Empir Software Eng
DOI 10.1007/s10664-013-9289-1

An empirical study on the impact of static typing on software maintainability

Stefan Hanenberg · Sebastian Kleinschmager · Romain Robbes · Éric Tanter · Andreas Stefik

Abstract Static type systems play an essential role in contemporary programming languages. Despite their importance, whether static type systems impact human software development capabilities remains open. One frequently mentioned argument in favor of static type systems is that they improve the maintainability of software systems—an often-used claim for which there is little empirical evidence. This paper describes an experiment that tests whether static type systems improve the maintainability of software systems, in terms of understanding undocumented code, fixing type errors, and fixing semantic errors. The results show rigorous empirical evidence that static types are indeed beneficial to these activities, except when fixing semantic errors. We further conduct an exploratory analysis of the data in order to understand possible reasons for the effect of type systems on the three kinds of tasks used in this experiment. From the exploratory analysis, we conclude that developers using a dynamic type system tend to look at different files more frequently when doing programming tasks—which is a potential reason for the observed differences in time.

Keywords Type systems · Programming languages · Empirical studies · Software engineering

Communicated by: Michael Godfrey and Arie van Deursen

Romain Robbes is partially funded by FONDECYT project 11110463, Chile, and by Program U-Apoya, University of Chile Éric Tanter is partially funded by FONDECYT project 1110051, Chile Andreas Stefik is partially funded by the National Science Foundation under grant no. (CNS-0940521). We thank them for their generous support of this work.

S. Hanenberg (✉) · S. Kleinschmager
Department for Computer Science and BIS, University of Duisburg-Essen, Essen, Germany
e-mail: Stefan.Hanenberg@icb.uni-due.de

R. Robbes · É. Tanter
PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Santiago de Chile, Chile

A. Stefik
Department of Computer Science, University of Nevada, Las Vegas, NV, USA

Published online: 11 December 2013

1 Introduction

The advantages and disadvantages of static and dynamic type systems in programming languages have been long debated. While many authors state that static type systems are important (see for example Bruce 2002; Pierce 2002; Cardelli 1997; Bird and Wadler 1988), others hold opposing views (see for example Tratt 2009; Nierstrasz et al. 2005). Typical arguments about the advantages of static type systems can be found in many text books on programming and programming languages:

- "Strong typing is important because adherence to the discipline can help in the design of clear and well-structured programs. What is more, a wide range of logical errors can be trapped by any computer which enforces it". (Bird and Wadler 1988, p. 8)
- "Types are also useful when reading programs. The type declarations in procedure headers and module interfaces constitute a form of documentation, giving useful hints about behavior." (Pierce 2002, p. 5)

Some of the drawbacks typically mentioned include (see for example Tratt 2009; Nierstrasz et al. 2005, pp. 149–159):

- A type system can be overly restrictive and forces the programmer to sometimes work around the type system.
- They can get in the way of simple changes or additions to the program which would be easily implemented in a dynamic type system but make it difficult in the static type system because of dependencies that always have to be type correct.

A more provocative statement about the possible drawbacks that can be found in the literature is:

- "Static type systems [...] are the enemy of change."(Nierstrasz et al. 2005, p. 4)

The debate regarding the advantages and drawbacks of static or dynamic type systems is ongoing in both academia and the software industry.[1] While statically typed programming languages such as C, C++, and Java dominated the software market for many years, dynamically typed programming languages such as Ruby or JavaScript are increasingly gaining ground—especially in web development.

The fact that the debate is still lively is not surprising, because settling it demands the presence of a theory of the respective advantages and disadvantages of static and dynamic typing, supported by empirical evidence. Unfortunately, such evidence is still lacking. This paper contributes to the emerging body of work on empirically validating type system research with controlled experiments (see Juristo and Moreno 2001; Wohlin et al. 2000; Sjøberg et al. 2005; Prechelt 2001 for introductions to controlled experiment) that investigates the possible benefits of static type systems.

In particular, this experiment investigates the two main claims made by proponents of static type systems: static type systems help in fixing programming errors, and type systems act as effective documentation. As such, the main research question for this experiment is whether a static type system is helpful to developers, given the following considerations: 1) a set of use cases involving new classes, and 2) in tasks involving fixing errors, both *type*

[1] The interested reader can find more arguments for both cases online, including the lively discussion available at: http://programmers.stackexchange.com/questions/122205/

errors—which manifest as no-such-method-errors in the dynamic setting—and *semantic errors*—which deal with correctness issues beyond the reach of simple type systems. The programming languages used in the experiment are Java and Groovy. Groovy was used as a dynamically-typed Java, i.e. additional language features in Groovy were not used in the experiment. The measurements are based on the time developers needed to solve a given programming task. The classes given to the subjects were either statically typed (Java) or dynamically typed (Groovy). Our observations show evidence for the following conclusions:

- **Static type systems are an effective form of documentation**: Subjects who used the statically typed version of the classes exhibited a significant positive benefit for tasks in which different classes had to be identified and used.
- **Static type systems reduce the effort to fix type errors**: Subjects who used the statically typed version of the classes showed a significant positive benefit for tasks in which type errors had to be fixed.
- **Static type systems may not be helpful in preventing semantic errors**: No significant differences between the statically and dynamically typed variants were found in regards to development time for tasks that involved fixing semantic errors.

This paper extends our previous paper published at ICPC 2012 (Kleinschmager et al. 2012). In comparison to the ICPC version, this paper adds: 1) an extended literature review of programming language studies and experiments that have been performed so far; 2) a more detailed description of the experiment itself; 3) a more complete description of the potential threats to validity; 4) an extended discussion of the results; 5) an exploratory study in which we look at additional data sources collected during the experiment; and 6) a comparison of the exploratory study with a similar one, conducted in a previous experiment (Mayer et al. 2012). Our goal with this extended version is to shed light on how the static or dynamic type system may impact the way a human interacts with the programming environment. The additional data sources we recorded, beyond development time, include:

- Test runs: we monitored how often the code of a task was compiled and run against unit tests.
- Number of opened files: we counted the number of files participants opened in order to solve a given task.
- Number of file switches: we measured the number of times a participant switched from file to file while solving a task.

In our previous experiment, the analysis of these aspects helped us develop a preliminary working theory to explain the observed effects. Based on the additional evidence presented in this experiment, we validate and refine this working theory.

Structure of the Paper Section 2 gives an overview of related work. Section 3 describes the experiment by discussing initial considerations, the programming tasks given to the subjects, the general experimental design, and threats to validity. Then, Section 4 describes the results of the experiment, presenting the measured data, descriptive statistics, and performing significance tests on the measurements. After discussing the results of the analysis in Section 5, we describe the exploratory study in Section 6, and compare it to our previous exploratory study. Finally, we conclude in Section 7.

2 Related Work

To date, the effect of static and dynamic type systems on software development has not been the subject of sustained study and the data that exists is at least partially contradictory. Some studies have found advantages for dynamic type systems, others for static type systems, while others still were either inconclusive or had mixed results. Clearly, additional evidence is needed.

Gannon's early experiment (Gannon 1977) revealed an increase in programming reliability for the subjects that used a statically-typed language. The experiment was a two-group experiment, in which each subject solved a task twice, with both kinds of type systems in varying order. The programming languages were artificially designed for the experiment. In contrast, the experiment we report on in this paper uses two existing programming languages, Java and Groovy. The experimental design is overall similar, but ours includes many more tasks.

Prechelt and Tichy studied the impact of static type checking on procedure arguments using two programming languages: ANSI C, which performs type checking on arguments of procedure calls; and K&R C, which does not (Prechelt and Tichy 1998). The experiment featured four groups and 34 subjects in total, each having to solve two programming tasks in each languages using each task—effectively a 2x2 design with four combinations. The experiment revealed a significant positive impact of the static type system with respect to development time for one task, but did not reveal a significant difference for the other. Interestingly, all programs came with both documentation and type annotations, so the difference was whether the type annotations were effectively *checked* or not. Our experiment draws inspiration from that of Prechelt and Tichy, as it uses two existing, similar languages, in order to control for variations between the languages, but is different in that type annotations are not present for the dynamically typed language, nor is documentation (in both settings).

A qualitative pilot study on type systems by Daly et al. (2009) observed four programmers who used a new type system (with type inference) for an existing language, Ruby—the resulting language is called DRuby. The programmers worked on two simple tasks. The errors messages shown by DRuby where manually classified according to whether they were "informative" or "not informative". Since only 13.4 % of the error messages were classified as "informative", the authors concluded that the benefit of the static type system could not be shown.

Denny et al. conducted an evaluation of syntax errors commonly encountered by novices with a result relevant to type systems research (Denny et al. 2012). Specifically, Denny et al. classified the amount of time spent fixing a variety of syntax errors in an introductory computer programming course in Java. Their results showed that type mismatch errors (e.g., trying to assign a double value to an integer) accounted for approximately 18.4 % of the time novices spend identifying and fixing compiler errors, second only to "cannot resolve identifier" errors, in the amount of time taken. Our study focuses on the time taken for more advanced programmers to write correct code, not introductory students' time spent fixing compiler errors.

An empirical evaluation of seven programming languages performed by Prechelt (2000) showed that programs written in dynamically typed scripting languages (Perl, Python, Rexx, or Tcl), took half or less time to write than equivalent programs written in C, C++, or Java. The study was performed on 80 programs in seven different languages. However, this study did not attempt to measure the performance of dynamic and static type systems and was not a controlled experiment. This introduces confounding factors, making comparisons between

the languages difficult. For example, Prechelt used data coming from both a controlled experiment and a public call for voluntary participation, in which participants self-reported timing information. Further, since programmers were free to use whatever tool support they wanted, there was considerable variability in the tools used to perform the tasks.

Two additional experiments were performed measuring the differences in productivity across several languages: Gat's study (Gat 2000) compared C, C++ and Lisp; earlier, Hudak and Jones (Hudak and Jones 1994) compared Lisp, Haskell and Ada. However, similarly to Prechelt's experiment, both of these experiments share the same issue: it is difficult to assign the observed difference between the languages to a single characteristic, as the set of languages varied substantially both in their design and how they were used in the experiments. In contrast to these three studies, we carefully selected and controlled the usage of two languages, Java and Groovy, so that the only difference is effectively the type system. We did this to focus our observations on the differences that could be attributed between static and dynamic type systems.

The study by Ko et al. analyzed how developers understand unfamiliar code (Ko et al. 2006). One of the outcomes of the experiment is that "developers spent, on average, 35 percent of their time with the mechanics of redundant but necessary navigations between relevant code fragments" (Ko et al. 2006, p. 972). The relationship to our work is that the study by Ko et al. gives a an indicator of where developers potentially waste development time. The exploratory study which complements our experiment concurs with Ko et al., and indicates that the choice of the type system has a direct impact on the resulting navigations between code fragments.

The notion that type systems act as a form of documentation is well supported in practice. Van Deursen and Moonen argue that "[Explicit types] can help to determine interfaces, function signatures, permitted values for certain variables, etc" (van Deursen and Moonen 2006). They propose a type inference system for Cobol systems in order to enable hypertext navigation and program comprehension of the system; validating it with a Cobol system of 100,000 lines of code.

The study presented here is part of a larger experiment series about static and dynamic type systems (Hanenberg 2011). In Hanenberg (2010) we studied the effect of static and dynamic type systems to implement a scanner and a parser. Forty-nine subjects were recruited to perform these two large tasks; each subject took an average of 27 hours to work on both tasks. Results showed that the dynamic type system had a significant positive time benefit for the scanner, while no significant difference with respect to the number of fulfilled test cases for the parser could be measured. However, the reasons for this result was unclear. In contrast, this experiment features more tasks, which are also shorter, allowing us to explore the impact of complexity on performance. This experiment also includes an exploratory study of additional data recorded in order to shed more light on the observed results.

In Stuchlik and Hanenberg (2011) we analyzed to what extent type casts, which occur in statically typed programs, influence simple programming tasks. In contrast, a dynamic type system makes type casts unnecessary. We divided twenty-seven subjects in two groups according to a within-subject design: the subjects were exposed to both the dynamic type system of Groovy, and the static type system of Java, varying the order of the tasks. We found out that type casts did negatively influence the development time of trivial programming tasks for users of the static type system, while longer tasks showed no significant difference. If this experiment investigated a potential downside of static type systems (type casts are used precisely at those places where the static type system is not expressive enough

to accept the program), the present one investigates various potential benefits of static type systems.

Finally, we previously performed two additional experiments that investigated similar aspects to those addressed in this experiment:

- The first experiment (Steinberg and Hanenberg 2012) revealed that fixing type errors is significantly faster with a static type system (in comparison to no-such-method errors occurring at runtime with a dynamic type system). The study was performed with 31 subjects, using a similar two-group, within subject designs, where the two treatments where Java and Groovy. There were 8 different tasks in the experiment, with variations of difficulty, and additionally 3 of the tasks had semantic errors instead of type errors. Java was found to be faster than Groovy for the discovery of type errors, but not for semantic errors.
- In the second experiment (Mayer et al. 2012), we analyzed the impact of static or dynamic type systems on the use of undocumented APIs. The study showed a positive impact of the static type system for three out of five programming tasks, and a positive impact of the dynamic type system for two other tasks. The additional exploratory study we performed seemed to indicate that the tasks with the larger number of types to identify, or the tasks featuring complex type definition (e.g., generics with nested types), were the tasks for which the static type system is the most beneficial. For simpler tasks, the static type systems seemed to incur a higher cost compared to the dynamic type system. The exploratory study suggested that the program comprehension strategy employed by users of static type systems was more systematic, hence slower for simple tasks, but paying off for larger tasks.

This experiment can be seen as a partial replication (Juzgado and Vegas 2011) of aspects of these two experiments. A partial replication allows differences in the design of the experiment in order to test variations of the original hypotheses. For the first experiment, we wish to see whether the advantage in fixing type errors we found still holds in a different experiment, and whether the lack of difference with respect to semantic errors holds as well. For the second experiment, we also want to investigate if the findings on undocumented code still hold, and additionally we want to investigate if the working theory issued from the first exploratory study is also confirmed in this experiment, or whether other factors are at play.

3 Experiment Description

We start with initial considerations, then discuss our choices of programming languages, programming environment, and measurement methodology. After introducing the experimental design, discussing alternatives and the learning effect, we give a detailed description of the programming tasks. Then, we describe the experiment execution and finally, we discuss the threats to validity.

3.1 Initial Considerations for Experimental Design

The intent of the experiment is to identify in what situations static type systems impact development time. The underlying motivation for this experiment is that previous experiments already identified a difference between static and dynamic type systems for programming tasks (Gannon 1977; Prechelt and Tichy 1998; Hanenberg 2010; Stuchlik and Hanenberg 2011). According to previous experiments and the literature on type systems,

our expectations were that static type systems potentially help in situations like 1) adding or adapting code on an existing system, and 2) finding and correcting errors. Therefore we examine three kinds of programming tasks:

1. Using the code from an existing system where documentation is only provided by the source code;
2. Fixing type errors (no-such-method errors for dynamically typed applications) in existing code;
3. Fixing semantic errors in existing code (e.g., not respecting the protocol when interacting with an object).

We phrase our hypotheses formally as being neutral, indicating that static type systems have no impact. The hypotheses followed by the experiment were:

1. Static type systems have no influence on development time if the classes that should be used are only documented by their source code;
2. Static type systems have no influence on development time if type errors need to be fixed;
3. Static type systems have no influence on the debugging time necessary to fix semantic errors.

The programming tasks reflect the three hypotheses. We did not want to have a single task for each hypothesis for two reasons: (1) we wanted to investigate variations in complexity in the same type of tasks, and (2) the experiment's result could be heavily influenced by a confounding factor specific to a particular task, such as its description. Thus we defined several tasks for each hypothesis.

3.2 Choice of Languages

Since we need statically and dynamically typed programming tasks, the choice of programming languages is crucial. To further reduce confounding factors, our goal was to use languages as similar as possible, and that do not require exhaustive additional training for the subjects. All the subjects were already proficient with Java, making it a strong candidate for the statically typed language. Groovy is a dynamic language for the Java Virtual Machine that is tailored for Java programmers and tries to be as compatible as possible with Java. Although Groovy's main interest is in the many advanced languages features that it offers over Java, it can be used simply as a dynamically-typed Java, essentially writing Java code where all type annotations are removed.[2] As such, this language pair was an obvious choice.

3.3 Environment and Measurement

We use the Emperior programming environment, which was used in previous experiments (Endrikat and Hanenberg 2011; Mayer et al. 2012). It consists of a simple text editor (with syntax highlighting) with a tree view that shows all necessary source files for the experiment. From within the text editor, subjects are permitted to edit and save changes in the code and to run both the application and test cases.

[2]For more information about Groovy and the differences it has with Java, the interested reader can consult the following web page: http://groovy.codehaus.org/Differences+from+Java

Subjects worked sequentially on each individual task, without knowing the next ones. Every time a new programming task was given to the subject, a new IDE was opened which contained all necessary files (and only those files). For each task, subjects were provided executable test cases, without their source code. We measured the development time until all test cases for the current programming task passed; we do not need to measure correctness, as we consider that passing all tests implies correctness. The programming environment (IDE with programming tasks, test cases, etc.) including the operating system (Ubuntu 11.04) was stored on an USB stick that was used to boot the machines used in the experiment.

3.4 Experimental Design

The experimental design followed in this paper is a repeated measures, within-subject, design that has been applied in previous experiments (Gannon 1977; Stuchlik and Hanenberg 2011; Endrikat and Hanenberg 2011; Mayer et al. 2012). The motivation for using such a design is that a relatively low number of participants are required to measure an effect. Indeed, while within-subject designs potentially suffer from the problem of learning effects, an important issue that should not be brushed away lightly, they have the benefit that individual differences in performance can be considered in the analysis by use of a statistical procedure known as a repeated measures ANOVA. Given that it is known that individuals exhibit a high variability in programming tasks (Curtis 1988) (with an order of magnitude differences reported McConnell 2010), such a procedure mitigates potential threats to statistical power.

The use of repeated measures within-subject designs can potentially take many forms and a discussion of the exact procedure is important for the purposes of scientific replicability and for identifying potential threats to validity. As such, we describe our exact procedures here in more detail. First, because our design is within-subjects, developers are given two treatments, one with static typing and another with dynamic. Second, for each treatment, participants complete a set of tasks (e.g., a set of classes with type annotations), and then complete the same tasks with the second treatment (e.g., a different set of classes, of similar size and complexity, but without type annotations). In order to study whether there is a difference between both solutions, we divide the participants into two groups, letting one start the development tasks with Java (that is, with type annotations and static type checking) and the other start with Groovy (that is, without type annotations and without static type checking). Having groups start in such a way is often termed counterbalancing, thus giving us a repeated measures, counterbalanced, within-subjects design, which is illustrated in Table 1. This design helps to mitigate the issue of variability between subjects, but leaves the potential threat of learning effects between treatments, which we discuss in the Section 3.6.

Table 1 General experimental design

	Technique for all tasks	
	Round 1	Round 2
Group A	Groovy	Java
Group B	Java	Groovy

3.5 Discussion of Alternatives in Experimental Design

While the design described above has been applied previously and has the benefit of finding effects with small numbers of participants, all experimental designs have threats to validity and we considered several alternatives. The chosen experimental design suffers from two main problems: a potential between-treatments learning effect when subjects switch treatments—i.e, from Groovy to Java or vice versa—and repeat the tasks (a within-task learning effect); and a potential learning effect when subjects switch from one task to another within the same type system treatment (a between-task learning effect). The between-treatments learning effect is especially problematic because, when the subjects are working on the same task for the second time, it is unclear how conducting the tasks a second time impacts the developer's scores. We discuss this effect in more detail in Section 3.6.

There are experimental designs that plausibly handle such situations, but they have significant drawbacks. For example, one possibility would be to conduct a full-factorial design, with different groups for each type system treatment and each programming task (Wohlin et al. 2000, Chapter 8.5.3). While the benefit of this approach is that it alleviates potential learning effects, conducting such an experiment is highly impractical—the high number of combinations of experimental configurations would require a considerable number of subjects; far more than would be feasible for most research groups. While a full factorial design is impractical, two potential alternatives are more realistic, namely the AA/AB design (also known as pre-test/post-test control group design Gravetter and Wallnau 2009) and the latin-square design (Gravetter and Wallnau 2009; Pfleeger 1995).

In the AA/AB design, the subjects are divided into two groups. One group (the AA group) solves the programming tasks under one treatment only, but twice (for instance, using the dynamically typed language). The other groups starts with the same treatment (e.g., the dynamically typed language) and then crosses over to the other treatment (the statically typed language). The intention of the design is to measure an interaction between both rounds. While this design is simple and provides a clear way to measure learning, it suffers from a different problem: when both groups are unbalanced, it is possible that the influence of individual subjects on the result of the group is too strong. Given that not every subject is measured under both treatments, this design also does not completely capture potential learning effects. Further, since only half of the participants perform both kinds of tasks, statistical power is lowered substantially and we can test fewer alternative tasks, limiting our investigative ability—we are effectively "sacrificing" half of the subjects to measure the interaction. Given the low number of subjects in our study, and since programming tasks exhibit high variance, we did not choose this design in the experiment.

Another possible alternative is the latin-square design. In this design, different treatment combinations of the variables *programming task* and *type systems* are assigned to different subjects. Such designs are thus similar to full factorial designs, except that only specific orderings are chosen to be run. For example, the first subject could receive a first programming task with the static type system, a second task with the dynamic type system, etc. The second subject might start with a second programming task using a dynamic type system. The benefit of the latin-square design is that it provides most of the benefits of a full factorial design (e.g., accounting for many possible experimental permutations, and helping to measure learning effects), due to the varying task and type system assignments. However, while latin-square designs do reduce the number of needed subjects compared to a full-factorial design, the tradeoff is, again, that the number of participants required is still larger than it would be for a repeated measures-style experiment, due to the between-subject variation.

One additional issue of using the latin-square design in our case is that it requires that the number of subjects to be a multiple of the number of rows (each row ideally has the same number of subjects). Our intention was to get as many subjects as possible in order to increase the statistical power of the experiment, but reaching a perfect multiple of the number of rows is usually difficult, forcing the design team to either choose to run the statistics anyway, despite the design flaw, or to drop subjects. This is problematic in our case, since we have 9 tasks in the experiment, which would translate to 9 rows. The risk of discarding as many as 5 to 8 subjects was deemed too large. Given that the number of participants in each row would also be very small, we further ran the risk of one subject influencing the score to much for one particular row. Further, while Latin-square designs are often considered a reasonable compromise from a full factorial design, there is no guarantee that such a design really will capture all learning effect considerations for the exact set of tasks and treatments we have used.

As one final issue with latin-square designs in our specific case, one issue is related to assigning subjects to a specific combination: we cannot determine upfront how many participants will finish the experiment (some may skip the experiment, others may not complete all the tasks), much less pre-assign them to a given combination without processing each participant sequentially. Processing participants sequentially would have significantly lengthened the time needed to do the experiment and added additional logistical issues due to the availability of the subjects.

Hence, we acknowledge readily that all experimental designs carry with them pros and cons, each with competing risks. While we would encourage other research groups to re-test our hypotheses under a variety of conditions, in our specific case, we considered our choice for the experimental design to be imperfect, but less risky than common alternatives.

3.6 Data Analysis in the Presence of Learning Effects

Learning effects are a potential problem in any experimental design in which the same participant completes more than one task. In our setting, the most problematic learning effect is the within-task learning effect that happens when subjects cross over to the second treatment: once a programming task is solved using language A, it becomes potentially easier to solve a similar programming task with language B. However, there are several standard statistical approaches for analyzing data in such studies. Indeed, this experimental design is common enough in Psychology to be included in any undergraduate (Gravetter and Wallnau 2009) or graduate level textbook (Rosenthal and Rosnow 2008). Our analysis treats both groups of subjects separately and combines the results in a subsequent step using a statistical procedure known as a *repeated measures ANOVA*,. This statistical test is specifically designed for such circumstances (Rosenthal and Rosnow 2008). We have successfully applied this approach in the past (Stuchlik and Hanenberg 2011; Mayer et al. 2012).

In the present experiment, let G be the group of subjects starting with Groovy, and let J be the group starting with Java. For demonstration purposes, we make the assumption that there exists an advantage to using Java. We then apply the following data analysis strategy:

- Apply a statistical test to determine if there is a statistical difference between each language in group G, and the same test in group J. There are four cases:
- Case A: If both tests find a statistical difference in favor of Java, we conclude that using Java has a positive effect.

– Case B: If one test finds a statistical difference (presumably group G), but the other does not, we still conclude that Java has a positive effect, albeit smaller than the previous case.
– Case C: If both tests find a statistical difference but in opposite directions (presumably G in favor of Java, and J in favor of Groovy), the results are inconclusive. The language effect, if present, is smaller than the learning effect.
– Case D: Both tests find no statistical difference; we can conclude that the language effect is too weak to matter, regardless of the learning effect.

We explain the impact of the learning effect in Fig. 1. The figure is a set of scenarios covering the relationship between language and learning effect. Each scenario is a column in the figure, where we see a pair of subjects: one in group G, and one in group J. Each subject performs a single task twice; hence there are two measurement bars, one for each language. Black bars represent times for Java, gray bar times for Groovy. The striped bar represents the learning effect: if it were absent, the measurement for the second language would reach the end of the bar. The difference between the measurements is shown with the dotted line. In group G, subjects start with Groovy, hence Java benefits from the learning effect (Fig. 1, top row). In group J, subjects start with Java, hence Groovy benefits from the learning effect (Fig. 1, bottom row). The three scenarios are:

– In the first scenario, the language effect is larger than the learning effect: subjects in both groups show significant benefits when using Java. The small learning effect makes the difference in measurements slightly more important for group G than for group J.
– The second scenario shows what happens when the learning effect is roughly equal to the language effect: in that case one of the groups will show a difference, while the second group will not, as both effects cancel out. We can see that the design will show a conclusive result even if the language effect and the learning effect are roughly equal as this would fall under case B. Of course, the evidence is less strong than in case A.
– The experimental design would fail if the learning effect is much larger than the effect of the type system, as is shown in last scenario of Fig. 1. Both groups yield

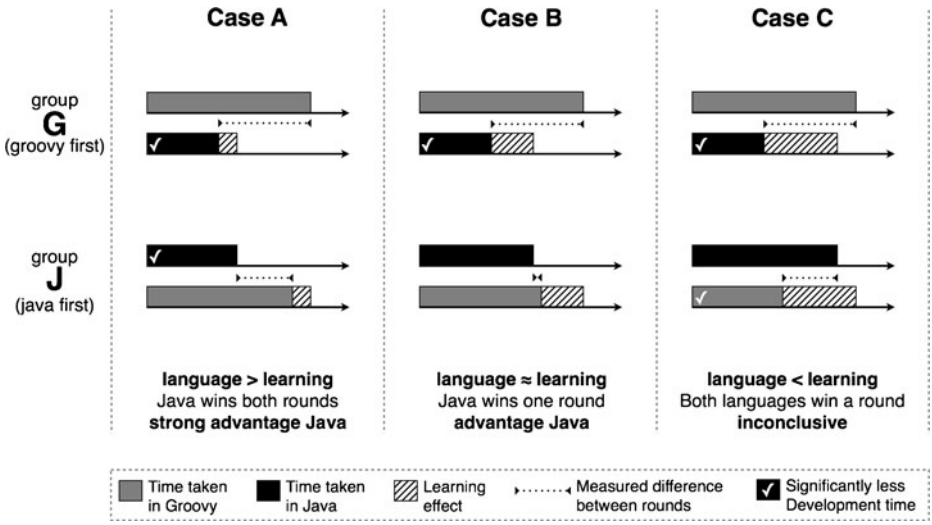

Fig. 1 Impact of the learning effect on the conclusions drawn by the experiment

opposite conclusions as the learning effect is what is effectively measured. If this was the case in our experiment, we would arrive to opposite conclusions in both groups (case C) in most of the tasks. The experiment would only show that subjects who perform a programming tasks for the second time are quicker in the second round—which is irrelevant for the research questions that this experiment aims to provide insight about.

Our goal is hence to reduce the impact of the learning effect as much as possible. We show in Section 3.7 how we achieve this, by giving subjects tasks that are similar applications in terms of size, complexity and structure, but with widely different application domains. We take the position that if the remaining learning effect is large enough to counterbalance the language effect, then the effect was not of enough practical significance to start with.

3.7 Base Application

The subjects worked on two base applications, one in Java, and another in Groovy. To limit confounding factors, we want to make the base applications be as similar as possible. To limit within-task learning effects, we want to make the applications *appear* as different as possible. We do this by transforming one of the application into a structurally equivalent one.

The software used by the participants was based on a small turn-based video game written in Java for a previous experiment (Hanenberg et al. 2009), which was slightly extended. This application consisted of 30 classes, with 152 methods, and contained approximately 1300 lines of code. To make the application suitable for this experiment, we applied three transformations to its source code:

1. **Translation to another language.** We translated the application to Groovy, removing all static type system information in the process.
2. **Translation to another domain**. To control for the complexity of the tasks and reduce learning effects at the same time, we translated the application to another program domain. While it is difficult to do so completely, we renamed all the classes, fields, methods, and other code constructs in the Java application, so that they appear to be from a different application domain, while having an identical structure. The end result is an application structurally similar to the game, yet appearing to be a simple e-mail client. The effectiveness of the translation can be seen by the visible differences in the solutions presented in Tables 10 and 11. We admit readily that this is an imperfect solution, as humans may pick up on the renaming. As such, while we think renaming is a plausible approach, it remains a potential threat to validity.
3. **Removing latent type information.** Since we want to study the effect of type systems as implicit documentation, we removed implicit type annotations. For both programs, we renamed field and method parameter names so that they would not reflect the exact type they contained. This was done to remove the documentation value of type names from variables, and to make the program type free. Variables were renamed using synonyms of the types they stood for.

3.8 Programming Tasks

The experiment consisted of 9 tasks, each of which had to be solved in both languages. In addition to these regular tasks, a warm-up task (not part of the analysis) was provided to

make the participants comfortable with the environment they had to use. The task descriptions were provided as a class comment. Example solutions for each task can be found in the Appendix (Tables 10 and 11). According to the hypotheses in the experiment we designed three kinds of tasks:

- **Class identification tasks (CIT)**, where a number of classes needs to be identified (Table 11). The participants had to fill a method stub in the task.
- **Type error fixing tasks (TEFT)**, where a type error needs to be fixed in existing code (Table 10).
- **Semantic error fixing tasks (SEFT)**, where a semantic error needs to be fixed in existing code (see Table 10).

In the following we describe the characteristics of each task. The numbering of the tasks corresponds to the order in which the tasks were given to the subjects. We explain the task description for only one of the domains to conserve space.

3.8.1 CIT1 (Two Classes to Identify)

For this task two classes have to be identified, namely a *Pipeline* class which took a generic type parameter and the *ActionsAndLoggerPipe* class. Instances of these have to be used by initializing a type *ActionsAndLoggerPipe* with two *Pipeline* instances and then passing the pipe along a method call.

3.8.2 CIT2 (Four Classes to Identify)

This task requires 4 classes to be identified. In Java, instances of *MailStartTag* and *MailEndTag* have to be sent to an instance of the type *EMailDocument*, along with an *Encoding* (which is an abstract class, but any of the provided subclasses was a correct choice for instantiation). Additionally, both start and end tag have to be provided with a *CursorBlockPosition* instance during their creation.

3.8.3 CIT3 (Six Classes to Identify)

For this task six classes need to be identified. A *MailElement* subclass of type *OptionalHeaderTag* has to be instantiated, outfitted with several dependencies and then returned.

3.8.4 SEFT1

In this task, a semantic error needs to be fixed. Subjects are given a sequence of statements that are executed during test runs and are given a starting point for debugging. An additional consistency check shows what was expected from the program. In the code, when a cursor reaches the last element of an e-mail, it should result in a job of type *ChangeMailJob* in order to load the next mail. Because of the error, a *SetCursorJob* instance is wrongly used instead, which reset the cursor back to the first element of the current mail. Note that this does not result in a type error because the interfaces of the two job classes are compatible. The consistency check provides a description of the error and tells the participants that the current document has not changed after reaching the last element.

3.8.5 SEFT 2

Subjects are given a code sample that interacts with the application and which contains a consistency check. In this task, the goal is to identify a missing method call. Table 10 shows both the wrong code and the correct solution for the Groovy video game. The problem is that once a player moves from one field to the next, a move command should execute and set the player reference to the new field and delete the reference from the previous field. The missing call leads to duplicate player references, which are detected by the consistency check. The participants had to insert the missing call.

3.8.6 CIT4 (Eight Classes to Identify)

This task requires instantiating the *WindowsMousePointer* class. This object has to be outfitted with dependencies on other objects (e.g., icons, cursors, theme). The task requires the subjects to identify the class hierarchies and subclasses that are needed. Enumeration types were used as well, although the specific values were not critical to the task.

3.8.7 TEFT1

This task contains a type error where the faulty code location is different from the program run-time exception. Because of the nature of this error, they are easily detected by the static type checker in Java. As such, only the Groovy tasks require explanation (see Table 10). In the erroneous code in Groovy, a *GameObject* instance is inserted into a property that is supposed to be a simple *String*. As such, when the consistency check runs, the properties are concatenated, which leads to a run-time exception, because the *GameObject* does not have a concatenate method. The solution is to remove the *GameObject* and keep the *String*.

3.8.8 CIT5 (Twelve Classes to Identify)

This task requires subjects to identify twelve classes; the largest construction effort of all type identification tasks. In Java, participants have to configure a *MailAccount* instance with dependencies to other objects, which represent a part of the mail account configuration (e.g., user credentials). These objects also have dependencies to other objects, resulting in a large object graph.

3.8.9 TEFT2

We suspect this task is one of the more difficult for Groovy developers. It contains a wrongly assigned object leading to a run-time error, but the distance between the bug insertion and the run-time error occurrence is larger than for tasks TEFT1. When a new *TeleportCommand* is created, it is outfitted with dependencies to the player and the level. The bug is that the order of the two parameters is wrong. The solution is to switch the order of the two types, which may not be obvious.

3.8.10 Summary of Programming Tasks

Table 2 gives an overview of the characteristics of each programming task. Five programming tasks required participants to identify classes; these tasks varied with respect to the

Table 2 Summary of programming tasks for subjects in Group A (Groovy starters), G=Groovy, J=Java

Task number	1	2	3	4	5	6	7	8	9
Task name	CIT1	CIT2	CIT3	SEFT1	SEFT2	CIT4	TEFT1	CIT5	TEFT2
#Identified classes	2	4	6			8		12	
Language	G	G	G	G	G	G	G	G	G
Task number	10	11	12	13	14	15	16	17	18
Task name	CIT1	CIT2	CIT3	SEFT1	SEFT2	CIT4	TEFT1	CIT5	TEFT2
#Identified classes	2	4	6			8		12	
Language	J	J	J	J	J	J	J	J	J

number of classes to be identified. For both the type and semantic errors, we designed two programming tasks.

For CIT Tasks, developers are given an empty method stub where parameters either need to be initialized or used for the construction of a new object (which requires additional objects as input). For TEFT Tasks, Java developers have code that does not compile due to a type error. In contrast, for Groovy developers, a test case fails. In both cases, the subjects have to change the code base. For SEFT Tasks, all subjects are given failing tests and the code base must be modified until all pass.

3.9 Experiment Execution

The experiment was performed with 36 subjects, of which 33 finished the tasks. Of these subjects, thirty were students, three were research associates, and three were industry practitioners. All subjects were volunteers and were randomly assigned to the two groups. Two practitioners started with Java and all three research associates started with Groovy. A more detailed description of the subjects can be found in Kleinschmager (2011). The experiment was performed at the University of Duisburg-Essen within a time period of one month. The machines used by the subjects where IBM Thinkpads R60, with 1GB of RAM.

3.10 Threats to Validity

As with any scientific study, this study has a number of potential threats to validity—and according to the literature, it is desirable to describe them (Kitchenham et al. 2006; Wohlin et al. 2000). Here, we discuss those threats that are from our point of view most problematic—and discuss to what extent these threats can bias the measurements and the result of the experiment. Some of the threats are general for these kinds of experiments (students as subjects, small programming tasks, artificial development environment), which are already discussed in detail in other related experiments (see for instance Stuchlik and Hanenberg 2011; Hanenberg 2010); we still discuss them here for completeness.

Multiple Learning Effects (Internal and External Validity) The experiment design is a repeated measures design—a single subject is working on multiple programming tasks and each task is repeated with both treatments. Due to this, there are probably multiple learning

effects. The potential learning effects are **within-tasks learning**, **between-tasks learning** and **familiarness effect**:

- First, there is obviously a **within-task learning effect**: once subjects have solved a programming task, it is likely that they will solve the programming task the second time faster. We tried to reduce this learning effect by changing the programming system between the two tasks, i.e. the concrete class names that developers are familiar with when they solve a programming task for the first time do not help, when they solve the same task for the second time. We do not have quantitative data about whether the subjects realized the similarity between the two systems, but during discussions with subjects after the experiment we did not encounter subjects who were conscious of it.[3] There may still be a learning effect, but it would have been much greater if the subjects realized that the systems were similar.
- The **between-tasks learning effect** occurs when subjects solve a certain kind of programming task repeatedly; they might be faster on subsequent occasions. In that case, the concrete knowledge from one task does not help when solving a similar task for the second time but the strategies that subjects might have developed for solving a certain task might help them for a similar task. We tried to reduce this problem by interleaving tasks of different kinds—for example, while the first three programming tasks are class identification tasks, the following tasks are of a different kind before coming back to class identification tasks (see Table 2).
- The **familiarness effect** is also a kind of learning effect, where subjects are getting used to the programming environment and the way how programming tasks are delivered by the experimenter. While we consider this learning effect as potentially problematic, we did not see a way to reduce it besides the introduction of a warmup task at the start of the experiment to absorb most of it.

In general, all of these effects could be removed by doing a completely different kind of design, where one subject only solves one concrete programming task—without doing any further tasks. This would imply, that a large number of different groups would be necessary and consequently, a large number of subjects. Although this kind of design is in principle possible, we did not consider this for practical reasons, since we were not able to recruit a sufficiently large number of subjects (see Section 3.5).

Chosen Sample—Randomization (Internal Validity) The experiment made use of randomization: subjects were randomly assigned into the Java first or the Groovy first groups. Randomization has, by definition, the potential problem that the groups are not balanced. The larger the sample size, the less likely the problem. The experiment was explicitly designed for (and performed on) a rather small sample size. Consequently, it is possible that both groups are not balanced with respect to the subjects' capabilities. However, the experiment uses a within-subject comparison which significantly reduces this problem, because the effect of static type systems is mainly detected by the effect it has on each individual itself. Consequently, even if both groups are not balanced, the experiment can detect the differences.

Chosen Sample—Experience (Internal and External Validity) One of the major problems in empirical studies is that it is, in general, unclear to what extent the software development

[3] We plan to gather data on this aspect in future experiments.

experience of subjects influences the results. While studies such as the one described in Feigenspan et al. (2012) try to determine the development experience by using questionnaires, the current state of the art in empirical software engineering currently lacks a method for meaningfully measuring developer experience. As a consequence, it is unclear to what extent the chosen sample is similar to the population of all software developers—even if professional developers are experimental subjects. Consequently, it is possible that the chosen sample, which consisted mostly of students, is an internal threat (because students have been considered in the experiment) as well as an external threat (because the results might differ from those that an experiment that had additional professional developers may have yielded). However, it should be noted that several studies show that in some circumstances the use of students as subjects does not change the results of the studies (see for example Höst et al. 2000 and Tichy 2000).

Chosen Tasks and IDE (External Validity) We explicitly designed the programming tasks so that they do not use complicated control structures such as loops, recursion, etc. The rationale for this choice is that using those control structures probably increases the variability amongst subjects. Given this potential problem, the effects observed here may not generalize to real-world programs in industry. The same is true for the chosen IDE. The IDE can be considered as a text editor (which provides syntax highlighting) with the ability to have a tree view of the files in the current software projects. However, typical features of an industrial IDE, like code refactoring, quick fix, intuitive navigation, were not available. As a consequence, the development times measured in the experiment cannot be considered as representative development times in an industrial scenario. However, such tools complicate the language comparison as the support accross languages varies—they introduce confounding factors. For instance, the Eclipse IDE's support for Java is much more mature than its support for Groovy. Consequently, using these tools in the experiment would have led to an unfair comparison of the languages and by extension their type systems—with the risk that the measurements would have been mainly an indicator for the maturity of the IDE instead of being usable for a comparison of type systems.

Type Annotations (Internal and External Validity) For practical reasons we decided to use Java and Groovy as representatives for languages with static and dynamic type systems. The type system of Java always requires explicit type annotations, as opposed to type systems with inference such as ML, Haskell, or to a lesser extent, Scala. As a consequence, Java source code requires more text. The downside is that writing code requires more keyboard input, while the upside is that the type annotations represent a form of explicit documentation. If our experiment reveals a difference between Java and Groovy, it is possible that the observed effect is not related to the type system, but is instead related to the syntactic element type annotation. In other words, the effect of either syntax or our choice of languages (e.g., Java and Groovy) on the experimental results is not clear from this study.

Expressiveness of the Type System (Internal and External Validity) The experiment considers Java as representative of a statically-typed language. However, the design space of type systems is very wide, and a variety of more expressive type systems exist, such as that of Haskell or Scala. Hence, it is possible that the results we obtain apply only for type systems similar to that of Java (i.e., nominal object types with subtyping and generics). Another possibility is that the experiment may not reveal any significant result due to the fact that the impact of the type system of Java is less than that of a more expressive type system.

Removing Latent Type Information (Internal Validity) The dynamically typed code was artificially constructed in a way that the names of the parameters, variables, etc. did not contain any direct hints concerning types. We may have introduced additional complexity to the dynamically typed solutions in the process. In fact, to what extent (or for which percentage or in what situation) parameter names in dynamically typed code reflect the classes/types that are expected is unclear. An empirical investigation of dynamically typed source code repositories could help answer this question (examples of such studies are Callaú et al. 2013; Richards et al. 2011). Despite our best efforts, it is possible that some indirect type information remains, as humans can recognize synonyms easily.

4 Experiment Results

We start with presenting descriptive statistics, before presenting the results of the analysis of variance, and continue with our task-wise and group-wise analyses.

4.1 Measurements and Descriptive Statistics

Table 12 shows the observed development time for all tasks, and Table 3 shows the corresponding descriptive statistics. The boxplot in Fig. 2 gives a more intuitive representation of the data. Both the data and the descriptive statistics reveal several facts. First, for no subject the total development time for Java was greater than the total development time for Groovy. Second, for all tasks (including the total sum of times) the minimum time is always smaller in Java than in Groovy. However, this tendency does not hold for the maximum times, the arithmetic means, or the medians:

– **Maximum**: For the tasks CIT2, SEFT2 and CIT4 the maximum development time is larger for the Java solution.

Table 3 Descriptive statistics of experiment results (time in seconds for all but standard deviation), J=Java, G=Groovy

	CIT1		CIT2		CIT3		SEFT1		SEFT2	
	J	G	J	G	J	G	J	G	J	G
min	124	227	193	295	320	591	104	153	74	87
max	1609	2215	4285	1354	1983	2433	2910	3062	2264	2262
arith. mean	535	818	781	669	813	1182	1111	814	507	429
median	480	575	567	562	711	1010	1015	639	293	282
std. dev.	336	552	787	309	410	453	798	696	528	426

	CIT4		TEFT1		CIT5		TEFT2		Sums	
	J	G	J	G	J	G	J	G	J	G
min	323	537	78	149	293	564	44	246	1907	4072
max	3240	2505	900	2565	1514	2538	535	2285	14414	14557
arith. mean	827	1026	236	928	691	1112	147	849	5648	7827
median	716	880	197	813	671	1046	116	750	4892	7349
std. dev.	543	461	159	549	246	417	101	557	2925	2214

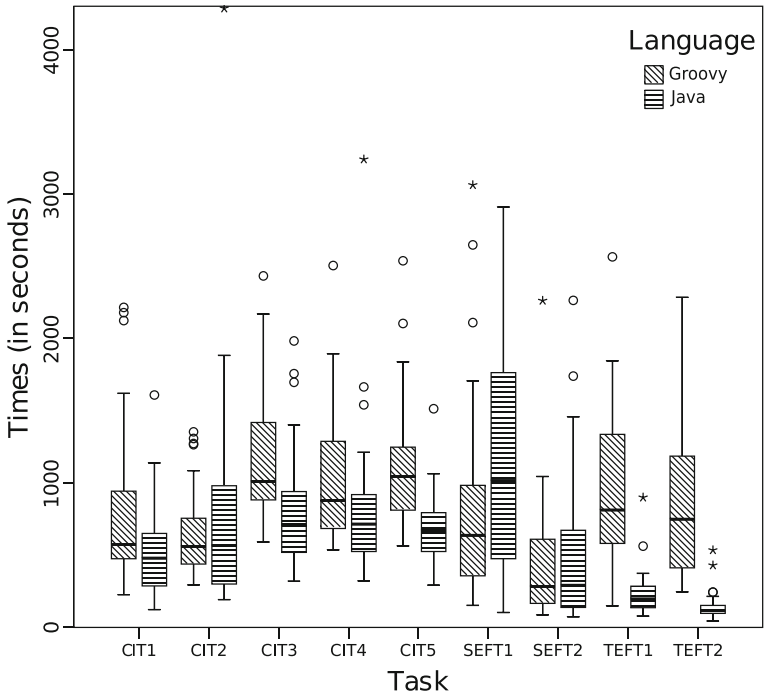

Fig. 2 Boxplot for measured data (ordered by kind of task)

- **Arithmetic mean**: For the tasks CIT2, SEFT1, and SEFT2 the arithmetic mean is larger for the Java solutions.
- **Median**: For the tasks CIT2, SEFT1, and SEFT2 the median for the Java solutions is larger.

Consequently, the first impression that the Java development times are always faster than the Groovy development times does not match closer inspection.

4.2 Repeated Measures ANOVA

We start the statistical analysis by treating each round of tasks separately. We first analyze round 1 of programming tasks (tasks 1–9) with all developer scores in the statistical model (i.e. those that solved the tasks in Java and those that solved the tasks in Groovy). Then, we do the same for the second round. The analysis is performed by using a Repeated Measure ANOVA, with two factors—*the programming task,* which is within-subject, and the *programming language,* which is between-subject. Since this analysis combines the different languages in each round, it benefits from the within-subject effect that each individual performs each task twice. Although developer performance is probably skewed, ANOVAs are, in general, quite stable against violations and since the number of data points (15*9) is relatively high, we simply apply this test here. Figures 3 and 4 show the boxplots for the two rounds. Apart from the type error fixing tasks (TEFT), for which there seems to be hardly a difference between both rounds, we see rather large differences for the other kinds of tasks in both rounds.

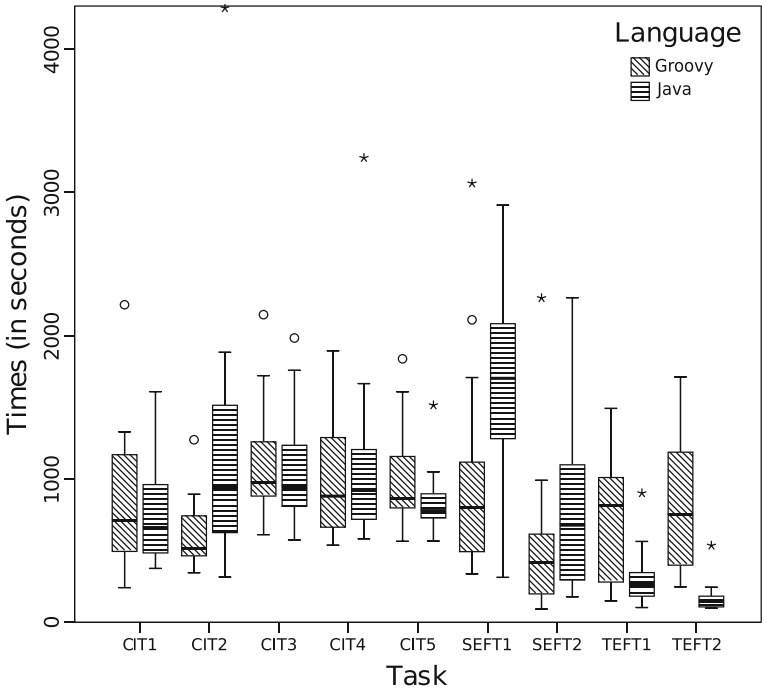

Fig. 3 Boxplot for first round (no repeated measurement of same tasks, ordered by kind of task)

Before conducting statistical tests, we ran Mauchly's sphericity test,[4] which was significant in both rounds. For those unfamiliar with a Mauchly test, it checks the condition where a repeated measures ANOVA violates sphericity, the condition where all tasks exhibit approximately the same variance. If this assumption is violated, a common correction, known as Greenhouse-Geisser, must be used on the results. SPSS can be configured to run such tests automatically when conducting a repeated measures ANOVA (e.g., across our nine tasks with the independent variable of type system). While the Mauchly test is standard, another way to think about this test is through a visual inspection of the boxplots for our tasks. Analyzing the boxplots makes it clear that the variance is probably different across many of the tasks, hence violating sphericity. Thus, analyzing the boxplots gives us a visual double check that a Mauchly test would likely be significant. Given the visual inspection, and the significant Mauchly, we used the standard Greenhouse-Geisser correction in reporting our Repeated Measure ANOVA results. Finally, the reader should realize that repeated measures ANOVA tests are very robust to minor violations in sphericity. Even if we had not followed standard procedure, using the Greenhouse-Geisser correction, the changes to the F-ratios would have changed little in our case, thus not changing our conclusions. Given our statistical procedures, results for our study show that:

1. The dependent variable of development time showed a significant difference in both rounds ($p < .001$, partial $\eta^2 = .275$ in the first round and $p < 0.001$, partial $\eta^2 = .246$

[4]Mauchly's sphericity test was performed using the standard packages implemented in SPSS. The used variables were the within-subject variable programming task and the between-subject variable programming language.

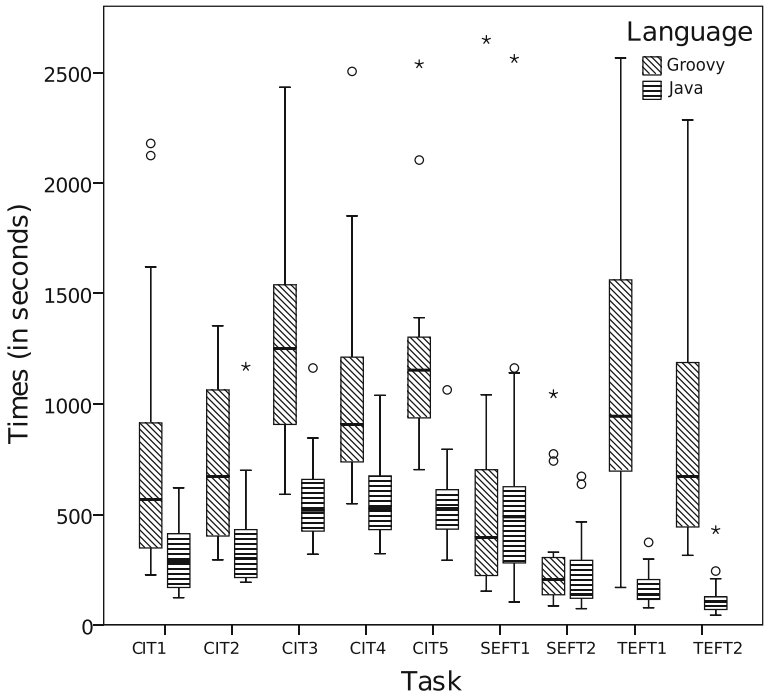

Fig. 4 Boxplot for second round (no repeated measurement of same tasks)

in the second round). In both cases the estimated effect size is comparable (.275 and .246, respectively).

2. There is a significant interaction between the factor programming task and the chosen programming language (in the first round $p < .001$, partial $\eta^2 = .181$ and in the second round $p < .001$, partial $\eta^2 = .172$). In both cases the estimated effect size is comparable. The significance indicates that the original motivation of this study holds—the effect of the programming language does vary for different programming tasks.
3. The impact of the between-subject factor programming language is non-significant for the first round ($p > .76$) and significant for the second ($p < .001$).

To summarize, different programming tasks influence the resulting development times. Furthermore, the resulting development times depend on the tasks as well as on the programming language. Hence, it is reasonable to analyze the different tasks and the languages in separation, with a within-subject study of each task.

4.3 Task-Wise and Group-Wise Analysis

To perform a within-subject analysis of each task, we combine, for each subject, the development times for both rounds. In other words, we separately compare the group starting with Java and the group starting with Groovy.

Figures 5 and 6 show boxplots for both groups. The groups are quite different. For the group starting with Groovy, Java has a clear positive impact. For the Java-first group, the effect is more nuanced. In all cases, we performed the non-parametric Wilcoxon-test. Table 4 gives the results of the test for the first group starting with Groovy ("Groovy first")

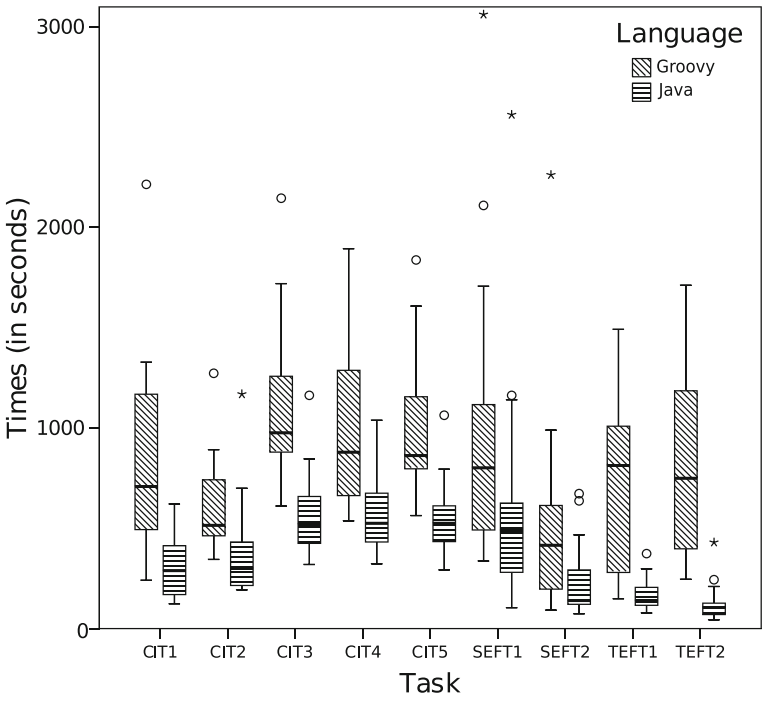

Fig. 5 Boxplot for group starting with Groovy

and the group starting with Java ("Java first") as well as their combination; to ease reading, the table reports the language with less development time instead of the raw rank sums.

We can see that for the group starting with Groovy, the effect of the programming language is always significant: in all cases, Java required less development time. Likely explanations are either the effect of the static type system *or* the learning effect from the experiment. For the group starting with Java, we observe a different result. For CIT2, SEFT1, and SEFT2 the subjects required less time with Groovy, while no significant impact of the programming language was found for CIT1, CIT3 and CIT4.

According to the procedure described in Section 3.4, we combine the results of the tasks for both groups. When one group reveals a significant result in favor of one type system, and the other group agrees or shows no significant difference, we conclude there is a significant difference. In case both groups show contradicting significant differences for one task, or neither shows significant differences, it is not possible to conclude in favor of any treatment. Combining both results, we obtain a positive impact of Java for: all Type Error Fixing Tasks (TEFT); all Class Identification Tasks (except CIT2); and no impact on the semantic error fixing tasks.

5 Discussion

The experiment revealed a positive impact of the static type system for six of nine programming tasks: For all tasks where type errors needed to be fixed (TEFT) and for most of the tasks where new classes need to be used (CIT), but no difference for semantic error

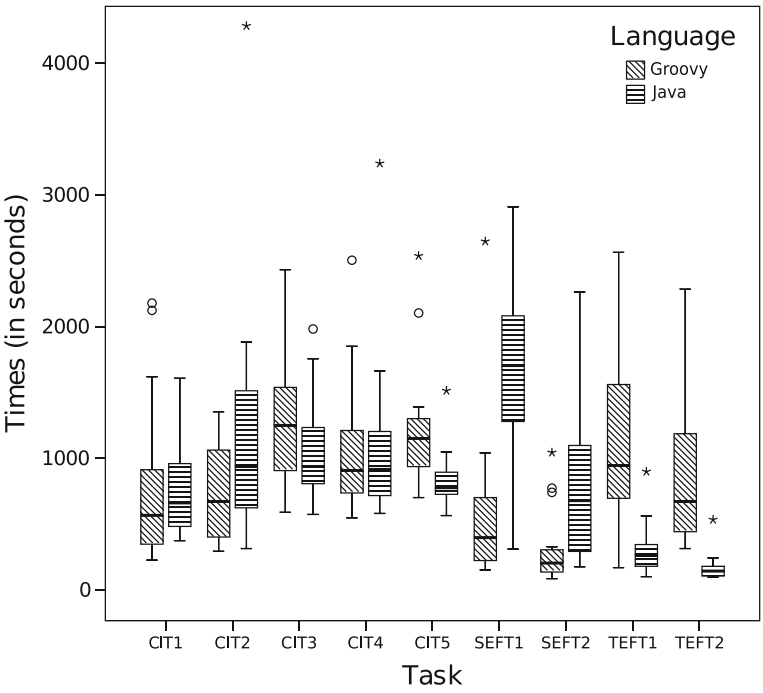

Fig. 6 Boxplot for group starting with Java

fixing tasks (SEFT). From this perspective, the experiment provides evidence that static type systems benefit developers in situations where a set of classes has to be used where documentation is limited or not available (which we suspect is quite common). An important implication of the experimental results is that no single task could be used to argue in favor of dynamic type systems—the best human performance with Groovy statistically tied human performance with Java.

Further, while no statistically significant differences were observed with regards to tasks involving debugging semantic errors, it is possible that learning effects masked any potential results.

In CIT2, the group starting with Groovy was faster with the statically typed solution, while the group starting with Java was faster with the dynamically typed solution. Following

Table 4 Wilcoxon-Test for the within-subject comparison of measured times

Groovy first									
Task	CIT1	CIT2	CIT3	CIT4	CIT5	TEFT1	TEFT2	SEFT1	SEFT2
p-value	.000	.001	.001	.000	.000	.000	.000	.028	.001
benefit	Java	Java	Java	Java	Java	Java	Java	Java	Java
Java first									
p-value	.91	.034	.215	.679	.003	.001	.001	.001	.003
benefit	–	Groovy	–	–	Java	Java	Java	Groovy	Groovy
Result									
benefit	Java	–	Java	Java	Java	Java	Java	–	–

the same argumentation as before, this likely means that the learning effect was larger than the (possibly) positive effect of the static type system. However, it was not obvious from our observations why this would be the case. Ultimately, the principle for solving CIT2 was the same as for CIT1, 3, 4, and 5. In all these cases, the developer had to use a number of new classes that were not known upfront. In CIT2, a relatively small number of new classes had to be identified (four types)—which is fewer than for CIT3, 4, and 5, but more than for CIT1. Hence, it is not sufficient to argue that this task is different because of the number of classes to be identified.

While we can not be certain of the reason, we offer several hypotheses that would explain this behavior:

- The increase in difficulty between CIT1 and CIT2 is not as large as it seems: For task CIT1, two types needed to be identified; for task CIT2, four. However, looking at the source code of the solutions in Table 11, we see that CIT1 needs to instantiate two slightly different Pipelines or GameQueues, while for CIT2, two of the types to instantiate are very similar (MailStartTag and MailEndTag, respectively StartLevelField and GoalLevelField). This means that CIT1 requires to identify more than strictly two types, while CIT2 requires less than strictly four types, reducing the overall difference of difficulty between the two tasks. Additionally, we think that the method names in the task might have given the developers too clear a hint about which classes to use. For instance, the methods *setStart* and *setGoal* in the Groovy code for task 2 seem to be more closely associated with the necessary types *StartLevelField* and *GoalLevelField* in comparison to task one (where *setTasksAndMessages* required a *Queue* object).
- Part of the observed advantage in CIT1 may be due to the novelty effects of the subjects' use of Groovy. Despite the presence of a warm-up task, we observed this phenomenom in our previous experiment (Mayer et al. 2012), so we can not exclude it here. This is supported by the large spread of time in the first boxplot of Fig. 2, where the values for CIT1 in Groovy are larger than for CIT2. We observe the same behavior in all the time measurement boxplots, where the boxplot for CIT1 for Groovy is systematically "wider" than the one for CIT2.[5]
- The unusually large amount of arguments, including 4 primitive types, may have distracted the subjects. It is possible that subjects spent an equal amount of time understanding the large number of arguments, which would make the difference between the treatments less pronounced.

The case of task CIT2 implies that the argumentation for or against static types cannot be trivially reduced to the question of how many (unknown) classes are needed in order to solve a programming task. There seem to be other factors that need to be identified. We think that identifying these factors is an important venue of future work, of which a first step is presented in the next section.

6 Exploratory Study

While measuring time is sufficient to gather the insights we set out to discover, time alone is not suitable to understand the possible *reasons* of the observed differences between statically and dynamically typed assignments. If we understand what causes people to spend

[5]If we anticipate in the next section, we also see that in the numbers of test runs in Fig. 7, where test runs for Groovy are significantly worse for the supposedly simpler CIT1 than they are for CIT2.

more time using one technique, it might be possible to adapt one of these techniques so that the cause will vanish or that the factor is weakened. This could provide programming language designers with a roadmap for how type systems could, or maybe should, be designed to maximize human performance. Additionally, it would give us some understanding about whether under different circumstances the effect could be larger or smaller.

We now exploit additional measurements gathered during the experiments in order to better understand the differences. The additional measurements that we use in the exploratory study are:

- **Number of test runs:** The number of test runs describes how often the subjects tried to start the tests associated with the programming tasks. The result of trying to start the test runs is that the code is getting compiled and—in the absence of compilation errors—executed. In case the code contains compile-time errors, the code does not execute, but it still counts as a test run. In essence, a test run is a static check followed by a dynamic check. The motivation for this measure is that the static type information may be helpful to identify more errors in the code earlier, at compile time, thanks to the static analysis performed by the compiler, whereas runtime errors are harder to identify and may need several test runs to be fixed. As a result, users can find and correct errors easier every time they compile or run the code—with the plausible expected consequence that the statically typed solutions required fewer test runs than the dynamically typed solutions.
- **Number of files opened:** We measured the number of opened files for each task, independently of how often each file has been viewed by the subject. The underlying motivation for this measurement is that, if the static type system helps developers to directly identify the classes they need in order to solve the programming tasks, the subjects will probably need to open fewer files than developers who work on the dynamic type system, who will need a more exhaustive search. Overall, we expect users of static type systems to perform less program exploration than users of the dynamic type system.
- **Number of file switches:** Finally, we measured how often developers switch between the different files while solving the programming tasks—where a file switch means that the developer who is currently viewing one file needs to view a different file in order to continue the task. Consequently, in contrast to the number of open files, this metric measures specific files more than once. We expect that it is more likely that developers will switch more often between different files (potentially back and forth) if they do not have the static type information. We assume that static type systems give developers more accurate information about the entities referenced in a file (e.g., the type of arguments or the return type), whereas users of the dynamic type system may need to find that information in the definition of the entity itself more often.

In our previous experiment (Mayer et al. 2012), analyzing these measurements provided us with valuable additional insights. Hence, we repeat the measurements and compare to what extent they might explain the effect measured in this experiment. We are particularly interested in the measurements concerning the class identification tasks, which are similar to the tasks in Mayer et al. (2012). Additionaly, we perform the same study for the type error and semantic error tasks to shed more light on the results.

For each additional measurement, we perform analyses similar to the ones we did before. We present the raw data for each measurement—test runs in Table 13, files opened in Table 14, and file switches in Table 15. We also present the results of performing first a

repeated measures ANOVA and the results of a Wilcoxon test on each individual programming task. In order to give a visual representation of the data, we provide boxplots (instead of giving a boxplot for each individual group).

Differences with the Previous Analysis Note that this analysis is finer than the analysis we adopted in Section 3.6, in that some results that would be deemed inconclusive are shown as trending in one direction. We mark a result as trending when the p-value from the statistical test is between 0.05 and 0.15. Due to the exploratory nature of the study, we are less stringent on the significance level of the observed results for the test runs, file switches, and files opened indicators. In the tables below, trending results are highlighted by showing the language between parentheses.

6.1 Repeated Measures ANOVA

To test whether there were statistically-significant differences between groups using either static or dynamic typing, we conducted a series of Repeated Measures ANOVA for the three measurements; this led to a total of six ANOVA tests. In all six cases, Mauchly's test for sphericity was significant, implying that the Greenhouse-Geisser correction must be used for our data. We used this correction in all cases when reporting results.

For each of the six ANOVAs, we looked at three results: 1) the within-subjects effect of the task, 2) the within-subjects effect of task and language interaction, and 3) the between-subjects effect of language itself—the latter is the most relevant because it concentrates on the main research question.

First, we found statistically significant differences across the within-subjects effects, for all tests. This is not surprising, as it implies there were differences in tasks. Second, in most cases, the within-subjects interaction was also significant, implying that for some metrics, complex interactions existed between the tasks and the language that was used. We flesh out the details of these interactions using different tests below.

Finally, the between-subjects effects exhibited some significant, and some non-significant effects. Specifically, on the first set of nine tasks, we see no significant differences for the total number of file switches or the total number of opened files, but significant and moderately sized differences on the number of file runs ($\eta_p^2 = .350$). For the second set of tasks, we observe very large differences in the number of file runs ($\eta_p^2 = .688$), in addition to moderate to large differences in file switches ($\eta_p^2 = .456$), and in the number of open files ($\eta_p^2 = .454$). This implies that between 35 % and 69 % of the differences between those measurements can be explained by the treatment type system.

6.2 Number of Test Runs

The boxplot for the number of test runs (see Fig. 7) shows an obvious difference in favor of Java—i.e., fewer test runs—for the tasks CIT3, CIT4, and CIT5, as well as for the type error fixing tasks TEFT1 and TEFT2. For the semantic errors as well as for the first two class identification tasks, it is unclear whether there is a difference at all.

The comparison of the number of test runs on a task by task basis for each individual group (see Table 5) reveals some similarities to the time measurements: the group starting with Groovy shows significant favorable results for Java, for all class identification tasks as well as for the type error fixing tasks. For the semantic errors, no significant differences between Java and Groovy were measured (although the results are close to significant with

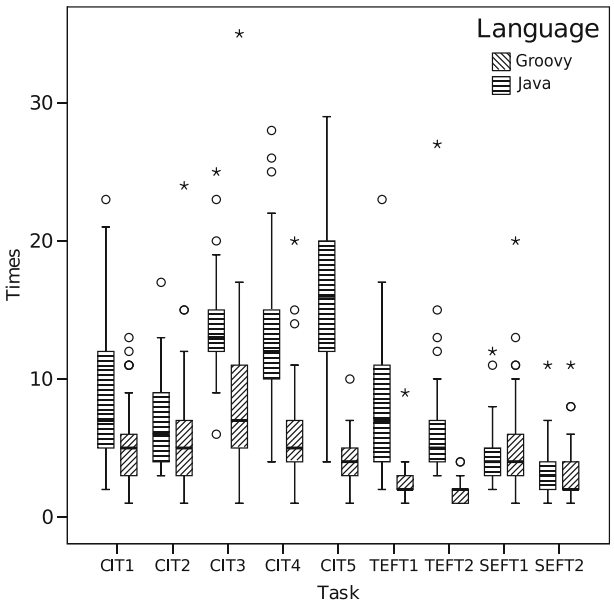

Fig. 7 Boxplot for number of test runs

the tendency that Java requires fewer test runs). For the Java-first group, the results are slightly different: we see significant differences pro Java for the class identification tasks CIT4 and CIT5 (and a tendency pro Java in CIT3), and we see (again) a positive impact of Java for the type error fixing tasks. For the semantic errors, we find one task where a positive significant difference of Groovy is found (SEFT1) whereas SEFT2 shows a tendency pro Groovy.

According to our previous argumentation, we see a positive impact of the static type system on the class identification tasks, and a positive impact of the static type system on the type fixing tasks. Note that in contrast to the time measurement, the number of test runs is not inconclusive for CIT2. For semantic errors, the results of both groups tend to contradict each other, similarly to the time measurements. As such, no clear conclusion can be drawn (although a slight positive impact of the dynamic type system could be measured for one of the tasks). There may be a language effect, but it seems offset by the learning effect.

Table 5 Wilcoxon-Test for the within-subject comparison for number of test runs

Groovy first									
Task	CIT1	CIT2	CIT3	CIT4	CIT5	TEFT1	TEFT2	SEFT1	SEFT2
p-value	.001	.003	.000	< .001	< .001	.002	< .001	.072	.061
less runs	Java	Java	Java	Java	Java	Java	Java	(*Java*)	(*Java*)
Java first									
p-value	.975	.280	.098	.011	< .001	.001	.001	.007	.067
less runs	-	-	(*Java*)	Java	Java	Java	Java	Groovy	(*Groovy*)
Result									
test runs	Java	Java	Java	Java	Java	Java	Java	(*Groovy*)	–

Italics indicates that the observed effect approached significance

6.3 Number of Files Opened

We now focus on the number of files opened by the subjects during the experiment (Fig. 8). Compared to the time and number of test runs measurements, we see similarities: CIT1 and CIT4 show a language effect and CIT 3 shows it as well, even if the boxplots overlap somewhat; the effect is much less pronounced for CIT2 and CIT5. Tasks TEFT1 and TEFT2 reveal again a large effect; it seems obvious that Java required less files to be opened. The semantic error tasks SEFT1 and SEFT2 do not show large differences, but a trend in favor of Groovy.

The comparison of the number of files opened on a task-by-task basis (Table 6) reveals a positive impact of the static type system for the group starting with Groovy for 3 out of 5 class identification tasks (except CIT2 and CIT5). For the type error fixing tasks, both tasks required less opened files in Java. For the semantic errors fixing tasks, no significant difference can be seen (but a trend in SEFT1). For the group starting with Java, no significant differences can be seen for the class identification tasks (with trends favorable to Java in CIT1 and CIT5). For type fixing error, we still see a significant effect in favor of Java even in the Java-first group, while for the semantic error tasks, Groovy turned out to require less files to be opened.

According to our previous argumentation, we see: less files opened for only three out of five type identification tasks with the static type system; less files opened with the static type system for type error fixing tasks; and less files opened for the dynamic type system, for the tasks where semantic errors need to be fixed.

6.4 Number of File Switches

Considering the boxplots for the number of file switches (Fig. 9), we observe indications of a language effect, although we see more overlap than in previous boxplots. The differences

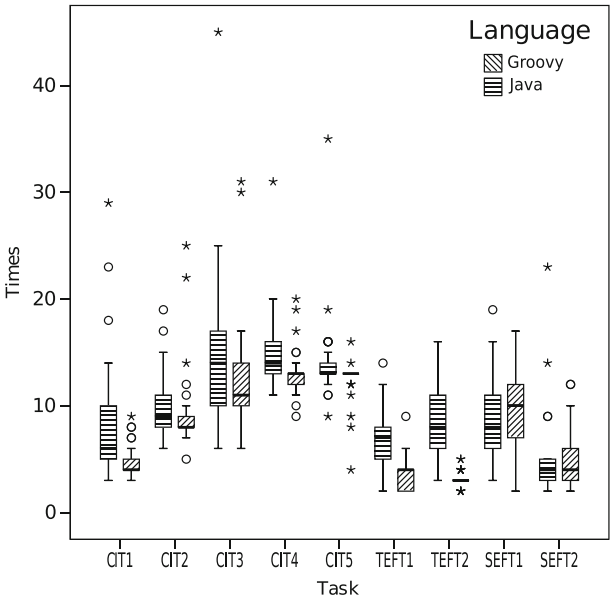

Fig. 8 Boxplot for number of files opened

Table 6 Wilcoxon-Test for the within-subject comparison for number of files opened

Groovy first									
Task	CIT1	CIT2	CIT3	CIT4	CIT5	TEFT1	TEFT2	SEFT1	SEFT2
p-value	.001	.377	.043	.001	.060	.001	.000	.073	.206
less files opened	Java	–	Java	Java	(*Java*)	Java	Java	(*Java*)	–
Java first									
p-value	.076	.806	.443	.977	.120	.001	.000	.001	.005
less files opened	(*Java*)	–	–	–	(*Java*)	Java	Java	Groovy	Groovy
Result									
less files opened	Java	–	Java	Java	(*Java*)	Java	Java	(*Groovy*)	Groovy

between CIT1 up to CIT4 are less large; for task CIT5, there is less overlap, indicating that a significant difference is possible. With regard to the type error tasks, the differences are obvious as usual, while for the semantic errors there seems to be a tendency towards Groovy requiring less file switches.

Considering the results from the Wilcoxon test (Table 7), the Groovy-first group shows almost the same result as the number of test runs: no obvious differences between Java and Groovy for the semantic errors (with however a tendency that Java required less file switches); fewer file switches using Java for the type fixing tasks; and, for the class identification tasks, only CIT3 does not show a significantly reduced number of file switches

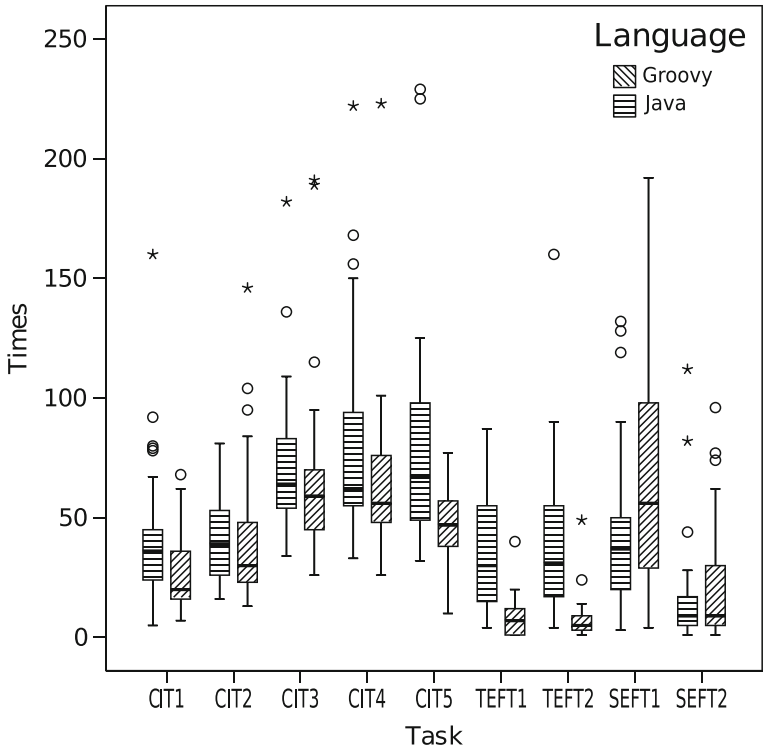

Fig. 9 Boxplot for number of file switches

Table 7 Wilcoxon-Test for the within-subject comparison for number of file switches

Groovy first									
Task	CIT1	CIT2	CIT3	CIT4	CIT5	TEFT1	TEFT2	SEFT1	SEFT2
p-value	.000	.015	.113	.003	.002	.001	.001	.107	.098
less switches	Java	Java	*(Java)*	Java	Java	Java	Java	*(Java)*	*(Java)*
Java first									
p-value	.939	.093	.796	.679	.002	.001	.001	.001	.004
less switches	–	(Groovy)	–	–	Java	Java	Java	Groovy	Groovy
Result									
less switches	Java	(Java)	(Java)	Java	Java	Java	Java	(Groovy)	(Groovy)

using Java (only a trend). For the Java-first group, we see a slightly stronger parallel with the time measurement: for task CIT2 there is a trend in favor of Groovy, and a significant result in favor of Java in CIT5. Likewise, we see significant effects towards Java for the type error tasks, and towards Groovy for the semantic error tasks. Consequently, it cannot be concluded that all tasks required less file switches using Java, due to CIT2, CIT3, SEFT1, and SEFT2.

6.5 Insights on the Experiment from the Exploratory Study

We begin by summarizing the measurements from the experiment and the exploratory study: Table 8 sums up the results. We adopt the following conventions: "J" stands for an observed effect favoring Java, "G" for an observed effect favoring Groovy, and "—" for inconclusive results. To observe an effect, we require at least a significant result in one of the group that is not contradicted by a significant effect in the other group; a "+" indicates a significant effect observed in both groups, or a significant effect and a trend in the same direction; a "–" sign indicates a significant effect that was contradicted by a trend in the opposite direction, or a trend in one direction that is not contradicted. We start by describing the error identification tasks, as their interpretation is more straightforward.

Type Error Tasks (TEFT1 and TEFT2) All additional measurements show significant effects of the static type system, and thus support the result of the original time measurements—overwhelming evidence favoring the static type system. All three measurements might reveal the cause of the differences in the time measurements. We conclude that for developers, the static type system requires less navigation through the code, and less

Table 8 Summary of measurements we performed, and our expectations before carrying the the experiment

Aspect	CIT1	CIT2	CIT3	CIT4	CIT5	TEFT1	TEFT2	SEFT1	SEFT2
Less Time	J	—	J	J	J+	J+	J+	—	—
Less test runs	J	J	J+	J+	J+	J+	J+	G–	—
Less files opened	J+	—	J	J	**J–**	J+	J+	G–	**G**
Less file switches	J	J–	J–	J	J+	J+	J+	G–	G–
Expected	J	J	J	J	J+	J+	J+	—	—

Unexpected results are shown in bold

testing in order to identify and fix the error. Of course, this is plausible and a consequence of the very nature of static type systems: with a static type system, the compiler points out the exact location of a type error. On the other hand the dynamic type system finds the error at runtime, and the symptoms of the error may appear far from the actual cause of the error. For type errors, the navigation effort of developers is much higher in dynamically typed systems.

Semantic Error Tasks (SEFT1 and SEFT2) The time measurements show that dynamic and static type systems do not impact the time required to fix semantic errors, as expected. However, the other indicators seem to point towards the dynamic type system—admittedly somewhat weakly for SEFT1, and in the case of SEFT2, more strongly for opened files. We are not sure why we observe this behavior: it may be that test runs, files opened, and file switches are simply not reliable indicators of the program comprehension process when subjects are fixing semantic errors. We can think of two alternative explanations. The first is that dynamic type systems enforce different cognitive processes that let developers explore the project in a different way and to go less often to different locations in the code (in terms of files, etc.). The second is the presence of a between-task learning effect: subjects using the dynamic language are slightly more efficient at debugging, since they practiced it in previous tasks. Although this is speculative, it is an interesting avenue of future investigation. Altogether, we can conclude that the further measurements do not provide meaningful explanations for the measured time: while the time measurement did not reveal any differences between static and dynamic type systems, the further measurements have the tendency to do so.

Class Identification Tasks (CIT1 to5) From the time measurements we expected to find a difference favoring Java. For all tasks except CIT2, all the indicators are roughly in agreement with an advantage for Java. For task CIT2, we find more mixed results, as both time and files opened measurements are inconclusive. However, we also find weak evidence in favor of Java for file switches, and a significant difference for the number of test runs. Our expectation—that the dynamic type system enforces an exploration of the source code that uses additional resources more often—seems to be supported by the file switches and test runs, but this is not reflected in the time measurement. The simple explanation that the learning effect was too strong does not seem to hold: in that case, the other measurements would also manifest a benefit in the second language being used, and be all inconclusive. Since this result concerns a single task, it might be the case that subjects explored the dynamically typed code in the same way as for the other tasks—but find the desired solution more quickly for a task-specific reason in the Java-first group.

Another surprising indicator is the weak tendency in favor of Java in terms of opened files for CIT5. We would expect a much stronger tendency. However, task CIT5 requires the identification of 12 different types. Given such a high miminum number of types to identify, it is conceivable that in any case, subjects solving this task did open a large proportion of the files in the system regardless of the treatment (the system has 30 classes), dimming the potential differences between them.

Altogether, we can argue that the number of file switches turn out to be a good indicator for the measured times (this result corresponds to the result in Mayer et al. (2012)). One interpretation is that static type systems reduce the navigation costs (because developers do not have to search in the code what classes are required by the API) and cause in that way a reduction of the development time for such kinds of programming tasks.

6.6 Comparison with Previous Study

To compare our set of indicators with those of our previous experiment (Mayer et al. 2012), we start by recalling them in Table 9.[6]

We employ the same convention as before, except that this previous experiment did not distinguish between weak, normal, and strong observed effect (i.e., the table does not contain any "+" or "–"). The tasks in this experiment were class identification tasks, with some types harder to identify than others (for instance, types parametrized by other types, such as generics, are harder to identify than simple types). The number in the name of the task refers to the number of types to identify. In that experiment, the overall conclusions of the study were:

- The effects are more pronounced when there are more types to identify, or when they are more difficult to implement.
- The first task Easy 1 may have been subject to a warm-up effect for Groovy users, explaining the advantage for Java.
- For simpler tasks such as Easy 3 and Medium, a "brute-force" comprehension strategy may be faster in solving the task. This comprehension strategy involves more test runs and files opened for Groovy users, whereas Java users are more systematic and slower. Task Medium 3 shows this clearly as test runs and files opened are higher in Groovy, yet the overall time is lower.
- For more difficult tasks, this strategy incurs more file switches and ends up being slower than the more systematic approach, guided by type annotations, employed by subjects using Java. Overall, file switches was the indicator most consistent with time.

Taking this into account, we can see some similarities with the present study, if we take into account the differences between the tasks in the study: there are variations in the number of types to identify, as well as their difficulty. We consider all the types to identify in CIT1 to 5 to be of medium difficulty, as all tasks feature types parametrized by other types. The number of types to identify are respectively 2, 4, 6, 8, and 12. As such, task Medium 3 from our previous experiment would fall in between CIT1 and CIT2 in terms of complexity. Given our remarks above on the complexity of CIT1 (potentially harder than it seems) and of CIT2 (potentially easier than it seems), we could see all three tasks as roughly equivalent. Task Easy 6 would be, in number of types, equivalent to task CIT3, although the types to identify are more complex in CIT3. Finally, CIT4 and CIT5 are more complex. Comparing with task Hard 3 from our previous experiment is difficult, since the types involved there were much more complex, involving complex nested generic types.

Furthermore, we mentioned earlier that we suspect that, as with task Easy 1, CIT1 may be subject to a warm-up effect as subjects were not familiar with Groovy. This makes us reduce the weight of the evidence towards Java in this particular task. Given all this, if we consider tasks CIT2–5, we see that:

- Similarly to our previous experiment, all of the indicators (time, test runs, open files, file switches) tend to be more pronounced in favor of Java as the difficulty of the tasks increase.
- Test runs are the weakest match with the time measurement, while file switches are the strongest match.

[6]Note that the programming tasks in Mayer et al. (2012) were exclusively CIT tasks. Hence, a comparison is only based on the CIT tasks in the here described experiment.

Table 9 Summary of measurements we performed in our previous experiment (Mayer et al. 2012)

Aspect	Easy 1	Easy 3	Medium 3	Hard 3	Easy 6
Less time	**J**	**G**	**G**	J	J
Less test runs	—	**G**	J	J	—
Less files opened	**J**	**G**	J	J	J
Less file switches	**J**	**G**	**G**	J	J
Expected	—	J	J	J	J

- The number of files opened indicator exhibited a different tendency for the last task, but this can be explained by the large number of types to identify, which, as we mentioned above, would dilute the differences between the two treatments.
- The fact that, for task CIT2, the evidence towards Java is not yet clear-cut matches to some extent the situation of task Medium 3, which was of a similar difficulty, although none of the indicators manifest an advantage in favor Groovy. Also, contrary to the earlier experiment, files opened and file switches behave differently.

Overall, we do observe some differences with our previous exploratory study, but not very large ones. It seems that tasks of the complexity of tasks Medium 3 and CIT2 are tasks for which the productivity benefits of dynamic type systems starts to wear off in favor of static type systems. To confirm this, we may need to carry out an experiment focused on simpler tasks, featuring several tasks with complexity between that of Easy 1 and CIT3. We believe that by now, the spectrum of more difficult tasks has been much more covered.

6.7 Summary

We introduced three additional measurements beyond raw time (number of test runs, files opened, and file switches), in order to shed light on our time measurements. We also compared our results with the exploratory study we performed earlier (Mayer et al. 2012).

In general, the measurements in the exploratory study point to the same conclusions as measuring developement times, although there were some differences among the types of tasks:

- For class identification tasks, file switches is the closest matching indicator to time. We speculate that the cause for the difference in time is the navigation and comprehension effort required by dynamically typed systems—of which the file switches are an indicator.
- For type errors, all the indicators unsurprisingly showed a very strong effect of the static type systems.
- For semantic errors, the indicators were more ambiguous: no effect was found of the type system on time, which we expected; however, some other indicators were slightly in favor of the dynamic type systems, for reasons that are still unclear.

Although we have determined so far only that there is a relationship between those measurements, it is unclear whether this relationship is causal. Of course, this experiment is not able to give any definitive answer to this question. However, especially for type identification tasks we have seen with the number of file switches a first indicator that seems to be worth to be analyzed in more detail in future experiments.

7 Summary and Conclusion

Although there is a long on-going debate about the advantages and drawbacks of static type systems, there is little empirical data available on their usage by human developers. This paper has presented an experiment analyzing the potential benefit of static type systems, complemented by an exploratory study of additional data gathered during the experiment. The experiment investigated two commonly used arguments in favor of type systems: that they assist the programmer in finding errors, and that they act as implicit documentation. Three kinds of programming tasks—all of them potential maintenance tasks—were given to 33 subjects: tasks where a set of previously unknown classes had to be used by the developers, tasks where developers had to fix type errors, and tasks where developers had to fix semantic errors. Altogether nine programming tasks were given to the subjects.

Each subject completed the programming tasks twice: with a statically typed language and with a dynamically typed one. For the statically typed language, the subjects used Java, while for the dynamically typed language they used Groovy. The result of the experiment can be summarized as follows:

- **Static type systems help humans use a new set of classes:** For four of the five programming tasks that required using new classes, the experiment revealed a positive impact of the static type system with respect to development time. For one task, we did not observe any statistically significant difference (possibly due to learning effects).
- **Static type systems make it easier for humans to fix type errors:** For both programming tasks that required fixing a type error, the use of the static type system translates into a statistically significant reduction of development time.
- **For fixing semantic errors, we observed no differences with respect to development times:** For both tasks where a semantic error had to be fixed, we did not observe any statistically significant differences.

The results of the additional exploratory study, where we analyzed the number of test runs, the number of files opened, and the number of file switches, can be summarized as follows:

- **The number of file switches is a good indicator for measured time differences in class identification tasks:** It looks like static type systems—which reduce the development time for such class identification tasks—reduce the number of file switches for such tasks.
- **The number of test-runs, the number of files opened, as well as the number of file-switches are good indicators for type error fixing tasks:** All three measurements show the same results as the time measurements—that static type systems reduce the development time for such tasks.
- **The number of test-runs, the number of files opened, as well as the number of file-switches seem to be inappropriate for semantic error fixing tasks:** All three measurements show to a certain extent some tendencies—which were not detected by the time measurement.

We believe that the most important result is that the static type systems showed a clear tendency in class identification tasks, and that we found a first indicator that this is caused by a reduced navigation effort. This makes the result closely related to the study by Ko et al. (2006) who measured in an experimental setting that 35 % of the development time was spent on navigating through the code. It seems plausible to assume that—due to the type annotations in the code—developers do not have to search in a large number of different

files how they can use a given (undocumented) API. Instead, the type annotations directly guide them to the right place. This implies a reduction of the navigation costs.

It must not be forgotten that this experiment (as every experiment) suffers from a number of threats to validity. Here, we should focus on one in particular: the experiment was based on the programming language Java—a programming language which is not known for having the most expressive type system. It would be interesting to see whether for other languages (such as Haskell) the results are comparable. Another issue is the question to what extent only the type annotations (and not the type checks) are responsible for the measured differences. Both issues are avenues for future work.

We have presented evidence that programmers do benefit from static type systems for some programming tasks, namely class identification tasks and type error fixing tasks. For the first kind of task we have seen that the reduced navigation effort (caused by the static type system) might be the cause for this benefit. This result hardly implies that static systems benefit programmers for all tasks—a conclusion that seems unlikely. However, we think the onus is now on supporters of dynamic typing to make their claims with rigorously collected empirical evidence with human subjects, so the community can evaluate if, and under what conditions, such systems are beneficial.

Appendix: A Raw Measurement Data

Table 10 Example solutions for semantic error fixing tasks (SEFT) and type error fixing tasks (TEFT)

	Faulty code	**Corrected Code (Solution)**

SEFT1	`void doCursorOnInteraction() {`
	`  Job job = new SetCursorJob(`
	`  cursorOnElement,`
	`  MailEditorServer.getInstance().`
	`  getCurrentDocument(), 0, 0);`
	`  MailEditorServer.getInstance().`
	`  addToActions(job);`
	`}`
		`void doCursorOnInteraction() {`
		`  Job job = new ChangeMailJob(this);`
		
		`  MailEditorServer.getInstance().`
		`  addToActions(job);`
		`}`

SEFT2	`...`
	`if (newField.setSubject(subject)) {`
	`  subject.setPosition(`
	`  newField.getX_position(),`
	`  newField.getY_position());`
	`  newField.subjectInteraction(`
	`  InteractionType.Move);`
	
	`}`
	`...`
		`...`
		`if (newField.setSubject(subject)) {`
		`  subject.setPosition(`
		`  newField.getX_position(),`
		`  newField.getY_position());`
		`  newField.subjectInteraction(`
		`  InteractionType.Move);`
		`  oldField.removeSubject();`
		`}`
		`...`

TEFT1	`// ERROR: Wrong Class GameObject`
	`dartTrapProperties.setHitByTrapMessage(`
	`  new GameObject(symbol,`
	`  "hit by a dart trap", 10));`
	`...`
	`// Code where the error shows up`
	`def getHitByTrapMessage(def subject) {`
	`  def message = myProperties.`
	`  getHitByTrapMessage().`
	`  concat(" " + subject.getName());`
	`  return message;`
	`}`
		`// Remove GameObject`
		`dartTrapProperties.setHitByTrapMessage(`
		`  "hit␣by␣a␣dart␣trap");`
		`...`
		`...`
		`// No changes here`
		`def getHitByTrapMessage(def subject) {`
		`  def message = myProperties.`
		`  getHitByTrapMessage().`
		`  concat("␣" + subject.getName());`
		`  return message;`
		`}`

TEFT2	`static def getTeleportCommand(`
	`  def subject, def level, def x, def y) {`
	`  return new TeleportCommand(`
	`  level, subject, x ,y);`
	`}`
		`static def getTeleportCommand(`
		`  def subject, def level, def x, def y) {`
		`  return new TeleportCommand(`
		`  subject, level, x ,y);`
		`}`

Table 11 Example solutions for class identification tasks

	Java solution	Groovy solutions

CIT1	`void initializeServer(`
	`  MailEditorServer server) {`
	`  Pipeline<String> stringPipe =`
	`  new Pipeline<String>();`
	`  Pipeline<Job> jobPipe =`
	`  new Pipeline<Job>();`
	`  ActionsAndLoggerPipe actionsLoggerPipe =`
	`  new ActionsAndLoggerPipe(`
	`  stringPipe, jopPipe);`
	`  server.setActionsAndLogger(`
	`  actionsLoggerPipe);`
	`}`
		`def configureManager(`
		`  def manager) {`
		`  def messages =`
		`  new GameQueue();`
		`  def commands =`
		`  new GameQueue();`
		`  def tasksAndMessages =`
		`  new TaskAndMessageQueue(`
		`  messages, commands);`
		`  manager.setTasksAndMessages(`
		`  tasksAndMessages);`
		`}`

CIT2	`void setMailStartEnd(EMailDocument email,`
	`  int startX, int startY,`
	`  int endX, int endY) {`
	`  email.setStartElement(new MailStartTag(`
	`  new CursorBlockPosition(`
	`  startX, startY)));`
	`  email.setEndElement(new MailEndTag(`
	`  new CursorBlockPosition(endX, endY)));`
	`  email.setFormat(new UTF8Encoding());`
	`}`
		`void configureLevel(def level,`
		`  def startX, def startY,`
		`  def goalX, def goalY) {`
		`  level.setStart(new StartLevelField(`
		`  new Position(startX, startY)));`
		`  level.setGoal(new GoalLevelField(`
		`  new Position(goalX, goalY)));`
		`  level.setLevelKind(`
		`  new DungeonLevelType());`
		`}`

CIT3	`MailElement intializeElement(`
	`  int x_position,`
	`  int y_position, char headerType)`
	`  throws InvalidHeaderException {`
	`  Header header = new Header(headerType);`
	`  OptionalHeaderTag newTag =`
	`  new OptionalHeaderTag(x_position,`
	`  y_position, header);`
	`  newTag.setElementInfo(`
	`  new DataList<MetaData>());`
	`  newTag.setCursor(new DefaultCursor(`
	`  new MetaDataCache(),`
	`  new MetaDataDisplay()));`
	`  return newTag;`
	`}`
		`def setUpLevelField(`
		`  def x_position`
		`  def y_position, def trapType)`
		`  throws InvalidTrapSymbolException {`
		`  def trap = new Trap(trapType);`
		`  def trapField =`
		`  new TrappedLevelField(x_position,`
		`  y_position, trap);`
		`  trapField.setItems(new GameList());`
		`  trapField.setSubject(new Player(`
		`  new Inventory(),`
		`  new Body()));`
		`  return trapField;`
		`}`

CIT4	`Cursor createPointer() {`
	`  WorkInProgressPresentation progressRep`
	`  = new`
	`  DefaultWorkInProgressPresentation(`
	`  Animation.HourGlass);`
	`  CursorFeatures features =`
	`  new CursorFeatures(progressRep,`
	`  new IdleRepresentation());`
	`  Theme theme = new Theme(`
	`  new ThemeLocator());`
	`  WindowsMousePointer pointer =`
	`  new WindowsMousePointer(`
	`  features, theme);`
	`  pointer.setTipOfDayPopup(`
	`  new ShowTipEventManager());`
	`  return pointer;`
	`}`
		`def createNewActorForGame() {`
		`  def attackType =`
		`  new`
		`  UnarmedAttackType(`
		`  DamageType.default);`
		`  def attributes =`
		`  new SubjectAttributes(attackType,`
		`  new Resistances());`
		`  def monster = new HillGiant(`
		`  attributes,`
		`  new Intrinsics(), new Giants());`
		`  monster.setDroppableItemGenerator(`
		`  new RandomItemBuilder());`
		`  return monster;`
		`}`

CIT5	`MailAccount createNewUserPrincipalAndAccount(`
	`  String userName, String password) {`
	`  MailFormatter mailDOMCreator =`
	`  new MailFormatter();`
	`  MailFormatReader rawFileInputReader`
	`  = new MailFormatReader();`
	`  MailReader mailParser = new MailReader(`
	`  mailDOMCreator, rawFileInputReader);`
	`  MailInServer incomingServer =`
	`  new MailInServer(`
	`  EncryptionType.TLS, ServerType.IMAP);`
	`  SendMailServer outgoingServer =`
	`  new SendMailServer(EncryptionType.TLS);`
	`  ServerConfiguration serverData =`
	`  new ServerConfiguration(`
	`  incomingServer, outgoingServer);`
	`  LocalArchive mailLocation =`
	`  new LocalArchive();`
	`  Credentials loginInfo =`
	`  new Credentials(userName, password);`
	`  MailAccount result =`
	`  new MailAccount(mailParser,`
	`  serverData, mailLocation, loginInfo);`
	`  UserInfo userProfile = new UserInfo();`
	`  result.setUserProfile(userProfile);`
	
	`  return result;`
	`}`
		`def createPrototypeNetworkFunctionality() {`
		
		`  def pastEvents =`
		`  new EventHistory();`
		`  def incidentManager =`
		`  new NetworkEventHandler(pastEvents);`
		`  def transmissionMethod =`
		`  TransportProtocol.TCP;`
		`  def endPoint =`
		`  new IPAddress();`
		`  def serverFacade =`
		`  new ServerProxy(`
		`  transmissionMethod, endPoint);`
		`  def io =`
		`  new FileAccess();`
		`  def parser =`
		`  new GameLevelParser();`
		`  def formatter =`
		`  new Serializer(io, parser);`
		`  def result = new NetworkAccess(`
		`  incidentManager, serverFacade,`
		`  formatter);`
		`  def gameInfo =`
		`  new GameData(GameState.Idle);`
		`  result.setNextContent(`
		`  new GamePackage(gameInfo));`
		`  return result;`
		`}`

Table 12 Measured development times(time in seconds) – Start=Language subjects started with, G=Groovy, J=Java

		CIT1		CIT2		CIT3		SEFT1		SEFT2		CIT4		TEFT1		CIT5		TEFT2		Sums	
Subject	Start	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy
1	G	286	1212	432	463	663	1258	1141	3062	169	174	812	1397	197	813	584	1021	70	332	4354	9732
2	G	233	507	567	892	1163	828	554	2110	210	828	675	950	130	1009	612	780	106	1308	4250	9212
3	G	382	886	308	516	425	901	476	492	121	990	432	1288	206	280	565	1060	80	783	2995	7196
4	G	124	709	214	345	418	696	181	1117	136	358	536	685	100	263	385	862	67	926	2161	5961
5	G	146	343	215	578	414	1010	104	788	127	92	323	537	98	149	434	785	46	273	1907	4555
6	G	413	563	301	483	520	904	187	458	76	420	367	602	88	927	514	770	44	1571	2510	6698
7	G	522	568	293	418	579	893	489	747	86	170	473	663	172	739	671	797	210	1263	3495	6258
8	G	170	241	209	452	468	1371	583	543	139	360	576	1621	139	258	526	884	127	510	2937	6240
9	G	502	2215	313	748	470	799	1163	984	419	684	858	633	374	1123	374	1838	94	812	4567	9836
10	G	621	1169	1169	1273	846	1274	1015	1707	293	382	1039	1894	298	747	1064	1608	108	398	6453	10452
11	G	166	408	313	497	432	1008	281	1091	74	611	511	808	116	308	450	812	137	413	2480	5956
12	G	288	476	669	516	612	2146	2562	1537	637	471	526	1422	262	847	664	1156	127	1712	6347	10283
13	G	153	1232	196	562	535	611	220	407	467	197	344	575	129	1371	501	564	244	636	2789	6155
14	G	321	897	275	430	793	1720	604	841	181	2262	610	1266	240	1492	409	1211	430	750	3863	10869
15	G	411	494	286	742	381	880	350	801	673	127	443	698	169	278	543	844	116	1186	3372	6050
16	G	271	733	193	682	320	978	626	358	90	416	358	880	137	1337	293	832	53	277	2341	6493
17	G	459	1328	700	881	659	976	324	337	137	614	719	948	78	943	795	1248	97	246	3968	7521
18	J	483	252	315	322	649	1055	1205	153	219	107	791	803	187	772	645	1141	114	450	4608	5055
19	J	562	227	981	386	1038	2169	1930	222	268	774	1212	2505	175	1375	1016	1262	98	547	7280	9467
20	J	375	684	388	419	941	2433	584	197	240	261	1127	549	563	1558	566	1326	108	315	4892	7742
21	J	764	512	558	295	938	1093	1764	377	475	120	3240	1388	286	550	740	971	535	498	9300	5804
22	J	1139	2179	613	1265	574	1411	2256	417	337	167	582	1479	344	1202	735	1054	153	437	6733	9611
23	J	479	575	1178	439	1983	1054	2910	1042	1022	282	1197	818	900	665	782	758	244	2275	10695	7908

Table 12 (continued)

		CIT1		CIT2		CIT3		SEFT1		SEFT2		CIT4		TEFT1		CIT5		TEFT2		Sums	
Subject	Start	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy
24	J	1106	943	895	1042	893	1566	2166	809	927	177	1011	997	324	730	841	1279	103	1604	8266	9147
25	J	914	1620	1051	508	842	869	787	226	325	140	716	741	173	728	732	798	192	381	5732	6011
26	J	673	358	1590	677	711	591	1423	252	177	87	650	640	160	170	691	703	102	594	6177	4072
27	J	907	886	1713	739	773	1581	1642	330	1458	329	920	600	352	732	722	1391	144	761	8631	7349
28	J	1609	2124	4285	756	1697	932	2278	2648	1174	1045	1541	1036	200	2565	1514	1166	116	2285	14414	14557
29	J	1007	556	1883	1308	1757	1419	2000	686	880	742	1665	1034	268	1845	950	2104	146	1004	10556	10698
30	J	484	325	735	349	953	1475	1356	218	320	134	626	1851	210	1118	825	904	215	1371	5724	7745
31	J	560	563	858	669	1069	1513	312	719	2264	251	781	817	258	582	823	1189	147	958	7072	7261
32	J	651	856	637	1085	1401	723	1384	639	885	233	915	735	348	1565	1049	2538	105	752	7375	9126
33	J	480	339	1437	1354	899	883	1822	531	1740	142	720	1004	102	1585	772	1046	170	397	8142	7281

Table 13 Number of test runs – Start=Language subjects started with, G=Groovy, J=Java

		CIT1		CIT2		CIT3		SEFT1		SEFT2		CIT4		TEFT1		CIT5		TEFT2		Sums	
Subject	Start	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy
1	G	3	19	4	5	5	19	6	12	3	2	11	17	2	9	4	12	2	4	40	99
2	G	4	5	4	11	12	11	2	11	4	3	3	22	3	8	3	16	2	10	37	97
3	G	6	10	5	6	7	17	4	3	3	7	3	13	3	2	4	11	2	4	37	73
4	G	2	7	2	5	5	12	2	5	3	3	6	10	2	2	2	12	1	4	25	60
5	G	2	9	2	8	7	12	2	4	4	2	5	12	3	2	5	16	2	3	32	68
6	G	5	6	4	4	6	12	2	3	1	4	4	10	2	5	3	17	1	15	28	76
7	G	5	3	3	5	6	12	2	4	2	2	7	10	2	4	5	12	2	6	34	58
8	G	3	3	4	5	8	14	4	4	2	3	6	25	3	2	4	11	1	5	35	72
9	G	11	21	5	13	10	15	3	4	3	4	14	14	3	14	2	28	2	8	53	121
10	G	4	15	6	17	9	15	5	4	2	4	6	28	3	6	10	21	2	4	47	114
11	G	2	5	4	5	5	13	2	6	1	4	3	11	3	3	3	11	2	4	25	62
12	G	2	7	7	4	7	20	13	3	2	2	8	15	3	12	4	20	2	13	48	96
13	G	2	17	2	8	7	12	4	5	4	2	5	8	3	14	7	10	4	6	38	82
14	G	2	12	5	4	11	25	4	4	2	11	5	11	4	10	3	19	4	10	40	106
15	G	1	7	1	6	1	12	1	2	1	2	1	13	1	4	1	19	1	7	9	72
16	G	4	13	2	9	6	13	3	3	2	5	6	15	2	11	4	14	2	4	31	87
17	G	6	17	3	10	3	14	3	3	2	3	2	12	3	5	2	14	2	3	26	81
18	J	6	2	3	4	6	9	4	2	2	1	7	7	2	6	3	13	2	4	35	48
19	J	5	3	9	4	8	18	11	4	2	6	4	26	3	12	4	25	1	4	47	102
20	J	3	6	6	3	12	13	3	2	2	3	4	5	2	9	2	11	2	4	36	56
21	J	12	9	8	4	13	14	9	4	5	2	15	18	2	7	5	8	3	3	72	69
22	J	11	23	5	11	5	14	6	2	2	2	4	13	2	10	3	20	3	4	41	99
23	J	6	5	15	4	35	14	20	4	8	3	20	15	9	9	7	13	2	27	122	94

Table 13 (continued)

		CIT1		CIT2		CIT3		SEFT1		SEFT2		CIT4		TEFT1		CIT5		TEFT2		Sums	
Subject	Start	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy
24	J	6	9	15	8	12	11	11	2	6	2	9	16	3	7	7	29	2	9	71	93
25	J	11	9	6	8	5	12	2	2	2	2	5	4	3	4	2	4	2	4	38	49
26	J	6	4	12	3	5	12	5	3	3	1	3	12	2	3	4	12	3	5	43	55
27	J	9	8	8	6	8	19	6	2	5	5	4	11	2	5	2	18	2	6	46	80
28	J	13	21	24	7	17	10	5	8	5	5	7	12	2	23	7	19	1	5	81	110
29	J	5	4	7	13	11	11	4	4	4	3	9	6	2	17	3	23	1	7	46	88
30	J	3	3	5	5	5	9	5	2	2	2	5	22	2	5	5	15	1	6	33	69
31	J	7	9	9	4	16	23	6	6	11	2	5	10	2	4	3	21	3	12	62	91
32	J	7	5	3	10	9	6	4	5	2	3	5	4	2	16	4	25	1	4	37	78
33	J	4	4	7	12	8	12	10	5	8	2	4	13	2	15	3	19	1	3	47	85

Table 14 Number of viewed files – Start=Language subjects started with, G=Groovy, J=Java

		CIT1		CIT2		CIT3		SEFT1		SEFT2		CIT4		TEFT1		CIT5		TEFT2		Sums	
Subject	Start	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy
1	G	4	29	8	9	17	18	11	19	3	3	12	20	3	8	8	16	3	6	69	128
2	G	4	5	8	9	16	10	7	12	4	5	13	14	2	12	13	13	4	8	71	88
3	G	3	13	5	17	8	16	10	11	4	14	10	17	4	2	12	16	3	10	59	116
4	G	4	5	7	6	8	10	5	8	4	9	13	15	2	6	13	13	3	10	59	82
5	G	4	5	9	10	8	12	4	13	3	3	13	14	2	4	14	13	2	6	59	80
6	G	6	8	9	8	10	13	7	8	4	4	11	12	2	4	13	13	2	8	64	78
7	G	4	8	7	7	9	14	7	7	3	3	12	12	4	6	13	13	5	12	64	82
8	G	4	6	8	10	14	45	11	8	3	4	13	16	2	5	13	14	4	11	72	119
9	G	8	14	8	8	11	10	11	10	5	3	14	12	4	7	13	12	3	9	77	85
10	G	8	10	12	10	13	11	9	16	3	5	15	17	5	7	16	13	3	3	84	92
11	G	4	5	8	8	10	10	7	10	3	4	13	15	2	5	13	13	3	6	63	76
12	G	4	4	10	7	13	23	12	11	8	5	11	31	4	5	13	16	3	11	78	113
13	G	4	9	7	8	12	8	5	7	10	3	11	14	4	5	12	13	3	9	68	76
14	G	4	4	7	7	11	17	12	10	4	23	11	15	4	10	13	16	4	6	70	108
15	G	4	6	8	13	7	8	4	7	3	2	9	19	2	4	4	11	2	9	43	79
16	G	3	5	8	15	6	17	9	8	2	4	11	16	5	7	12	12	2	5	58	89
17	G	3	13	10	8	10	10	6	4	3	9	13	14	2	4	11	13	3	4	61	79
18	J	5	4	8	7	12	18	11	4	3	3	12	17	2	10	13	14	4	7	70	84
19	J	3	3	8	7	10	9	8	3	3	5	13	11	2	7	9	9	3	5	59	59
20	J	4	10	7	10	14	25	6	5	4	4	19	13	6	10	13	19	2	4	75	100
21	J	9	4	8	7	12	19	13	6	4	3	15	16	4	6	13	13	5	6	83	80
22	J	7	13	7	11	10	17	11	5	4	3	11	18	3	8	13	11	3	6	69	92
23	J	5	11	8	10	31	15	14	13	12	5	14	15	9	6	13	13	4	16	110	104

Table 14 (continued)

		CIT1		CIT2		CIT3		SEFT1		SEFT2		CIT4		TEFT1		CIT5		TEFT2		Sums	
Subject	Start	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy
24	J	6	6	9	14	10	19	14	8	9	3	13	11	4	14	13	12	3	12	81	99
25	J	3	23	11	11	15	15	10	6	6	4	13	14	4	11	13	14	3	9	78	107
26	J	7	4	8	11	10	6	12	6	2	3	13	13	3	4	13	13	3	5	71	65
27	J	5	18	22	19	12	17	17	13	12	2	20	14	4	7	13	13	3	13	108	116
28	J	6	5	10	6	11	9	15	13	7	5	13	12	2	6	13	13	3	12	80	81
29	J	5	6	25	11	15	18	12	7	9	3	17	16	4	10	13	35	3	12	103	118
30	J	5	6	8	8	11	16	11	4	6	3	12	15	5	9	13	13	3	11	74	85
31	J	4	4	9	13	14	11	2	5	6	3	13	12	5	6	13	15	3	7	69	76
32	J	4	9	8	8	30	12	10	8	6	4	13	14	4	8	13	15	3	10	91	88
33	J	5	5	14	11	15	11	13	11	10	4	12	13	2	7	13	13	4	7	88	82

Table 15 Number of file switches – Start=Language subjects started with, G=Groovy, J=Java

		CIT1		CIT2		CIT3		SEFT1		SEFT2		CIT4		TEFT1		CIT5		TEFT2		Sums	
Subject	Start	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy
1	G	9	37	52	21	59	53	19	56	31	67	58	59	57	5	6	41	46	63	337	402
2	G	16	17	85	6	57	28	2	20	34	41	36	84	33	60	4	4	18	4	285	264
3	G	10	39	72	49	126	62	12	58	9	67	48	58	96	65	8	91	5	6	386	495
4	G	12	54	87	22	81	147	15	72	88	96	38	94	226	108	10	28	56	49	613	670
5	G	7	28	78	79	79	31	21	54	44	190	61	58	230	106	4	35	46	42	570	623
6	G	6	44	224	32	88	30	11	23	29	64	64	82	99	116	50	38	24	19	595	448
7	G	21	10	57	42	79	24	2	19	22	46	48	45	125	10	4	49	65	7	423	252
8	G	5	43	57	26	61	68	6	51	5	72	77	39	54	76	6	11	6	2	277	388
9	G	45	26	46	66	40	34	2	28	40	63	41	44	50	16	4	12	12	3	280	292
10	G	5	28	72	25	157	42	13	27	48	71	61	102	101	99	8	9	91	17	556	420
11	G	15	14	54	29	169	24	3	41	14	64	78	277	110	54	14	51	32	10	489	564
12	G	2	21	27	22	38	14	4	34	7	27	11	35	42	7	3	21	27	18	161	199
13	G	8	17	67	161	113	49	9	37	62	76	76	75	71	49	8	14	20	14	434	492
14	G	113	9	77	37	95	18	8	22	73	67	41	110	97	48	25	38	33	8	562	357
15	G	11	9	43	80	47	18	8	40	86	56	45	36	70	20	13	30	42	46	365	335
16	G	5	12	44	17	57	31	2	45	8	38	54	55	42	7	2	86	17	5	231	296
17	G	6	30	57	18	76	96	2	60	56	72	50	65	59	88	11	44	26	78	343	551
18	J	15	10	63	21	77	42	2	65	73	116	49	60	49	47	10	129	89	16	427	506
19	J	8	69	51	81	141	46	12	82	59	46	56	98	84	115	8	27	21	12	440	576
20	J	16	30	79	46	63	85	41	30	25	192	48	72	49	191	15	66	161	75	497	787
21	J	18	46	80	93	56	27	18	43	34	46	36	36	98	45	6	39	22	10	368	385
22	J	8	27	62	42	34	37	9	47	29	49	36	80	42	115	10	27	38	38	268	462
23	J	5	19	45	10	63	25	4	17	28	54	46	97	109	70	9	6	24	4	333	302

Table 15 (continued)

		CIT1		CIT2		CIT3		SEFT1		SEFT2		CIT4		TEFT1		CIT5		TEFT2		Sums	
Subject	Start	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy	Java	Groovy
24	J	4	21	55	68	151	33	5	35	55	55	22	82	60	66	4	133	8	7	364	500
25	J	7	21	32	39	59	19	4	22	21	34	43	60	47	32	7	40	64	5	284	272
26	J	18	18	44	25	223	44	13	27	16	60	39	183	110	193	14	120	90	31	567	701
27	J	16	25	51	37	54	25	2	48	10	54	66	55	55	18	2	49	54	7	310	318
28	J	20	8	49	25	56	19	2	27	12	37	40	57	68	13	5	73	17	5	269	264
29	J	20	17	52	32	84	24	16	64	46	32	32	70	56	57	2	23	9	2	317	321
30	J	2	38	70	44	39	79	18	50	18	57	34	67	61	91	4	21	35	97	281	544
31	J	6	13	85	38	38	15	14	29	61	69	48	137	126	30	3	14	5	8	386	353
32	J	83	20	39	45	111	18	12	59	10	39	53	55	87	34	3	31	83	6	481	307
33	J	29	63	102	41	43	105	2	19	82	92	71	36	41	138	4	90	86	30	460	614

References

Bruce KB (2002) Foundations of object-oriented languages: types and semantics. MIT Press, Cambridge

Bird R, Wadler P (1988) An introduction to functional programming. Prentice Hall International (UK) Ltd., Hertfordshire

Cardelli L (1997) Type systems. In: Tucker AB (ed) The computer science and engineering handbook, chap 103. CRC Press, Boca Raton, pp 2208–2236

Callaú O, Robbes R, Tanter É, Röthlisberger D (2013) How (and Why) developers use the dynamic features of programming languages: the case of small talk. Empir Softw Eng 18(6):1156–1194

Curtis B (1988) Five paradigms in the psychology of programming. In: Helander M (ed) Handbook of human-computer interaction. Elsevier, North-Holland, pp 87–106

Daly MT, Sazawal V, Foster JS (2009) Work in progress: an empirical study of static typing in ruby. Workshop on evaluation and usability of programming languages and tools (PLATEAU). Orlando, Florida, October 2009

Denny P, Luxton-Reilly A, Tempero E (2012) All syntax errors are not equal. In: Proceedings of the 17th ACM annual conference on innovation and technology in computer science education, ITiCSE '12. ACM, New York, pp 75–80

Endrikat S, Hanenberg S (2011) Is aspect-oriented programming a rewarding investment into future code changes? A socio-technical study on development and maintenance time. In: The 19th IEEE international conference on program comprehension, ICPC 2011, Kingston, ON, Canada, June 22–24, 2011, pp 51–60

Feigenspan J, Kästner C, Liebig J, Apel S, Hanenberg S (2012) Measuring programming experience. In: IEEE 20th international conference on program comprehension, ICPC 2012, Passau, Germany, June 11–13, 2012. ICPC'12, pp 73–82

Gannon JD (1977) An experimental evaluation of data type conventions. Commun ACM 20(8):584–595

Gat E (2000) Point of view: LISP as an alternative to Java. Intelligence 11(4):21–24

Gravetter FJ, Wallnau LB (2009) Statistics for the behavioral sciences. Wadsworth Cengage Learning

Hanenberg S (2010) An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. In: Proceedings of the ACM international conference on object oriented programming systems languages and applications, OOPSLA '10. ACM, New York, pp 22–35

Hanenberg S (2011) A chronological experience report from an initial experiment series on static type systems. In: 2nd workshop on empirical evaluation of software composition techniques (ESCOT). Lancaster

Hudak P, Jones MP (1994) Haskell vs. ada vs. c++ vs. awk vs.... an experiment in software prototyping productivity. Technical report

Hanenberg S, Kleinschmager S, Josupeit-Walter M (2009) Does aspect-oriented programming increase the development speed for crosscutting code? An empirical study. In: Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement, ESEM '09, Lake Buena Vista. IEEE Computer Society, Florida, pp 156–167

Höst M, Regnell B, Wohlin C (2000) Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. Empir Softw Eng 5(3):201–214

Juristo N, Moreno AM (2001) Basics of software engineering experimentation. Springer

Juzgado NJ, Vegas S (2011) The role of non-exact replications in software engineering experiments. Empir Softw Eng 16(3):295–324

Kitchenham B, Al-Khilidar H, Ali Babar M, Berry M, Cox K, Keung J, Kurniawati F, Staples M, Zhang H, Zhu L (2006) Evaluating guidelines for empirical software engineering studies. In: ISESE '06: proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering. ACM, New York, pp 38–47

Kleinschmager S, Hanenberg S, Robbes R, Tanter É, Stefik A (2012) Do static type systems improve the maintainability of software systems? An empirical study. In: IEEE 20th international conference on program comprehension, ICPC 2012, Passau, Germany, June 11–13, 2012, pp 153–162

Kleinschmager S (2011) An empirical study using Java and Groovy about the impact of static type systems on developer performance when using and adapting software systems. Master thesis at the institute for computer science and business information systems, University of Duisburg-Essen

Ko AJ, Myers BA, Coblenz MJ, Aung HH (2006) An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans Softw Eng 32(12):971–987

McConnell S (2010) 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, O'Reilly series. O'Reilly Media, pp 567–575

Mayer C, Hanenberg S, Robbes R, Tanter É, Stefik A (2012) An empirical study of the influence of static type systems on the usability of undocumented software. In: ACM SIGPLAN conference on object-oriented programming systems and applications, OOPSLA '12

Nierstrasz O, Bergel A, Denker M, Ducasse S, Gälli M, Wuyts R (2005) On the revival of dynamic languages. In: Proceedings of the 4th international conference on software composition, SC'05. Springer-Verlag, Berlin, Heidelberg, pp 1–13

Pfleeger SL (1995) Experimental design and analysis in software engineering. Ann Softw Eng 1:219–253

Pierce BC (2002) Types and programming languages. MIT Press, Cambridge

Prechelt L (2000) An empirical comparison of seven programming languages, IEEE computer (33). Computer 33:23–29

Prechelt L (2001) Kontrollierte experimente in der softwaretechnik. Springer, Berlin

Prechelt L, Tichy WF (1998) A controlled experiment to assess the benefits of procedure argument type checking. IEEE Trans Softw Eng 24(4):302–312

Richards G, Hammer C, Burg B, Vitek J (2011) The eval that men do - a large-scale study of the use of eval in javascript applications. In: ECOOP 2011 - object-oriented programming - 25th European conference, Lancaster, UK, July 25–29, 2011 Proceedings, pp 52–78

Rosenthal R, Rosnow R (2008) Essentials of behavioral research: methods and data analysis. McGraw-Hill higher education. McGraw-Hill Companies, Incorporated

Steinberg M, Hanenberg S (2012) What is the impact of static type systems on debugging type errors and semantic errors? An empirical study of differences in debugging time using statically and dynamically typed languages - unpublished work in progress

Stuchlik A, Hanenberg S (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 2011, October 24, 2011, Portland, OR, USA. ACM, pp 97–106

Sjøberg DIK, Hannay JE, Hansen O, Kampenes VB, Karahasanović A, Liborg N-L, Rekdal AC (2005) A survey of controlled experiments in software engineering. IEEE Trans Softw Eng 31(9):733–753

Tichy WF (2000) Hints for reviewing empirical work in software engineering. Empir Softw Eng 5(4):309–312

Tratt L (2009) Dynamically typed languages. Adv Comput 77:149–184

van Deursen A, Moonen L (2006) Documenting software systems using types. Sci Comput Program 60(2):205–220

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in software engineering: an introduction. Kluwer, Norwell

Stefan Hanenberg works as a researcher at the University of Duisburg-Essen, Germany. From the same university he received his PhD in 2006. His research interests are mainly in the empirical evaluation of programming language constructs in general and in the area of evaluation of type systems in particular.

Sebastian Kleinschmager is a software engineer from Germany who has a special interest in creating a scientific foundation for his field. During his studies of applied computer science (BSc) and business information systems (MSc) at the University of Duisburg-Essen, he focused his research on conducting empirical experiments to evaluate programming techniques. After his studies he pursued a career in business and currently works as a Senior IT Specialist, specializing in business application and web software development.

Romain Robbes is assistant professor at the University of Chile (Computer Science Department), in the PLEIAD research lab, since January 2010. He earned his PhD in 2008 from the University of Lugano, Switzerland and received his Masters degree from the University of Caen, France. His research interests lie in Empirical Software Engineering and Mining Software Repositories. He authored more than 50 papers on these topics at top software engineering venues (ICSE, FSE, ASE, EMSE, ECOOP, OOPSLA), and received best paper awards at WCRE 2009 and MSR 2011. He was program co-chair of IWPSE-EVOL 2011, IWPSE 2013, and WCRE 2013, is involved in the organisation of ICSE 2014, and the recipient of a Microsoft SEIF award 2011.

Éric Tanter is an Associate Professor in the Computer Science Department of the University of Chile, where he co-leads the PLEIAD laboratory. He received the PhD degree in computer science from both the University of Nantes and the University of Chile (2004). His research interests include programming languages and tool support for modular and adaptable software. He is a member of the ACM, the IEEE, and the SCCC.

Andreas Stefik is an assistant professor at the University of Nevada, Las Vegas. He completed his Ph.D. in computer science at Washington State University in 2008 and also holds a bachelor's degree in music. Andreas's research focuses on computer programming languages and development environments, including their use by sighted, blind, and visually impaired people. He won the 2011 Java Innovation Award for his work on the NSF funded Sodbeans programming environment and has developed a computer programming language, Quorum, designed in randomized controlled trials with human users to evaluate and improve its ease of use.