of-1 self experiment (C01) We decided to study the possible differences between LEs and AICs independent of the Stream API. Since we furthermore wanted to use our recruited subjects solely on the Stream API experiment (and not on the preliminary experiment) and taking into account that recruitment and experiment execution for traditional RCTs is high, our goal was to run such a study with low costs – as an N-of-1 trial. And more precisely, we decided to run such a study as an N-of-1 self experiment because we did not plan to invest additional participants for the study. Consequently, the authors of the study tested themselves under all conditions. I.e., instead of studying some dozen participants with maybe some dozen data points each our goal was to study single participants but hundreds of data points each. While this motivation seemed quite conclusive to us, we were aware that such a decision has an impact on the design of the experiment.

Possible participant biases (C02) We believe that the bias of single participants is a high risk for N-of-1 studies. Hence, there is always the need to check whether a potential bias of the participants possibly influences the experiment's results. The participants for the experiment (the authors) do neither have a special interest in AICs nor in LEs. I.e., we do not benefit in any way when one or the other turns out as superior in respect to readability. The participants have quite some experience in software development and are familiar with

LEs as well as AICs, i.e., we do not assume that any special training for such an experiment was needed.

Random task generation (C03) First, there was the need that knowledge about the experiment's design should not influence the response variable for the following reason. Our goal was to follow the tradition to use time as a response variable (such as for example Stuchlik 2011; Petersen et al. 2014; Endrikat et al. 2014 etc.). While time has been nowadays widely used for performance measurements in programming, time is also a known response variable for studying readability, starting from coarse grained measurements such as time it takes to read something (see for example Gould and Grischowsky 1986) up to fine-grained reaction time measurements resulting from a reading stimulus (see for example Rudell and Hu 2010). The problem with time is that too easily novelty effects appear: when you give participants an unknown task, it typically requires additional time to read and understand such a task for the first time. However, such time is later on no longer required in order to understand the task, and a period effect appears. We concluded from that that there is a need to know the task upfront. But such knowledge must not lead to the situation that subjects benefit from it – and there must not be the situation that the concrete setting is explicitly designed by the experimenters that helps them later on to solve the tasks. We concluded from that a need to generate the concrete setting for the experiment randomly (although the concrete tasks are known upfront).

Reduction of tasks to essential differences – focus on syntax (C04) We saw the need to reduce the tasks to the essential differences between LEs and AICs. While they are very similar on a technical level, they differ much with respect to their syntax. AICs (in the easiest case) consist of a class name, a method header and a method body. LEs just have a parameter list – the method body for AICs and LEs is the same (again, except the use of `this`). Hence, the goal was to focus on these differences in the syntax.

Reduction of tasks to essential differences – ignoring return types and values (C05) There is one obvious difference between LEs and AICs that we explicitly did not want to study: an AIC has in its overriding method a declared return type, while a LE does not have it. We know already from several previous studies that type declarations have a large positive effect ((Hoppe and Hanenberg 2013; Petersen et al. 2014; Endrikat et al. 2014; Fischer and Hanenberg 2015)) for tasks where return values are used. Hence our intention was not to study anything related to the type declaration. As a consequence, we did not want to give tasks such as “complete this code for a given header” or “use the return value from the LE or AIC”. Instead, our goal was to give tasks where an LE or an AIC should be identified.

Reduction of tasks to essential differences – easy body (C06) Another question was what actually should be in the method body. Since this part is identical in LEs and AICs, the more complex the body is, the larger are probably measured deviations caused by the code (and not caused by the difference between LEs or AICs). Consequently, the body must not be too complex. On the other hand, if the body is too simple (in the simplest way: empty), we assume that the identification of the body becomes too trivial. Consequently, our goal was to have easy but not completely trivial method bodies. This also implies that we were not interested in the body's functionality. I.e., following this argumentation, our goal was not to give tasks to subjects where any algorithmic complexity in the method's bodies should influence the experiment's results.

```

1 String dog = "tongue";
2 Loss loss = new Loss();
3 loss.register(new ChangeListener<List>() {
4     @Override
5     public void next(Object hall, Long twig) {
6         System.out.println(dog);
7         hall.improve(hall);
8     }
9 });

```

Fig. 2 Example code snippet used in the experiment (code snippet no. 35)

4.2 Resulting Tasks and Hypotheses

The task design was influenced by C04–06. Our goal was to give subjects a code snippet from which they needed to identify something related to LEs or AICs. But since the code body for LEs and AICs is more or less identical, our goal was not to define tasks that were somehow related to the code body. Consequently, our goal was to define tasks that are related to the header – without referring to the return type (because the effect of type declarations was studied elsewhere).

One idea was to give a larger code base and let subjects just count the number of LEs and AICs. However, we estimated that such a code basis increases the deviation in response times. Additionally, we saw the potential problem that in a larger code base indentation potentially negatively influences AICs (whose indentation is from first glance similar to the indentation of, for example, an if statement or a loop). Although one could argue that this problem inherently exists in source code, we think that one should also take into account that this has more something to do with nowadays code formats and not with the language construct itself.

Hence, we did not follow this idea to give a larger code base – and we expect that experiments that rely on such a larger code base would reveal an even smaller effect than we found in the present paper. Instead, we came up with two different kinds of tasks for a given code snippet:¹¹

- Task 1: “*Count the number of parameters in the listener.*”
- Task 2: “*Count the number of used parameters in the listener.*”

These tasks inherently require first an understanding what a listener is and second that the code just has one listener. Hence, these tasks also restrict the possible code in the experiment.

Figure 2 illustrates one code snippet we wanted to use in the experiment (and that we actually used). The code just contains one AIC which is passed to a method `register`. The method name is used to make explicit that a listener is passed. In this code snippet two parameters are passed in the listener (parameters `hall`, `twig`) but only one parameter (`hall`) is used in the listener. Another used variable (`dog`) is declared outside the listener.

We think that is is quite likely that the questions depend on different factors that need to be taken into account and that the effect of the kind of listener has a different effect on the dependent variable.

- **Parameters in Listener (Task 1):** For task 1 we believe that the core problem is the identification of the listener. We think that it is harder to identify the listener method

¹¹The here described single sentences are what was given in the experiment.

in an AIC than the lambda expression because the listener creation in an AIC possibly distracts the participant from those lines where the listener method appears. Next, we think that the number of parameters makes a difference: counting 3 parameters takes more time than counting 1 parameter. Additionally, we think that the type annotations possibly distract the counting process. On the other hand, we do not think that the length of the listener body influences the counting of the parameters because we assume that participants in the experiment start scanning the code from top to bottom and stop when the listener appears.

- **Used Parameters in Listener (Task 2):** For task 2, we think that (again) the number of parameters influences the reaction time (just because we already assume that identifying the number of parameters depends on the number of parameters). But we think that the reading of the body in order to see what variables are used requires more time and causes additional deviation in the dependent variable. If for example three variables appear in the listener header, these variables need to be kept in mind while the body is read. And we think it is possible that participants, while reading the body, forget what the variables were and need to read them again. We would assume that the larger the body of a listener is, the higher is the probability that such a situation happens. This possible situation would increase the deviation in the measurements and by that causes an obfuscation of the main factor (in case the main factor exists). Additionally, we need to take the special situation into account where the listener has zero parameters. In that case we think that the reader of the code would not take a look into the body of the code because the correct answer can be given without knowing anything about the listener body.

From these considerations we extract the following hypotheses for the first task:

- **H_{0countPs}:** There is no difference in time required for identifying the number of parameters in a listener written as an AIC, LE_{NT}, or LE_T.
- **H_{0countPsI}:** There is no interaction between the number of parameters of a listener written as an AIC, LE_{NT}, or LE_T and the time required for identifying this number.

For the second task, we extract the following hypothesis:

- **H_{0usedPs}:** There is no difference in time required for identifying the number of used parameters in a listener written as an AIC, LE_{NT} or LE_T.
- **H_{0usedPsI}:** There is no interaction between the number of used parameters of a listener written as an AIC, LE_{NT} or LE_T and the time required for identifying this number.

Hence, we assume that the dependent variable depends on different factors: for task 1 we take the number of parameters into account, for task 2 we take the number of used parameters into account. In the following, we use the term *listenerType* for the variable that describes the kind of listener, *numParams* for the number of parameters of a listener, and *numUsedParameters* for the number of used parameters of a listener.

4.3 Initial Considerations for the Code Generator

Now that the tasks were known and a first example (Fig. 2) was available, additional considerations for the code generation were made.

Differences in positions of area of interest (C07) Both tasks require the subject to first identify the position in the code where the parameters are declared. If the LEs and AICs

always appear between two tasks at the same position on the screen, it is plausible that hardly any difference can be detected: In case it turns out that AICs are harder to read (what we assume, because the AIC code is much more verbose than the code for a LE), this probably cannot be detected that way if the subject has his eyes already directed to the right place on the screen. Thus, in case the additional syntax elements in AICs decrease the readability, this possible negative effect would be compensated if AICs or LEs appear always at the same position on the screen, i.e., if the areas of interest always appear at the same position. One alternative to overcome this problem is, for example, to position code snippets freely on the screen.¹² However, we felt that this approach would not be appropriate because free positioning on the screen also means that time gets lost for identifying where on the screen the code snippet is - even before identifying the area of interest. Additionally, we think a free positioned code snippets is rather artificial because developers are rather used to finding the source code at relatively similar positions on the screen. Instead of free positioning we decided to vary the position of the area of interest by varying the lines of code that appear before this area.

Variability in generated non-listener code (C08) We already argued in C06 that the code bodies of the lambda expression or the anonymous inner classes should not be complex (and that it probably does not play a role for the first task). For the same reason we argue that the rest of the code should not be too complex: in case the rest of the code contains complex algorithmic relationships, readers would probably spent too much time on that – as a consequence, we expect a larger deviation caused by the code.

Readable identifiers (C09) Because of the existing evidence on the readability of different kinds of identifiers (see for example (Lawrie et al. 2006; Binkley et al. 2009; Hofmeister et al. 2019)), the generated code should consist of non-artificial identifiers, i.e., the identifiers in the code should be natural words. Otherwise, time gets lost for searching the identifiers in the code and the resulting deviation reduces the chance to measure the effects that we intend to measure.

Handling of the `@Override` annotation in AICs (C10) One probably minor point requires some attention: the handling of the annotation `@Override`. In Java, this annotation is not required. But it is quite common that IDEs automatically generated it when an AIC overrides a method¹³ – hence, it is quite common to see this annotation in ordinary source code. It seems plausible that this annotation, which requires an additional line of code, increases the reading time of a code snippet. At that point we decided to keep the annotation in the code in order to have a more realistic representation of AICs.

4.4 Code Generator and Generated Code

Taking the previous considerations into account, we implemented a corresponding code generator. Figure 3 illustrates code examples from the generator (one example for each treatment of the independent variable listener type): the AIC with two parameters is the code snippet T1#14 from the code base, the LE_T is the T1#66 from the code base, the LE_{NT} example is T1#1 from the code base.

¹²Such an approach that was for example chosen in the experiment by Binkley et al. where identifiers appeared at arbitrary positions on the screen (Binkley et al. 2009).

¹³This done by the IDE Eclipse (<https://www.eclipse.org/>) as well as IntelliJ (<https://www.jetbrains.com/idea/>).

```

1  Rate rate = new Rate();
2  rate.rush(rate, rate);
3  String mask = "committee";
4  String oranges = "fruit";
5  rate.register(new DeleteListener<Long>() {
6      @Override
7      public void next(Integer seed, Integer time) {
8          rate.peel(seed);
9      }
10 });

1  String nation = "army";
2  Exchange exchange = new Exchange();
3  exchange.register((Integer religion, Long yak) -> {
4      Instrument instrument = new Instrument();
5      String bath = "amount";
6      Friend friend = new Friend();
7  });

1  String range = "plants";
2  Squirrel squirrel = new Squirrel();
3  squirrel.register((rail, smoke, swing) -> {
4      String snow = "army";
5      rail.suggest(squirrel, range, rail);
6      String decision = "yam";
7  });

```

Fig. 3 Example code snippets from the generator for the first task: (a) AIC with three parameters (T1#14), (b) LE_T with two parameters (T1#66), (c) LE_{NT} with three parameters (T1#1)

The first lines of the code are simple variable declarations and definitions or simple printouts. The code generator creates between 0 and 5 of such definition lines. In order to keep the code as simple as possible, the assigned values in the declarations are just literals: in case a string is assigned it just consists of word chosen from a dictionary. Additionally, an object is created where the object type corresponds to the variable name. This definition appears either in the first or the last line of the definition lines. This object is used in the following line as a target object for the listener: on this object the method `register` is invoked with the listener as a parameter. The listener is either an AIC, a LE_{NT} , or a LE_T . The technique is randomly chosen – we only guarantee that for each technique the total number of code snippets is the same (in our case 200 code snippets per technique and per task).

The AIC consists of a new listener creation where the name of the target class makes again explicit that it is a listener, such as `Listener`, `CreateListener`, `UpdateListener` (again, the name is randomly chosen). Since newer architectures in Java use generic types in listener code¹⁴, we also add one type parameter to the listener. Then, the AIC code has one line consisting of the `@Override` annotation, then, the rest of the code consists of the overridden method. The name of such public method is always `next` with return type `void`.

¹⁴See for example JavaFX that uses generic types in the `ChangeListener` interface, <https://docs.oracle.com/javafx/2/api/javafx/beans/value/ChangeListener.html>

The generator differs for task 1 and task 2:

- **Task 1:** The generation of the overridden method is similar between AIC, LE_{NT} , or a LE_T . For task 1, between 0 and 4 parameters are passed, for AIC and LE_T , the types or the parameters are (again) randomly chosen (among the types Object, Long, Integer, and String). The body of the listener consists of 1–7 lines of code. Each line in the body is either a variable definition, a method call, or a printout. For a method call, either variables defined within the body of the listener, or variables defined in the header of the listener, or variables defined in the beginning of the code snippets are used. A printout uses one of the previously described variables.
- **Task 2:** In order to reduce the assumed deviation caused by longer parameter lists we only generate listeners with two parameters for the second task. Furthermore, the body of the listener consists of one single line of code. This line is a method call which passes two parameters to the listener. Thus, three objects are involved: the receiver of the method and the parameters. As a consequence, 0–2 parameters from the parameter list can be used in the listener body.

Altogether, 600 code snippets were generated for task 1: 40 repetitions for each *numParams* (0–4) and each *listenerType* (i.e. $40 \times 5 \times 3 = 600$ code snippets). For task 2, we generated 50 repetitions for each *numUsedParam* (0–3) and each *listenerType* (i.e. $50 \times 4 \times 3 = 600$ code snippets). The generated code follows the common Java guidelines that types or classes start with a capital while variables start with a lowercase. Furthermore, the code followed common guidelines for indentation.

4.5 Experiment Layout and Execution

Both experiments (one for task 1 and one for task 2) have different independent variables.

- The first experiment has the independent variables *listenerType* (AIC, LE_T , and LE_{NT}) and *numParams* (0–4) which are the number of parameters. Therefore, the design follows a 3×5 layout with 40 repetitions per treatment combination. The dependent variables are the reaction time and the correctness.¹⁵
- The second experiment has the independent variables *listenerType* (AIC, LE_T , and LE_{NT}) and *numUsedParams* (0–3) which are the number of used parameters. This results in a 3×4 layout with 50 repetitions per treatment combination. The dependent variables are the reaction time and the correctness.

The reason why there is a difference in the number of repetitions per treatments combination is that our goal was to keep the number of code snippets per task constant.

4.6 Measurements and Execution

We used a desktop application to collection the data. The first 600 code snippets were used for task 1, the following 600 snippets for task 2. Since we are measuring reaction times, we intentionally did not shuffle the tasks in order to not disturb the concentration of the

¹⁵Essentially, two treatment combinations are the same: if the number of parameters is zero, there is no difference between the code snippets for LE_T , and LE_{NT} because if no parameter exists, there are no type annotations that make the difference between LE_T and LE_{NT} . Being aware of this problem, we still keep this in order to have a complete 3×5 layout.

Table 1 Ages, current job and industrial background of the used participants

	Participants							
	1 author 1	2 author 2	3	4	5	6	7	8
Age	46	25	39	25	24	33	26	25
Current Job	Lecturer	Software Developer	Software Developer	Software Developer	Student/Software Developer	PhD student	Software Developer	Software Developer
Industry Experience	–	3 year	15 years	5 years	2 years	7 years	4 years	3 years
Applied Java since	1999	2016	2006	2014	2017	2007	2017	2016
Java Usage in Industry	–	3 years	5 years	5 years	–	–	4 years	3 years

Participant 1 was used for the original N-of-1 study, participant 2 for a replication of the study. Additionally, the experiment was replicated 6 times on participants not involved in the present project (participants 3–8)

participants. Within a task, the code snippets (i.e., the treatment combinations) were randomly shuffled.

In the beginning of each question block the question is shown once, then the participant has to press the space bar to start the data collection. Afterwards, a code snippet is shown (the measurement for the reaction time begins at that point) and the participant has to press one of the keys 0–4. Once a button is pressed, the measurement stops (respectively, it is measured whether the result was correct) and the participant has to press return to go on with the next code snippet.¹⁶

We assumed that it is necessary to run the experiment in a quiet environment where the participant is highly concentrated. Hence, we permitted a participant to chose freely between two code snippets to take a break. Since reaction time is measured, it is necessary that a participant is willing to give correct answers: the experiment can be compromised if a participant just pushed the buttons as fast as possible without trying to answer the questions correctly. Since we assume that responses to the questions can be given within few seconds, we advised the participant to have his fingers on the keyboard's numpad in order to enter the result. Otherwise additional time would be required to put a finger on a key to respond to a question. We assume that the resulting deviation could possibly hide the main effect (in case a main effect exists).

4.7 Participants in Original Study and Replications

The experiment was originally executed on the first author or the paper, i.e., the N-of-1 self trial was executed once on the first author. A first repetition of the experiment was executed on the second author of the paper. In order to check, whether the results of the study where possibly influenced by the bias of the authors, the experiment was replicated 6 times more. Altogether, there are 7 replications of the original N-or-1 experiment.

¹⁶The Java-code for the application is available in the additional material of this paper.

While the choice of the second author for a replication seemed natural from our perspective, the other participants were chosen based on purposive sampling (Patton 2014) in the following way.¹⁷ The authors contacted potential participants and asked them whether they are willing to volunteer. The requirements were that the participant should have at least 2 years of industrial experience as a software developer. Additionally, the participant should be familiar with the programming language Java and its LEs and AICs but we did not see a need that Java was part of a participant's industrial experience. Additionally, the candidates were informed that the participation takes about 2 hours.

Table 1 summarizes the 8 participants that took part in the experiment (including the authors). The first author has no industrial experience, participant 5 was student when he participated (but worked half-time for industry since more than 2 years), participant 6 was a PhD student with previous industrial experience as a software developer. All 3 previously mentioned participants had no industrial experience with Java. All other participants (including the second author) were software developers with 3–15 years of experience with at least 3 years of industrial experience with Java.

5 Analysis and Results of the Original Study and its First Replication

The results of the following analysis were computed using SPSS v26. All time measurements for the dependent variable in the following section are given in milliseconds.

Before analyzing the data, we first study whether the code from the code generator is appropriate for the study, i.e., we check whether there are possible confounding factors introduced by the random code generation (Section 5.2). Then, we study whether there are time-dependencies in the measurements. Based on the results, we run the analysis.

For running the analysis, we initially execute the experiment on a first subject and afterwards we double check the results by running the experiment on the second. We will present the analysis task by task (and not subject by subject) in order to simplify the reading of the paper. For the same reason, we report here already the results of both subjects, although the data from the second subject was collected after the first subject and the second subject's data was used to double-check the results of the first subject.

5.1 Comparability of Generated Code – Checking for Confounding Factors

The code snippets were randomly created. However, it is still necessary to check whether the random generator accidentally introduces large differences between individual snippets and in that way causes an overreaching of one of the main treatments (*listenerType*) in the experiment. What we do is to speculate about the possible confounding factors in the code snippets and use them in an ANOVA as an independent variable and run a Tukey-HSD posthoc test in order to check whether significant differences can be found.

Before checking for other possible confounding factors, we first check whether the task order possibly gives one of the treatment combinations a benefit such that it appears earlier in the experiment. In order to check for this possible effect, we divided the code snippets for each task into three equal parts of 200 snippets. Thus, the first 200 snippets are in the first

¹⁷Purposive sampling is a non-probability sampling technique that is quite often applied in software engineering. The study by Baltes and Ralph (2020) revealed that about 70% of empirical studies in software engineering use purposive sampling.

section for task 1, snippets 201–400 in the second section for task 1, and snippets 401–600 in the third section.

For the first task we run an ANOVA with the dependent variable *section* (with treatments 1–3) as well as the independent variables *listenerType* (treatments AIC, LE_T and LE_{NT}) and *numParams* (treatments 0–4). For the second task we run an ANOVA with the dependent variable *section* and the independent variables *listenerType* and *numUsedParams* (treatments 0–3). For the first task, there is a small (not significant) tendency of an influence of *listenerType* but the effect is very low ($p = .064$, $\eta_p^2 = .009$). The factor *numParams* does not show an effect ($p = .424$, $\eta_p^2 = .007$). For the second task neither *listenerType* ($p = .627$, $\eta_p^2 = .002$) nor *numUsedParams* ($p = .534$, $\eta_p^2 = .004$) reveals potential problematic results. Since there is some tendency for *listenerType* in the first task, we ran a posthoc test which reveals a small (not significant) tendency that AICs appear earlier than LE_T but no difference between the other treatment combinations ($p_{AIC-LE_T} = .082$, $p_{AIC-LE_{NT}} = .140$, $p_{LE_T-LE_{NT}} = .967$). Because of that and taking the low effect sizes into account we did not consider the situation as problematic and considered the ordering of the treatment combinations as fair.

We think it is plausible that the readability of code snippets depends on the lines of code (LOC). Consequently we first check whether the LOC are equally distributed across snippets. Table 2 contains the descriptive statistics for LOC. From that, it can be seen that LOC are largest for AIC, followed by LE_T and LE_{NT} . This difference is not accidental but the result of the previous considerations: an AIC requires inherently three lines of code more than a comparable LE because of the new operator, the `@override` annotation, and the additional closing braces for the listener method. Between LE_T and LE_{NT} there must not be a difference in LOC. Taking this into account, we subtract 3 from LOC for the AIC and compare it with the LOC for LE_T and LE_{NT} for both tasks and call the resulting number $LOC_{corrected}$.

For the first task, the kind of listener had no significant effect on $LOC_{corrected}$ ($p = .917$) and the posthoc test did not find any difference in $LOC_{corrected}$ for *listenerType* ($p_{AIC-LE_T} = .99$, $p_{AIC-LE_{NT}} = .926$, $p_{LE_T-LE_{NT}} = .937$) either. Repeating the same test for the second task reveals also no significant differences in the ANOVA ($p = .347$) and the posthoc test ($p_{AIC-LE_T} = .440$, $p_{AIC-LE_{NT}} = .308$, $p_{LE_T-LE_{NT}} = .968$).

While it is plausible that LOC play a role, it is also plausible that the number of declarations in the beginning of the code snippets have an effect: in case the reader of a snippet reads the code line by line (which we do not assume, but which is possible) and stops when the listener starts, a difference in the number of declarations in the beginning plays

Table 2 Descriptive statistics for the 1,200 generated code snippets

	Tasks					
	Task 1 (N=600)			Task 2 (N=600)		
	AIC N=200	LE_T N=200	LE_{NT} N=200	AIC N=200	LE_T N=200	LE_{NT} N=200
LOC						
min	7	5	4	7	4	4
max	15	12	12	12	9	8
mean	10.56	7.55	7.50	9.095	6.22	6.25

a role. Hence, we compare the number of LOC before the listener (and call the variable LOC_{before}). An ANOVA with the dependent variable LOC_{before} for the first task does not reveal an effect of *listenerType* ($p = .87$), neither does the posthoc test (all p -values for the pairwise comparisons are above or equal .866). For the second task the *listenerType* has no effect ($p = .347$) and the pairwise comparisons reveal p -values above or equal .363 for all pairs.

For the first task, the code snippets contain a random number of LOC in the listener body (we call that LOC_{body}). Although we assume that these are not relevant for the first task, we cannot exclude that subjects might spend more time on the task if LOC_{body} is larger. However, the ANOVA reveals no significant effect of *listenerType* on LOC_{body} ($p = .91$, p -values above .925 for all pairwise comparisons).

Hence, we conclude that the generated code snippets are a fair basis for a comparison of the factors *listenerType* and for task 1 for the *numParams* as well as for task 2 for *numUsedParams*.

5.2 Testing for Time-Dependencies in Measurements

Before analyzing the data, we wanted to check for possible time-dependent effects (such as learning effects or fatigue effects) because in case such effects appeared they need to be taken into account in the main analysis.

In a first step, we divided all code snippets from task 1 in the order as they appeared for the subject into three parts, each consisting of 200 snippets and call this variable *section*. Then, we ran an ANOVA to measure the effect of *section* on the measured time.

For task 1, the first did not reveal a time effect ($p = .259$, $\eta_p^2 < .004$) but we detected such an effect for the second subject ($p < .001$, $\eta_p^2 < .134$). We see a problem in the second subject, not only because the variable *section* is significant but because the effect size is not neglectable. The posthoc test revealed a larger difference between the first section and the other sections ($p_{sec1-sec2} < .001$, $M_{sec1-sec2} = 139\text{ms}$, $p_{sec1-sec3} < .001$, $M_{sec1-sec3} = 179\text{ms}$, $p_{sec2-sec3} = .104$, $M_{sec2-sec3} = 40\text{ms}$). Based on these numbers, our interpretation is that for the second subject a learning effect appeared and consequently the analysis of the second subject needs to take this into account.

In order to get a more detailed understanding of this, we ran a linear regression for both subjects with time as the dependent variable and the position of the code snippet in the experiment sequence as an independent variable. For subject 1, the snippet position was significant but the effect was very low ($p = .035$, $R^2 = .007$). For subject 2, the snippet was significant as well but with a larger R^2 ($p < .001$, $R^2 = .125$).

We repeat the same analysis for task 2 by dividing the code snippets into three sections. The resulting analysis also reveals a time-dependency for subject 1 but the effect is weak ($p = .042$, $\eta_p^2 < .011$). The posthoc test reveals a significant difference between the first and the third section ($p_{sec1-sec2} = .164$; $M_{sec1-sec2} = -81\text{ms}$; $p_{sec1-sec3} = .041$; $M_{sec1-sec3} = 109\text{ms}$; $p_{sec2-sec3} = .816$; $M_{sec2-sec3} = -27\text{ms}$), but this time the effect is negative, indicating a (low) fatigue effect for subject 1. For the second subject, the effects are (again) stronger ($p < .001$, $\eta_p^2 < .095$) while a (positive) learning effect appeared ($p_{sec1-sec2} < .001$, $M_{sec1-sec2} = 250\text{ms}$; $p_{sec1-sec3} < .001$; $M_{sec1-sec3} = 254\text{ms}$; $p_{sec2-sec3} = .991$; $M_{sec2-sec3} = 5\text{ms}$). Running another linear regression shows results comparable to the ANOVA for subject 1 ($p = .006$, $R^2 = .013$) as well as for subject 2 ($p < .001$, $R^2 = .093$).

Fig. 4 Boxplot for task 1 (subject 1) illustrating the reaction times (in milliseconds) per number of parameter for the different listener types. For reasons of readability we removed the outliers from the boxplots

Hence, we conclude for the main analysis the need to consider the time-dependency as well: for subject 2 the positive learning effect in both tasks, for subject 1 a negative fatigue effect in task 2.

5.3 Analysis of Task 1

Figure 4 illustrates the measured results for the first subject for tasks 1. There seem to be differences between AIC, LE_T , and LE_{NT} : in all cases the medians of the LEs are below the median of the AIC and in most cases LE_T has a higher median than LE_{NT} (Table 3).

We analyzed the data for task 1 using an ANOVA with the dependent variable reaction time and the independent variables *listenerType* and *numParams*. However, due to the effect of the variable section, we also used section as an additional independent variable (see Table 4).

The results reveal for the first subject *listenerType* and *numParams* as significant factors with rather small effects ($\eta_p^2 \leq .12$). For the first subject the interaction between both

Table 3 Descriptive Statistics for the treatments of the variable listenerType: N, arithmetic mean, and standard deviation (in milliseconds)

			Subject 1		Subject 2	
	listenerType	N	Mean	SD	Mean	SD
Task 1	AIC	200	1,185	360	941	233
	LE_T	200	1,004	256	854	194
	LE_{NT}	200	984	230	843	188
Task 2	AIC	200	2,706	452	2,386	364
	LE_T	200	2,738	492	2,396	396
	LE_{NT}	200	2,585	382	2,268	386

Table 4 Result of the ANOVA for both tasks on the dependent variable reaction time and the result of the χ^2 -test on the dependent variable correctness

Reaction Time	ANOVA	df	Subject 1		Subject 2	
			p	effect	p	effect
Task 1	ListenerType(T)	2	<.001***	.115	<.001***	.049
	NumParams(N)	4	<.001***	.099	<.001***	.12
	Section(S)	2	.232	.005	<.001***	.149
	T*N	8	.109	.023	.078	.025
	T*S	4	.733	.004	.808	.003
	N*S	8	.466	.014	.574	.012
	T*N*S	16	.495	.027	.251	.034
Task 2	ListenerType(T)	2	<.001***	.023	<.001***	.029
	NumUsedPs(U)	3	<.001***	.065	<.001***	.085
	Section(S)	2	.033*	.012	<.001***	.112
	T*U	6	.974	.002	.238	.014
	T*S	4	.172	.011	.955	.001
	U*S	6	.992	.001	.206	.015
	T*U*S	12	.431	.021	.146	.030
Errors	χ^2 -Test					
Task 1		2	.169		.846	
	AIC			11		9
	LE_t			5		9
	LE_{NT}			5		7
Task 2		2	.926		.705	
	AIC			20		10
	LE_t			20		10
	LE_{NT}			18		7

The effects for the ANOVA are given as η_p^2 , the effects for the χ^2 are given as number of absolute errors

variables is not significant but approaching significance ($p = .109$). However, the effect size is very small. To check whether the results are trustworthy, we repeated the experiment with a second subject and received comparable results: again, both *ListenerType* and *numParams* are significant and the effect sizes are small. The only difference between both subjects is the significance of the factor *section*: while it is not significant for the first subject, it is significant for the second one. We already detected this time effect in Section 5.2 but the absence of an interaction effect between section and any other factor is more important. This means the subject had a measurable time effect but it did not influence the results of the other factors.

In order to analyze the difference between the treatments of *listenerType*, we ran a Tukey-HSD posthoc test. It turns out AIC requires significantly more time than LE_T as well as LE_{NT} and the mean differences between AIC and LE_{NT} are higher than for AIC and LE_T

Table 5 Tukey posthoc test on variable ListenerType. Mean difference and 95% confidence interval in milliseconds

	Subject 1			Subject 2			
	Tukey-HSD	p	Mean Diff	95% CI	p	Mean Diff	95% CI
Task 1	AIC-LE_T	<.001***	181	[117;245]	<.001***	87	[45;130]
	AIC-LE_{NT}	<.001***	236	[172;300]	<.001***	98	[56;141]
	LE_T-LE_{NT}	.112	55	[-10;119]	.882	11	[-31;53]
Task 2	AIC-LE_T	.742	-31	[-134;70]	.954	-10	[-91;71]
	AIC-LE_{NT}	.015*	236	[19;223]	.002**	118	[36;200]
	LE_T-LE_{NT}	.001**	152	[51;254]	.001**	128	[46;210]

(see Table 5). However, no differences were detected between the two different kinds of LEs.

Repeating the same analysis for the second subject reveals comparable results. Again, the pairwise comparisons reveal differences between AIC and LE_T as well as between AIC and LE_{NT} where the mean difference for the second comparison is higher. And again, no significant difference between the two kinds of LEs were measured.

To get an idea what the statistical results actually mean, we related the mean times for LE_{NT} to the mean times for AIC. For the first subject AIC took about 20.5% longer than LE_{NT}, for the second subject AIC took about 11.6% more time than LE_{NT}.

In order to check whether the measured difference could be caused by errors made by each subject, we ran a χ^2 -test on the number of errors. While subject 1 did in comparison to subject 2 less errors in task 1, no significant differences between the number of errors for the AIC, LE_T, and LE_{NT} were detected.

Altogether, the analysis of task 1 leads to a rejection of $H0_{countPs}$ and an acceptance of $H0_{countPsI}$ (although there are tendencies towards $H0_{countPsI}$).

5.4 Analysis of Task 2

Figure 5 illustrates the measured results for the first subject for task 2. However, the differences do not seem to be too clear: while in all cases LE_{NT} has a lower reaction time than AIC, the (possible) differences between AIC and LE_T are hard to detect visually.

We repeat the same analysis we did for task 1 now for task 2, i.e., we use an ANOVA with the dependent variable reaction time and the independent variables *listenerType*, *numUsedParams* and *section* (see Table 4).

The results are quite comparable to the analysis for task 1. For the first subject both *listenerType* and *numUsedParams* are significant with rather small effects ($\eta_p^2 < .12$). In contrast to task 1, *section* was significant as well which again confirms the found effect. But, comparable to the situation we had for subject 1 in task 1, there is no significant interaction between *section* and any other factor. This means that subject 1 had a measurable time effect which did not influence the results of the other factors. Therefore, the differences between *listenerType* are independent of the time effect. Again, we ran a Tukey-HSD posthoc test. It turns out AIC requires significantly more time than LE_{NT} while there is no significant

Fig. 5 Boxplot for task 2 (subject 1) illustrating the reaction times (in milliseconds) per number of parameter for the different listener types. For reasons of readability we removed the outliers from the boxplots

difference between AIC and LE_T . This time (in contrast to task 1), there is a significant difference between LE_T and LE_{NT} .

Again, we repeated the experiment with a second subject and received comparable results: both *listenerType* and *numUsedParams* are significant with rather small effect sizes. Once more, *section* is significant (this time stronger than the other effects) but there is no interaction effect with any other factor. The posthoc test also reveals that AIC required more time than LE_{NT} but not more time than LE_T . And again, a significant difference between LE_T - LE_{NT} was detected.

Once more, to get an impression of the effect, we relate the mean reaction time for AIC to LE_{NT} . For the first subject AIC took 4.7% more time, the second subject required 5,2% more time for AIC.

We run another χ^2 -test on the number of errors. While subject 1 had, in comparison to subject 2, much more errors in task 2, no significant differences between the number of errors for the AIC, LE_T and LE_{NT} were detected.

Altogether, the analysis of task 2 leads to a rejection of $H_{0_{usedPs}}$ and an acceptance of $H_{0_{usedPsI}}$.

6 Results of Six Additional Replications and Comparison to Original Study

One might argue that the gathered results are just the result of the authors' biases. In order to test whether this was possibly the case, we did six additional repetitions with participants not involved in the project. Although we think that each individual replication deserves to be analyzed in detail, we think that it would unnecessarily increase the length of the present

paper. Because of that, we only give an overview of all replications and mention parallels and differences to the original study (and its first replication) and present the combined results of both tasks (and both dependent variables).

Table 6 shows the results of the ANOVA on the reaction times for each replication for both tasks, as well as the results of the χ^2 -test on the correctness for each task. The results of the post-hoc tests for each replication can be found in Table 7.

The replications reveal differences in the carry-over effects (*variable section*). For task 1, subjects 4, 5, and 7 did not have a measurable carry-over effect. All other subjects had a measurable learning effect. For task 2, only subject 8 had no measurable carry-over effect. And while subject 7 had a measurable fatigue effect, the other subjects had a measurable learning effect. We observed a similar phenomenon in the first replication in contrast to the initial study where different carry-over effects occurred. But again (in parallel to the original study), there are no measurable interaction effects between *section* and *listenerType* which indicates that the presence of the carry-over did not influence the experiment results in an undesirable way.

For task 1, *listenerType* is a significant factor ($p < .001$) with effect sizes between $\eta_p^2 = .032$ and $\eta_p^2 = .118$ (in the initial study the effect sizes were between $\eta_p^2 = .049$ and $\eta_p^2 = .115$). With respect to interactions between the *listenerType* and *numParams*, 2 subjects (6 and 8) showed such an interaction, while the other subjects did not.

Hence, we reject $H0_{countPs}$ but we cannot reject $H0_{countPsI}$ although there are tendencies towards a rejection of $H0_{countPsI}$ (which corresponds to the result of the original study).

With respect to the pairwise comparisons in the post-hoc test, the replications show small differences. While in the original study AIC was different to LE_T and LE_{NT} for task 1 (but no differences between AIC and LE_T were detected in task 2), subjects 4 and 7 did not reveal a difference between AIC and LE_T . But still, differences between AIC and LE_{NT} are shown in all cases which leads to the conclusion that LE leads to an improvement of readability. The ratios $\frac{M_{AIC}}{M_{LE_{NT}}}$ vary between 1.13 and 1.35, i.e., AIC took between 13% and 35% more time (in the original study, this was between 12% and 21%).

With respect to correctness for task 1, we found in no replication an indication for differences between AIC, LE_T and LE_{NT} (which corresponds to the original study). But what's noteworthy is that one subject (subject 4) had a large error rate (with 37 errors in total).

For task 2, we get a comparable picture. First, *listenerType* is significant with $p < .001$ for all replications except for subject 3 (which still revealed a significant interaction with $p = .002$) and the effect sizes are small and vary between $\eta_p^2 = .022$ (subject 2) and $\eta_p^2 = .083$ (subject 8) – the original study revealed effect sizes between $\eta_p^2 = .023$ and $\eta_p^2 = .029$). With respect to the interaction, no replication revealed an indicator for an interaction between *listenerType* and *numUsedParams*.

Hence, we reject $H0_{usedPs}$ and accept $H0_{usedPsI}$ (which corresponds to the results of the initial study).

Again, no differences in the correctness are measured in the replications (which corresponds to the original study). The replications show a slightly higher effect in terms of the ratios $\frac{M_{AIC}}{M_{LE_{NT}}}$ in comparison to the original study: while in the original study AIC were just 5% slower than LE_{NT} , the replications revealed slow downs between 8% and 20%.

Table 6 Result of the ANOVA for both tasks on the dependent variable reaction time and the result of the χ^2 -test on the dependent variable correctness on 6 additional participants. The effects for the ANOVA are given as η_p^2 , the effects for the χ^2 are given as number of absolute errors

Time	Subject 3			Subject 4			Subject 5			Subject 6			Subject 7			Subject 8		
	df	p	effect	p	effect	p	p	effect	p	p	effect	p	p	effect	p	p	effect	effect
Task 1	ListenerType(T)	2	<.001***	.032	<.001***	.070	<.001***	.130	<.001***	.037	<.001***	.039	<.001***	.039	<.001***	.118		
	NumParams(N)	4	.002**	.029	<.001***	.137	<.001***	.046	<.001***	.015	.030*	.019	<.001***	.019	<.001***	.187		
	Section(S)	2	<.001***	.268	.182	.006	.432	.003	<.001***	.180	.065	.010	<.001***	.010	<.001***	.100		
	T*N	8	.417	.015	.234	.019	.068	.026	.006**	.038	.338	.016	.046*	.016	.046*	.028		
	T*S	4	.231	.010	.138	.012	.940	.001	.972	.001	.077	.015	.805	.015	.805	.003		
	N*S	8	.550	.012	.101	.024	.207	.019	.114	.023	.052	.013	.483	.013	.483	.013		
	T*N*S	16	.365	.030	.391	.032	.340	.031	.673	.023	.080	.043	.240	.043	.240	.034		
	ListenerType(T)	2	.002**	.022	<.001***	.028	<.001***	.065	<.001***	.061	<.001***	.057	<.001***	.057	<.001***	.083		
	NumUsedPs(U)	3	.001***	.028	<.001***	.053	<.001***	.046	<.001***	.043	<.001***	.126	.022*	.126	.022*	.017		
	Section(S)	2	<.001***	.084	<.001***	.076	<.001***	.032	<.001***	.052	.001**	.024	.062	.024	.062	.010		
Task 2	T*U	6	.697	.007	.492	.010	.613	.008	.122	.018	.240	.014	.664	.014	.664	.007		
	T*S	4	.187	.011	.456	.006	.702	.004	.522	.006	.338	.008	.100	.008	.100	.014		
	U*S	6	.090	.019	.485	.010	.191	.015	.091	.019	.128	.017	.310	.017	.310	.012		
	T*U*S	12	.173	.028	.575	.018	.527	.019	.281	.025	.290	.025	.926	.025	.926	.010		
	Errors (χ^2)																	
	AIC	2	.446		.145		.172		.881		.087		.134		.134			
	LE_I			1		7		1		2		2		2		2		
	LE_I			3		16		4		3		8		8		0		
	LE_{NT}			1		14		6		3		3		3		0		
	LE_{NT}	2	.310		.972		.065		.667		.696		.193		.193			
Task 2	AIC			8		12		12		13		7		7		1		
	LE_I			11		12		9		9		5		5		2		
	LE_{NT}			5		13		3		12		8		8		5		

Table 7 Tukey posthoc test on variable ListenerType and ratios $\frac{M_{AIC}}{M_{LENT}}$ for both tasks. The 95% confidence interval and ratio is given in milliseconds

	Subject 3		Subject 4		Subject 5		Subject 6		Subject 7		Subject 8	
	Tukey-HSD	p	95% CI	p	95% CI	p	95% CI	p	95% CI	p	95% CI	p
Task 1	AIC-LE_T	.047*	[1;180]	.085	[-8;165]	<.001***	[209;427]	.001**	[44;202]	1	[-135;137]	.003**
	AIC-LE_{NT}	<.001***	[107;286]	<.001***	[161;334]	<.001***	[305;523]	<.001***	[92;250]	<.001***	[129;401]	<.001***
	LE_T-LE_{NT}	.016*	[16;196]	<.001***	[82;255]	.099	[-13;205]	.323	[-31;137]	<.001**	[128;400]	<.001***
Task 2	AIC-LE_T	.996	[-190;204]	.123	[-30;341]	.610	[-85;2001]	.037*	[6;246]	.365	[-150;558]	.867
	AIC-LE_{NT}	.003**	[81;475]	<.001***	[149;521]	<.001***	[234;520]	<.001***	[208;449]	<.001***	[512;1220]	<.001***
	LE_T-LE_{NT}	.004**	[74;468]	.060	[-6;366]	<.001***	[177;462]	<.001***	[82;323]	<.001***	[308;1016]	<.001***
Subject												
Ratio $\frac{M_{AIC}}{M_{LENT}}$	3	4	5	6	7	8						
Task 1	1.720	1.332	1.590	1.466	1.731	3.430						
	1.523	1.085	1.177	1.295	1.466	2.579						
	3.769	4.190	3.192	3.224	5.466	6.147						
Task 2	3.491	3.854	2.815	2.896	4.601	5.130						

7 Threats to Validity

As every experiment, this experiment potentially suffers from a number of threats that should be documented.

Generalizability of Code Snippets (Part 1) Obviously, the generated code snippets are not representative. The code just consists of fragments that could be found in the body of a method. Neither multiple classes nor multiple methods, or instance variables are used. Furthermore, language constructs such as loops, conditionals, etc., are not applied. And finally, the code does not represent any algorithmic code where it is necessary to understand at what point what variable changes the results of the algorithm. However, this non-generalizability of the code snippets was intentional in order to control as many variables as possible.

Generalizability of Code Snippets (Part 2) The code snippets use AICs and LEs in listener implementations. We see other alternatives such as their usage as comparators, i.e., AICs and LEs could be passed as additional parameters to methods (i.e., the target method would receive more than one parameter). We believe that such usage would probably increase the deviation in the reaction time as well because it probably depends on how many parameters are passed to a method and at what position the AIC or LE is passed.

Generalizability of Tasks The tasks cannot be considered as ordinary development or readability tasks. They are intentionally designed to focus on the syntactic differences between AICs and LEs. Hence, saying that LEs improve the readability of code in general up to 33% is from our perspective an over-interpretation of the experimental results. Instead, we only say that the improvement in readability is achieved under the given experimental conditions – we assume that the effect in real-world code is much less.

Identification and Handling of Time-Dependency We emphasized the need to study the possible time-dependency of the resulting measurements. We did that by dividing the code snippets into three chunks of the same size and ran an ANOVA with the number of chunks as an independent variable. The choice of dividing it into three parts (and not, e.g., just two) is more or less arbitrary. In order to soften this possible threat, we also ran a regression analysis which revealed comparable results. Still, we need to consider that under other conditions the regression analysis could have revealed fundamentally different results. Consequently, the handling of the parameter *section* in an ANOVA could potentially lead to problems in some situations. Probably, in such situations the application of regressions instead of ANOVAs would be helpful.

Possible Problem in Experiment Protocol The experiment requires to be executed in a quiet environment where the participants can be highly concentrated – and that they needed to have their fingers already on the keyboard before running the experiment. We think that this rather vague description has the potential problem that other people can easily run the experiment with different results – just by having a more relaxed understanding of what quiet environment or being highly concentrated actually means. Going forward, we think it is necessary to come up with experiments that depend on reaction time with clear and measurable conditions to decide whether or not data collected from a subject should be considered as valid data. First and rather simple ideas could be to take the number of errors into account or to define reaction times that have to be achieved. For example, if a subject requires 10 or more seconds to identify a listener in the experiment, it is plausible that the

results from this subject should not be considered in the experiment because the reaction time is larger than what can be expected.

Generalizability of Results of N-of-1 trials The presented study uses an N-of-1 trial in order to study readability and it is commonly known and accepted that the evidence from N-of-1 trials is less than the evidence gathered from a RTC with multiple subjects. Still, one needs to consider that we repeated the trials with comparable results.

8 Discussion

The present work consists of two core topics: the readability of lambda expressions (which is a technical contribution) and the application of N-of-1 trials in software science (which is a methodological contribution). Hence, we discuss the present work from both perspectives.

8.1 Readability of Lambda Expressions

Our approach was to run an N-of-1 trial in order to study the readability of anonymous inner classes (AICs) and lambda expressions (LEs) in the programming language Java – for LEs we distinguished between those ones with type declaration (LE_T) and those without (LE_{NT}). The comparison was based on the use of AICs and LEs as listeners. We generated a large number of code snippets (600 code snippets per task) and defined two tasks. The tasks were intentionally simple and focused on the syntactical differences between AICs and LEs. The main dependent variable was reaction time.

Before running the experiment we first checked whether the ordering of the code snippets was fair. The result was that there was a (non-significant) indicator for a difference between the treatments AIC and LE_T . But the effect was so small that we decided not to change the ordering. Next, we speculated about the possible effect of LOC per snippet and the number of declarations before the listener. We checked whether there are imbalances in the code snippets with respect to these parameters but we did not find any. From that we concluded the code snippets to be a fair basis for a comparison.

Next, we tested for time-dependencies in the measurements and it turned out that both subjects had a time dependency: the first subject became slower (only in task 2) while the second subject became faster. Our interpretation of this is a fatigue effect for the first and a learning effect for the second subject. But the most essential information we get from this is that we need to take the time-dependency into account as a separate factor.

Then, we ran the experiment on a single subject (an author of this paper). The result was that we found differences between AICs, LE_T , and LE_{NT} in both tasks. For the first task the reaction time of both LEs was faster than for AICs while for the second task only a difference was found between AIC and LE_{NT} . In both cases, the effect size was relatively small. In order to check whether the results are trustworthy, we executed the experiment on a second subject (again, author of this paper). The results were comparable: the two factors that were significant for the first subject were significant for the second subject as well and the effect sizes were comparable (and small). Finally, we executed the experiment on 6 additional subjects individually. Altogether, for the first task, the use of a lambda expression without type annotations reduced the reaction time between 12% and 35% compared to AICs.

One might argue that the measured differences, especially for the first task, are quite high which justifies the execution of larger RCT with multiple subjects. However, this is

not our interpretation. From our perspective, the first task is a very simple task that hardly requires to take anything else into account except the syntactical characteristics of parameter definitions in AIC, LE_T and LE_{NT} . The second task is already cognitively more demanding because parameter names need to be read and identified in a method call. And although the experimental setup already controls a number of confounding factors (single line of code in listener body, fixed number of participating objects in the method call) the effect is already much smaller (between 5% and 20%). We think that if we relax the experimental conditions slightly more (e.g., by increasing the lines of code in the body) the positive reading effect of lambda expressions will no longer become visible. Additionally, we need to consider that the present paper is just about readability – and not understandability nor writability of the code. So, we think that as soon as mentally more demanding tasks are given that also require understanding the code, the positive effect of lambda expressions will be gone.

What's noteworthy is the role of the time-dependency in for the subjects: the carry-over effects were different in different repetitions. Actually, we do not think that this is a problem is the experiment but rather a phenomenon one has to accept in crossover trials in general.

To get a more complete picture of the effect of LEs in comparison to AICs we think it is necessary to regard that the absence of type annotations has probably a negative effect on the usability. Some experiments have shown a negative effect of dynamic type systems in comparison to static type systems (see Endrikat et al. 2014; Fischer and Hanenberg 2015; Petersen et al. 2014). And while lambda expressions in Java are still statically typed, they do not syntactically reveal all type information to the developer: in all cases the return type is missing, and in case type annotations are not used for parameters, type information are syntactically not available, too. Additionally, one should note that lambda expressions might reveal a further level of complexity: at least lambda expressions have shown in comparison to ordinary iterators a negative effect (see Uesbeck et al. 2016) for writing code. Consequently, one should not overestimate the measured, positive effects in the present study because code readability is just one activity in code development.

When the present study is used, for example, to judge, whether in the IDE IntelliJ the visual representation of anonymous inner classes as lambda expressions improves the readability, the answer is yes. But the effect is so small that we do not deem this IDE feature worth mentioning (at least not, if it is not combined with other features that also reveal a positive effect on readability).

8.2 On N-of-1 Self Trials

We propose and applied in the present paper the application of N-of-1 trials because they reduce the effort to select data due to a manageable number of subjects. Even more, we propose the application of N-of-1 self trials where the researcher runs with himself as a subject an N-of-1 trial. Obviously, such an approach has a number of risks (that we will discuss later on) but we still believe that N-of-1 self trials are an important initial step before executing larger experiments run on a number of subjects. This belief is based on a relatively simple thought: why should certain treatments have an effect on a number of different subjects if they have not even an effect on the researcher himself? So, why should someone invest into a research project with a large number of subjects involved if there was not even an indicator that the treatments have an effect?

The motivation for N-of-1 trials is quite simple: first, experiments with multiple subjects suffer from the deviation problem among such subjects. This problem cannot occur in N-of-1 trials because just one single subject participates. Consequently, such experiments are

able to detect effects more easily (under the assumption that the tested subject reacts on the treatment in a comparable way as other subjects).

The large benefit of running N-of-1 trials at an early stage of experimentation is that they permit to quantify the expected effect size of a treatment – an information which is necessary in order to estimate how many subjects are required for an experiment in order to detect whether or not an effect exists. This information becomes even more necessary in more recent times where researchers think about applying the idea of trial registration (see for example (Rennie 2004)) to the field of software science¹⁸, i.e., where trials are registered upfront. If a trial is registered, a previous N-of-1 study could give an argument for the aimed sample size. And reviewers for such registered trials could rely on this information in order to judge whether a registered trial is promising or not.

But N-of-1 have risks with respect to the publication bias. From our perspective the most obvious one is the bias of those researchers who test their own invented technology. We assume that in those cases it is more likely that such experiments reveal positive effects: not because they exist but because the researcher wants them to exist. We think that in such situations N-of-1 trials could still be helpful when they are based on data collected from a subject that is not involved in the research project.

Although N-of-1 trials reduce the data collection effort, such reduction is not for free: it requires to take a number of preliminary considerations and threats into account.

First, it requires to increase the number of data points selected from a single subject. In our case, we achieve this via random code generation and a random order in which the code was given to the subjects – and we personally believe that this is the most obvious approach. However, in case the single subject is not involved in the research project, it might also be possible to hand-select or hand-generate tasks that are then delivered to the subject.

Second, N-of-1 trials have the risk of depending on the point in time when the treatments are given to the subject. Thus, time dependency is a possible confounding factor that requires special handling. This risk probably plays a larger role in N-of-1 trials although it is a problem of crossover trials in general. We see the urgent need to be as critical as possible with this threat and we think it is necessary to analyze the data given to a subject upfront (i.e., before running the experiment) to make sure that random generation or ordering does not accidentally introduce or increase this threat. Furthermore, we see the need to still analyze whether a time-dependency existed in the measured dependent variable after the experiment execution. Possibly, this dependency has to be considered when analyzing the main experimental data.

Another facet of the here proposed trials is that their design tries to focus on the very special difference between the treatments. In our case, it was the difference in the syntax of the different treatments and the resulting experiment focused on this difference via corresponding tasks (count number of parameters). For obvious reasons, the measured differences cannot be generalized to ordinary code. But arguing that way against N-of-1 trials completely misses the point. The whole idea is to check whether some treatment has an effect or could have an effect. The goal is not to generate generalizable results. The goal is to search for the existence of an effect. Hence, we argue in the complete opposite direction as, for example, Sjøberg et al. who state that in order “to convince industry about the validity and applicability of the experimental results, the tasks, subjects and the environments of the experiments should be as realistic as practically possible.” (Sjøberg et al. 2002)

¹⁸See for example the blog entry by Felienne Hermans <https://www.felienne.com/archives/5931> or the registered records track at the MSR conference in 2020 <https://2020.msrconf.org/track/msr-2020-Registered-Reports>

We agree that such kind of large-scale experiments are desirable. But we see the need to handle such advice with caution because we see a high risk that experiments targeting generalizability end up in null experiments, i.e., experiments from which nothing can be concluded. And it is a known misconception that researchers argue that “a nonsignificant difference [...] means there is no difference between groups” (Goodman 2008, p. 136).¹⁹

N-of-1 trials have another advantage that researchers should factor in: an N-of-1 trial permits a single subject to experience the effect of the studied factors on his or her own. While in general the results of an experiment can only be interpreted by collecting data from a number of subjects, it is hard to retrace for a single subject that the measured effects are real. An N-of-1 trial inherently gets its results from a single subject. Therefore, by running the experiment on their own, a single subject can repeat the experiment and experience the effects measured in the experiment.

But N-of-1 trials have obvious risks, too. The most obvious one: What, if the single measured subject is one of the rare subjects who reacts different to the treatments than other subjects? In that case, the results are the opposite from an experiment with multiple subjects. Because of that we see the urgent need to repeat an N-of-1 experiment.

There is another risk researchers should be aware of. Since the experiment depends on a single subject it is probably easy for a single subject to compromise the data. For example, in the present experiment it would be possible for one subject to ignore the given tasks and just to press buttons as fast as possible. While we are quite sure that this did not happen in the present experiment (otherwise we would not have measured any effect at all), we think there is a need to define clear conditions that state whether or not a subject represents a valid subject for the experiment.

Again, we do not argue for the execution of N-of-1 trials because we think that the results are generalizable. We argue for such trials in order to ease the process of experimentation and to give an idea of the expected effect sizes for factors that are in the focus of an experiment. Further, we argue that such trials give the opportunity to estimate whether or not it is rewarding to execute an experiment on multiple subjects.

8.3 Possible Reasons for Measured Differences

The results of the experiment and its replications are quite clear. However, what’s not so clear are the reasons what there should be such a difference. Actually, we can only speculate about the reasons for these difference and can contribute some impressions we received from the data collection (where we typically informally spoke with the participants).

Just by looking into the code of a LE passed as a a listener or AIC, it become apparent that the AIC is much more verbose. Some participants in the experiment mentioned (after the experiment) that they had the feeling that is is harder to find the relevant information from an AIC. But they also said that the type information in the LEs were rather obstructive for the first task. It might be that there is a rather trivial reason for the differences between readability of AICs and LEs: the number of tokens that have to be identified. For an AIC one has to look at the return type or find the method of the AIC while this is not the case for a LE.

However, we did not get much comparable feedback for the second task. We personally had the feeling that the differences for the second task were already too small to be detected

¹⁹ According to Aczel et al. only 5% of the experiments insignificant results actually found strong evidence for the null hypothesis (Aczel et al. 2018).

by a subject. In fact, some participants said after the experiment that they do not expect to find any difference for the second task.

9 Conclusion

This paper focuses on two topics: a technical topic (readability of code depending on the usage of anonymous inner classes and lambda expressions) and a methodological topic (experimental design – the usage of N-of-1 trials).

First, this paper is about the readability of lambda expressions in comparison to anonymous inner classes in the programming language Java. The result is that for two tasks lambda expressions without type annotations were more readable than anonymous inner classes. However, for lambda expressions with type annotations we only measured sometimes differences to anonymous inner classes. For the given tasks – although they focused directly on the essential syntactical differences between both language constructs – the effect sizes were quite low. From that we conclude that the readability of lambda expressions without type annotations is better than the readability of anonymous inner classes but it probably hardly plays a role in reality (if used as the only factor to improve readability). Still, if language designers ask themselves whether their language should support both constructs, the answer is rather no – at least as long as there is no technical need to introduce anonymous inner classes such as, for example, the ability to override multiple methods. For Java developers it means that in scenarios where either lambda expressions or anonymous inner classes could be used (such as the application of the Stream API) they should rather use the first ones.

The second topic of this paper is the application of N-of-1 trials in the field of software science. We see a large opportunity in N-of-1 trials because the effort to design and execute such experiments is relatively low. Nevertheless they permit to analyze data using traditional statistical procedures. Consequently, the conclusion step, thus, the step from an observation and the interpretation of the results is easily traceable for other researchers. We also see risks in N-of-1 trials and the largest risk from our perspective is that people stop running RCTs with multiple subjects. At that point, it is up to the community to emphasize that the evidence from an N-of-1 trial is less than the evidence from an RCT with multiple subjects.

We are quite sure that a number of people will argue against N-of-1 trials in general and the present experiment in particular because of the missing generalizability. Still, we have to emphasize that in other empirical disciplines N-of-1 trials are a common research method. There, the application of N-of-1 trials is well-accepted: even restrictive research standards such as CONSORT, APA JARS or WWC accept N-of-1 trials as a valid study design.

It is important to note that we do not argue that N-of-1 trials should replace RCTs with multiple subjects. These should always be the goal of experimentation in software science. In other words: we do not argue that N-of-1 trials provide the same evidence as RCTs with multiple subjects. They do provide evidence. But the evidence is weaker than the evidence gathered from a multi-subject RCT.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the

article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

- Aczel B, Palfi B, Szollosi A, Kovacs M, Szaszi B, Szecsi P, Zrubka M, Gronau QF, van den Bergh D, Wagenmakers E-J (2018) Quantifying support for the null hypothesis in psychology: An empirical investigation. *Adv Methods Pract Psychol Sci* 1(3):357–366
- Appelbaum M, Cooper H, Kline R, Mayo-Wilson E, Nezu A, Rao S (2018) Journal article reporting standards for quantitative research in psychology: The apa publications and communications board task force report. *Am Psychol* 73:3–25
- Baltes S, Ralph P (2020) Sampling in software engineering research: A critical review and guidelines. *arXiv:2002.07764*
- Binkley D, Davis M, Lawrie D, Morrell C (2009) To camelcase or under_score. In: The 17th IEEE international conference on program comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17–19, 2009. pp 158–167
- Bossavit L (2015) The Leprechauns of software engineering. Laurent Bossavit
- Buse RPL, Sadowski C, Weimer W (2011) Benefits and barriers of user evaluation in software engineering research. In: Proceedings of the 2011 ACM international conference on object oriented programming systems languages and applications, OOPSLA '11. Association for Computing Machinery, New York, pp 643–656
- Microsoft C. (2007) C# Language Specification Version 3.0
- Dantas R, Carvalho A, Marcílio D, Fantin L, Silva U, Lucas W, Bonifácio R (2018) Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs. In: 2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER). pp 497–501
- Dony C, Malenfant J, Cointe P (1992) Prototype-based languages: From a new taxonomy to constructive proposals and their validation. In: Conference proceedings on object-oriented programming systems, languages, and applications, OOPSLA '92. Association for Computing Machinery, New York, pp 201–217
- Endrikat S, Hanenberg S, Robbes R, Stefik A (2014) How do api documentation and static typing affect api usability? In: Proceedings of the 36th international conference on software engineering, ICSE 2014. ACM, New York, pp 632–642
- Evans SR (2010) Clinical trial structures. *J Exper Stroke Transl Med* 3(1):8–18
- Fakhoury S, Roy D, Hassan SK, Arnaoudova V (2019) Improving source code readability: Theory and practice. In: Proceedings of the 27th international conference on program comprehension, ICPC '19. IEEE Press, pp 2–12
- Fischer L, Hanenberg S (2015) An empirical investigation of the effects of type systems and code completion on api usability using typescript and javascript in ms visual studio. *SIGPLAN Not* 51(2):154–167
- Forbes D (2013) Blinding: an essential component in decreasing risk of bias in experimental designs. *Evid Based Nurs* 16(3):70–71
- Franklin L, Gyorfi A, Lahoda J, Dig D (2013) Lambdaficator: From imperative to functional programming through automated refactoring. In: 2013 35th international conference on software engineering (ICSE), pages 1287–1290. IEEE
- Gamma E, Helm R, Johnson RE (1994) Design patterns. Elements of reusable object-oriented software, 1st edn., Addison-Wesley Longman Amsterdam, Reading. reprint edition
- Glass RL, Vessey I, Ramesh V (2002) Research in software engineering: an analysis of the literature. *Inf Softw Technol* 44(8):491–506
- Goodman S (2008) A dirty dozen: twelve p-value misconceptions. *Semin Hematol* 45(3):135–40
- Gosling J, Joy B, Steele G, Bracha G (2000) Java language specification, second edition: the java series, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston
- Gosling J, Joy B, Steele GL, Bracha G, Buckley A (2014) The Java language specification, Java SE 8 Edition, 1st edn., Addison-Wesley Professional, Boston
- Gosling J, Joy WN, Steele GL Jr (1996) The Java language specification, 1st edn. Addison-Wesley, Boston
- Gould JD, Grischowsky N (1986) Does visual angle of a line of characters affect reading speed? *Hum Factors* 28(2):165–173

- Hanenberg S (2010) Faith, hope, and love: An essay on software science's neglect of human factors. In: Proceedings of the ACM international conference on object oriented programming systems languages and applications, OOPSLA '10. ACM, New York, pp 933–946
- Hofmeister JC, Siegmund J, Holt DV (2019) Shorter identifier names take longer to comprehend. *Empir Softw Eng* 24(1):417–443
- Hollmann N, Hanenberg S (2017) An empirical study on the readability of regular expressions Textual versus graphical. In: IEEE working conference on software visualization, VISSOFT 2017, Shanghai, China, September 18–19, 2017. IEEE, pp 74–84
- Hoppe M, Hanenberg S (2013) Do developers benefit from generic types? an empirical comparison of generic and raw types in java. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '13. Association for Computing Machinery, New York, pp 457–474
- Institute of E. S. (2020) What works clearinghouse: standards handbook. U.S. Department of Education version 4.1 edition
- Jemiłniak D, Masukume G, Wilamowski M (2019) The most influential medical journals according to wikipedia: Quantitative analysis. *J Med Internet Res* 21(1):e11429–e11429
- Kaijhanaho A-J (2015) Evidence-based programming language design: a philosophical and methodological exploration. University of Jyväskylä, innland
- Kam-Hansen S, Jakubowski M, Kelley JM, Kirsch I, Hoaglin D, Kaptchuk TJ, Burstein R (2014) Altered placebo and drug labeling changes the outcome of episodic migraine attacks. *Sci Transl Med* 6:218ra5
- Kaptchuk TJ (1998) Powerful placebo: the dark side of the randomised controlled trial. *Lancet* 351(9117):1722–1725
- Katzmarski B, Koschke R (2012) Program complexity metrics and programmer opinions. In: 2012 20th IEEE international conference on program comprehension (ICPC). pp 17–26
- Kitchenham B, Fry J, Linkman S (2003) The case against cross-over designs in software engineering. In: Eleventh annual international workshop on software technology and engineering practice. pp 65–67
- Ko A, Latoza TD, Burnett MM (2015) A practical guide to controlled experiments of software engineering tools with human participants. *Empir Softw Eng* 20(1):110–141
- Kratochwill T, Hitchcock J, Horner R, RI Levin J, Odom SL, Rindskopf DM, Shadish W (2010) Single-case designs technical documentation. What works clearinghouse
- Lammers WJ, Badia P (2004) Fundamentals of behavioral research. Thomson/Wadsworth, Belmont
- Lawrie DJ, Morrell C, Feild H, Binkley DW (2006) What's in a name? A study of identifiers. In: 14th international conference on program comprehension (ICPC 2006), 14–16 June 2006, Athens, Greece. IEEE Computer Society, pp 3–12
- Lucas W, Bonifácio R, Canedo ED, Marcílio D, Lima F (2019) Does the introduction of lambda expressions improve the comprehension of java programs? In: Proceedings of the XXXIII Brazilian symposium on software engineering, SBES 2019. Association for Computing Machinery, New York, pp 187–196
- Madeyski L, Kitchenham B (2018) Effect sizes and their variance for AB/BA crossover design studies. *Empir Softw Eng* 23(4):1982–2017
- Mazinanian D, Ketkar A, Tsantalis N, Dig D (2017) Understanding the use of lambda expressions in java. *Proc ACM Program Lang* 1(OOPSLA):85:1–85:31
- McConnell S (2011) What does 10x mean? measuring variations in programmer productivity. In: Oram A, Wilson G (eds) Making software - what really works, and why we believe it, theory in practice. O'Reilly, pp 567–574
- Mirza RD, Punja S, Vohra S, Guyatt G (2017) The history and development of n-of-1 trials. *J R Soc Med* 110(8):330–340
- Nielebock S, Heumüller R, Ortmeier F (2019) Programmers do not favor lambda expressions for concurrent object-oriented code. *Empir Softw Eng* 24(1):103–138
- Okon S, Hanenberg S (2016) Can we enforce a benefit for dynamically typed languages in comparison to statically typed ones? a controlled experiment. In: 2016 IEEE 24th international conference on program comprehension (ICPC). pp 1–10
- ACM Transactions on C. E. (2021) Available at <https://dl.acm.org/journal/toce/author-guidelines>, last access: 07/31/2021
- Ortin F, Conde P, Fernandez-Lanvin D, Izquierdo R (2014) The runtime performance of invokedynamic: An evaluation with a java library. *IEEE Softw* 31(4):82–90
- Parnin C, Bird C, Murphy-Hill ER (2013) Adoption and use of java generics. *Empir Softw Eng* 18(6):1047–1089
- Patton MQ (2014) Qualitative research & evaluation methods: integrating theory and practice. SAGE Publications, Philadelphia

- Perdices M, Schultz R, Tate R, McDonald S, Togher L, Savage S, Winders K, Smith K (2006) The evidence base of neuropsychological rehabilitation in acquired brain impairment (abi): How good is the research? *Brain Impair* 7:119–132
- Petersen P, Hanenberg S, Robbes R (2014) An empirical comparison of static and dynamic type systems on api usage in the presence of an ide: Java vs. groovy with eclipse. In: *Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014*. Association for Computing Machinery, New York, pp 212–222
- Pierce BC (2002) *Types and programming languages*, 1st edn. MIT Press, Cambridge
- Rennie D (2004) Trial registration: a great idea switches from ignored to irresistible. *JAMA* 292:1359–62
- Rudell AP, Hu B (2010) Effects of long-time reading experience on reaction time and the recognition potential. *Int J Psychophysiol* 76(3):158–168
- Scalabrino S, Bavota G, Vendome C, Linares-Vásquez M, Poshyanyk D, Oliveto R (2017) Automatically assessing code understandability: How far are we? In: *2017 32nd IEEE/ACM international conference on automated software engineering (ASE)*. pp 417–427
- Senn SS (2002) *Cross-over trials in clinical research. Statistics in Practice*. Wiley, Hoboken
- Shadish W, Sullivan KJ (2011) Characteristics of single-case designs used to assess intervention effects in 2008. *Behav Res Methods* 43(4):971–980
- Sjøberg DIK, Anda B, Arisholm E, Dybå T, Jørgensen M, Karahasanovic A, Koren EF, Vokác M (2002) Conducting realistic experiments in software engineering. In: *Proceedings of the 2002 international symposium on empirical software engineering, ISESE '02*. IEEE Computer Society, p 17
- Stefik A, Hanenberg S (2017) Methodological irregularities in programming-language research. *IEEE Comput* 50(8):60–63
- Stefik A, Hanenberg S (2014) The programming language wars: Questions and responsibilities for the programming language community. In: *Proceedings of the 2014 ACM international symposium on new ideas, new paradigms, and reflections on programming & software, Onward! 2014*. ACM, New York, pp 283–299
- Stroustrup B (2013) *The C++ programming language*, 4th edn. Addison-Wesley Professional, Boston
- Stuchlik A (2011) Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In: *Proceedings of the 7th symposium on dynamic languages, DLS '11*. Association for Computing Machinery, New York, pp 97–106
- Tate RL, Perdices M, Rosenkoetter U, McDonald S, Togher L, Shadish W, Horner R, Kratochwill T, Barlow DH, Kazdin A, Sampson M, Shamseer L, Vohra S (2019) The single-case reporting guideline in behavioural interventions (scribe) 2016: Explanation and elaboration. *Prat Psychol* 25(2):119–151
- Tichy WF (1998) Should computer scientists experiment more. *IEEE Computer* 31:32–40
- Tichy WF, Lukowicz P, Prechelt L, Heinz EA (1995) Experimental evaluation in computer science: A quantitative study. *J Syst Softw* 28(1):9–18
- Tsantalis N, Mazinanian D, Rostami S (2017) Clone refactoring with lambda expressions. In: *Proceedings of the 39th international conference on software engineering, ICSE '17*. IEEE Press, pp 60–70
- Uesbeck PM, Stefik A, Hanenberg S, Pedersen J, Daleiden P (2016) An empirical study on the impact of c++ lambdas and programmer experience. In: *Proceedings of the 38th international conference on software engineering, ICSE 2016, Austin, TX, USA, May 14–22, 2016*. pp 760–771
- Van Rossum G (1991) *Documentation of the Python Programming Language* 0.9
- Vegas S, Apa C, Juristo N (2016) Crossover designs in software engineering experiments Benefits and perils. *IEEE Trans Softw Eng* 42(2):120–135
- Vieira R, McDonald S, Araújo-Soares V, Snihotta FF, Henderson R (2017) Dynamic modelling of n-of-1 data: powerful and flexible data analytics applied to individualised studies. *Health Psychol Rev* 11(3):222–234
- Vohra S, Shamseer L, Sampson M, Bukutu C, Schmid CH, Tate R, Nikles J, Zucker DR, Kravitz R, Guyatt G, Altman DG, Moher D (2015) Consort extension for reporting n-of-1 trials (cent) 2015 statement. *BMJ*: 350
- Wellek S, Blettner M (2012) On the proper use of the crossover design in clinical trials part 18 of a series on evaluation of scientific publications. *Deutsches Ärzteblatt Int* 109:276–81
- Zelkowitz MV (2009) An update to experimental models for validating computer technology. *J Syst Softw* 82(3):373–376
- Zelkowitz MV, Wallace DR (1998) Experimental models for validating technology. *Computer* 31(5):23–31

Affiliations

Stefan Hanenberg¹ · Nils Mehlhorn²

Nils Mehlhorn
mail@nils-mehlhorn.de

¹ Paluno – The Ruhr Institute for Software Technology, University of Duisburg–Essen, Essen, Germany

² 45128 Essen, Germany