

An Empirical Comparison of Static and Dynamic Type Systems on API Usage in the Presence of an IDE: Java vs. Groovy with Eclipse

Pujan Petersen
University of Duisburg-Essen
Institute for Computer Science
and Business Information
Systems, Essen, Germany
pujan.petersen@stud.uni-
due.de

Stefan Hanenberg
University of Duisburg-Essen
Institute for Computer Science
and Business Information
Systems, Essen, Germany
stefan.hanenberg@icb.uni-
due.de

Romain Robbes
University of Chile
Computer Science
Department (DCC)
Santiago de Chile, Chile
robbes@dcc.uchile.cl

ABSTRACT

Several studies have concluded that static type systems offer an advantage over dynamic type systems for programming tasks involving the discovery of a new API. However, these studies did not take into account modern IDE features; the advanced navigation and code completion techniques available in modern IDEs could drastically alter their conclusions. This study describes an experiment that compares the usage of an unknown API using Java and Groovy using the IDE Eclipse. It turns out, that the previous findings that static type systems improve the usability of an unknown API still hold, even in the presence of a modern IDE.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features

General Terms

Human Factors, Languages

Keywords

programming languages, type systems, empirical research

1. INTRODUCTION

The relative merits of static and dynamic type systems (see [21, 2]) are still subject to discussions, both in academia and in industry. Since languages with dynamic type systems are used in industry—especially in the area of web-programming, where dynamic languages such as JavaScript, PHP or Ruby are being used—, this discussion is very relevant to the practice. The argument that a static type system increases the development performance of developers

has been made many times. For instance, TypeScript has been proposed as an alternative to Javascript partly for those reasons¹.

This phenomenon is also studied in academia. A number of experiments [10, 11, 28, 27, 14, 16, 13] indicate that the application of statically typed languages has indeed a measureable benefit for software developers—especially in those situations, where a previously unknown API has to be used in order to conclude a given programming task (see [14]). However, controlled experiments have to fix some variables in order to measure the effect of other variables. Some of these choices may reduce the generalizability of the results. From a practical perspective, the previous studies have some weaknesses:

- the studies did not take the existence of IDEs into account,
- the studies did not document the code in a Java-like fashion (using JavaDocs), and
- the studies did use to a certain degree artificial parameter names, which made it possibly harder to use the dynamically typed API.

Each of these criticisms is a valid point in order to argue against the generalizability of these experiments. In particular, the absence of IDEs is a serious threat to the validity of previous experiments. IDEs offer many tools that boost programmer productivity, which possibly counterbalance the effects observed in previous experiments. In particular, in the presence of code completion, the IDE gives already a number of hints on how a class of an API could be used. Consequently, in the presence of code completion, there is less of a need to manually explore the source code in order to search for methods that are appropriate in a given situation. Indeed, the study of Murphy et al. found that code completion was one of the most frequently used features of the Eclipse IDE [17].

In order to verify the findings of these previous experiments, each of the mentioned critiques could be checked individually with an additional controlled experiment. However, running controlled experiments is a costly endeavour. Given that the measured differences between static and dynamic languages on API usage were relatively large, it seems straight forward to design a single experiment that considers all mentioned points together in order to

¹See <http://blogs.msdn.com/b/somasegar/archive/2012/10/01/typescript-javascript-development-at-application-scale.aspx>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICPC '14, June 2-3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06 ...\$15.00.

check, whether in the presence of an appropriate IDE, corresponding JavaDocs, and non-artificial parameter names there is still a measurable difference between the API usage of statically and dynamically typed languages.

This paper reports on an experiment that is closely related to the experiment by Kleinschmager et al. [14] and which compares the development times of Java and Groovy using the Eclipse IDE. The tasks are programming tasks where a previously unknown API has to be applied. Classes and methods of this API were documented using JavaDocs, and the parameter names were chosen in a way, that seemed most appropriate for the given situation: type names as part of parameter names were chosen whenever it seemed appropriate.

The results of the experiment are that even under these conditions, a measurable positive effect of the statically typed programming language Java is observed. Further, the differences between Java/Eclipse and Groovy/Eclipse are (still) quite large.

The paper is structured as follows. In section 2 we give an overview of existing related work. Section 3 describes the design of the experiment. The results of the experiment are then shown and discussed (section 4). Then, we perform an exploratory study in order to test, what effects might be potentially responsible for the measured differences. After discussing the paper in section 6, section 7 summarizes and concludes the paper.

2. RELATED WORK

There are three different kinds of work that we consider mainly related to the here described experiment. First, works about IDEs and code completion, works about API usability, and works about the usability of type systems. Each of these kinds of work is described below in a separate section.

2.1 IDEs and Code Completion

There has been studies on the effect of integrated development environments, and of code completion in particular, on developer productivity.

Most recently, Zayour and Hajjdiab performed a qualitative study of the usage of Visual Studio in a company [29]. In it they found developer productivity improvements, which were however tempered by additional complexities in libraries and components, leading to mixed results.

Alternative IDEs have been shown to be improvements over state of the practice IDEs. A study comparing Code Bubbles with Eclipse [1] showed that programming tasks done with Code Bubbles were done in less time and were more correct than when they were done with Eclipse. Similarly, Code Canvas [7], Debugger Canvas [6], and Gaucho [19] showed varying degree of improvement over their respective baselines.

There has been several studies focused on the code completion component of IDEs since Murphy's study that showed that code completion was comprehensively used by developers [17]. Robbes and Lanza [23] showed that history-aware code completion yielded both quantitative and qualitative improvements, while Omar et al. [20] and Nguyen et al. [18] showed that contextual and structural improvements to code completion were well-perceived and useful to developers.

2.2 API usage

Several studies have shown that API documentation can be both a blessing and a curse to software developers. A qualitative study by Robillard and DeLine highlighted that the type of documentation that was perceived as most useful contains common use cases of the API and examples using several API elements [24]. On

the other hand, the study by Roehm et al. showed that most industrial developers in the observed sample preferred to query the source code if available, as the documentation risked being out of data [25]. Additionally the study by Maalej and Robillard of .Net and the JDK showed that nearly half of the documentation of these frameworks was not useful [15].

Nevertheless, a controlled experiment by Dekel and Herbsleb [4], and a subsequent qualitative study of the same data [5] showed that highlighting relevant information in an API yielded an increase in the correctness of tasks. For a more thorough survey on the related work in the area of API usage, we refer to our previous paper [8].

2.3 Usability of static and dynamic type systems

The here described related work for the usability of type systems corresponds to the descriptions that were done in previous papers by some of the authors (see for example [14, 16]). The only differences are experiments that have been performed more recently.

Studies by other authors: The first experiment we are aware of about the usability of type systems was performed by Gannon [9]. It revealed a positive influence of a static type system in comparison to an untyped system: An increase in programming reliability for subjects using the language with static type checking was detected.

Approximately 20 year later, Prechelt and Tichy run a study about the impact of static type checking on procedure arguments (see [22]). The study used ANSI C and K&R C as programming languages. The result of the study was, that for one task subjects using the statically type checked ANSI C were faster in solving the programming task.

In a qualitative pilot-study on a static type system for the programming language Ruby, the authors Daly et al. concluded that the benefits of static typing could not be shown [3].

Studies by this paper's authors: The study presented here is part of a larger experiment series that started about 10 year after the study by Prechelt and Tichy. The series studies the influence of static and type systems on software development time. So far, eight experiments have been executed:

1. The first study of this series (see [11]) revealed a significant positive time benefit for a smaller task for dynamically typed languages, while no significant difference between dynamically and statically typed languages on development time could be measured for a larger task.
2. Stuchlik and Hanenberg analyzed the influence of type casts on development time for simple programming tasks [28]. The results of the study were that type casts influence the development time of rather completely trivial programming tasks in a negative way, while already for code longer than eleven LOC no significant difference was measured.
3. An experiment by Steinberg and Hanenberg analyzed to what extent static type systems help to identify and fix type errors as well as semantic errors in an application [27]. The result of the experiment was that static type systems have a positive impact on the time required to fix type errors, but no differences with respect to fixing semantic errors were measured.
4. Mayer et al. studied the influence of static types to help using undocumented APIs [16]. For most of the programming tasks static types were beneficial.
5. Kleinschmager et al. performed a repetition of previous experiments [14, 12]. Among others, the key findings from

[16]—that static type systems help using undocumented APIs—were confirmed; this time without any exception.

6. Hoppe and Hanenberg showed in an empirical study that the additional type information given in generic types help using unknown APIs compared to raw types in Java [13].
7. Spiza and Hanenberg studied whether the finding from previous experiments could be reduced to the question, whether the pure syntactical representation of type declarations (without a static type check) already helps developers when using unknown APIs [26]. The result was, that they help indeed – but when one of these unchecked declarations uses a wrong type name, it has a measurable negative effect on development time.
8. Endrikat et al. [8] showed in an experiment, that the effect of type systems in the presence of documentation (available as external documents with code examples) is still measurable—and even larger than the effect of documentation.

From the experiment series it can be concluded (among other statements), that there is quite strong empirical evidence for the usability of static type systems when unknown APIs have to be applied (from the experiments [16, 14, 13, 26, 8]). However, none of these experiments used a state of the practice IDE in the experimental setting—a fact that is mentioned in each paper’s threats to validity section.

3. EXPERIMENT DESCRIPTION

This experiment can be considered as a partial replication of the experiment by Kleinschmager et al. [14]. Because of this, we start this section with a more detailed description of the previous experiment. Then, we describe our adaptations to the experimental design. Then, we describe threats to validity.

3.1 Background: Initial Experiment by Kleinschmager et al. [14, 12]

The starting point for this experiment was the experiment performed by Kleinschmager et al. [14, 12] which itself was based on previous experiments ([27, 16]): while the experiment focused on three different possible characteristics of type systems (time for fixing type errors, time for fixing semantic errors, time for using unknown APIs), the main outcome of the experiment was, that the benefits of static type systems for using unknown APIs was shown without exception.

The experiment was performed as a within–subject design where each participant solved the programming task twice. Thereto, each participant first solved all programming tasks in one language, then he switched to the different language and again solved all programming tasks. The tasks were solved using a statically typed programming language (which was Java) and with a dynamically typed language (Groovy). The dynamically typed language Groovy was used as a dynamically typed Java by ignoring the additional language features Groovy provides in addition to Java and by declaring each variable and parameter as `def` – i.e. without giving them a named type.

3.1.1 Constraints

The experiment set a number of constraints in its design in order to study only the effect of type systems:

Programming environment: As a programming environment, the participants were given a text editor with an additional tree view, which contained all files required for the programming tasks.

```
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;
}
```

Figure 1: Possible solution code for a class identification class (taken from [12])

That the intention of the experiment was to study the effect of the type system—not the possible different maturity levels of IDEs that could be found for statically or dynamically typed languages.

Variable and Parameter Names: Furthermore, all variable and parameter names were given artificial names: while these names were still meaningful in the programming context, they were defined in a way that developers could not directly conclude from the parameter names, what the names of the classes were, that could be passed to the API. The intention for this design choice was, that we intended to measure the possible documentation effect of type systems—without disturbing this possible effect by giving the same information in parameter or variable names.

Documentation: Finally, it was decided that no documentation was given to the subjects about the API (except the API source code). The motivation for this was, that we assumed that the effect to be measured would be stronger without documentation.

3.1.2 Programming tasks

The three different characteristics of type systems were covered by nine different programming tasks: five tasks to test whether the declared types in the (unknown) API help using the API; two tasks to check whether type errors can be fixed faster with a statically typed language; and two tasks to check whether semantic error can be fixed faster with a statically typed language.

j

All programming tasks had in common, that they did not require the participant to apply conditionals or loops, because it was assumed that each of these constructs potentially increases the deviation in the measurements – which would possibly make it harder to measure the effect that the experiment was intended to measure. From the perspective of LOC the programming tasks could be considered as trivial – up to 15 LOC were required to solve the programming tasks.

Figure 1 illustrates one possible solution code for one of those tasks, which tested the documentation characteristics of static type systems (the given example is the solution for Groovy).

3.1.3 Results

The results of the study were, that a positive impact of the statically typed language could be measured for API usability as well as for type error fixing time. No difference could be measured for fixing semantic errors.

The primary measurement for the experiment was development time – the time between delivering the programming task until a

valid solution was provided. In an additional exploratory study, the number of file switches, i.e. how often subjects switched from one file to another one turned out to reveal comparable results pro static type system as the development time. One interpretation of this measurement was, that such file switches indicate how largely developers explore the code.

3.2 Changes for the replication

The general idea was to replicate the experiment to a certain extent, but trying at the same time to add a number of changes to the experiment in order to get additional insights.

The obvious changes for a repetition were to remove the given constraints from the experiment, because each of these constraints represent some threat to validity (especially it's external validity).

Hence, the intention was to give participants in the experiment an IDE, to give them some documentation and to remove the constraint, that parameter and variable names must not directly refer to type names (especially for the dynamically typed solution).

While changes to the parameter and the variable names are relatively straight forward (just by giving each parameter a corresponding name where we think that such name is more appropriate), this is not the case for the programming environment and documentation.

Programming environment: With respect to the programming environment, it is necessary to find an appropriate IDE for the experiment. One alternative could be to use just one IDE for the Java group and a different one for the Groovy group – with the potential problem, that both are not directly comparable. We took two IDEs into account. The first one was IntelliJ² which provides support for Java as well as for Groovy. The other alternative we took into account was Eclipse³ for which also a Groovy plug-in exists⁴. Our main concern here was the Groovy IDE – although IDE support is available for Groovy, it turns out that in both cases, the IDE is not that strong, as it should be. Especially for code completion, the Groovy IDEs are relatively weak, because the underlying IDEs hardly infer types in order to approximate for the purpose of code completion the at least possible methods that could be send to an object. However, it turns out that this weakness exists on both IDEs taken into account. Additionally, just by reading comments from developers in a number of forums, we considered both IDEs valid for the experiment, because it looks like both are being actually used by Groovy developers. For the purpose of our experiment, we then decided to use Eclipse for Java and with the additional plug-in for Groovy. The motivation for this was, that we were aware that probably most of our participants in the experiment would come from the University of Duisburg–Essen, and most of them are already trained using Eclipse.

Programming tasks: With respect to the programming tasks, our intention was to concentrate on those tasks, where we expected the largest effect from the IDE: those tasks where the API had to be used and where we expected code completion (which we consider as one of the main features as IDEs) play a major role – because developers do not have to navigate to a classes source code in order to detect, which methods are provided by this class. We used the largest development task from the previous experiment (the tasks, whose code is illustrated in Figure 1). Additionally, we create a new comparable programming tasks with a comparable number of parameters, etc. Again, we provided this additional task in a statically typed as well as in a dynamically typed way. I.e. in contrast

²<http://www.jetbrains.com/idea/>

³<http://www.eclipse.org/>

⁴<http://groovy.codehaus.org/Eclipse+Plugin>

to the previous experiment, we just used two programming tasks, here. Both focused on the (possible) documentation characteristic of static type systems. The original programming task is the one that we will refer to as task 1 in the rest of the paper, the new task as task 2.

Documentation: While the previous experiment did not provide any documentation to the developers, our goal was to provide a kind of documentation, developers using the experiment's programming languages are typically familiar with. In the Java world, it is common to document classes and methods using JavaDocs, a technique to add documentation to the source code and which the IDEs in use are able to handle by highlighting the documentation in the code completion window.

	Round 1	Round 2
Group 1	Java/Task 1, Groovy/Task2	Groovy/Task1, Java/Task2
Group 2	Groovy/Task 1, Java/Task2	Java/Task 1, Groovy/Task2

Table 1: General experimental design

General experiment design: The previous experiment was a two-group experiment where one group started with one language and solved all programming tasks with it, then switched the language and solved the same programming tasks with the different language. A potential problem of this approach is, that there is probably a larger carry-over effect when all tasks are first being done using the first language before switching to the other language. In order to reduce this possible carry-over effect, we decided to make one change to the experimental design: instead of letting the participant solve both tasks with the same language first, we gave him the first task with the first language, then let him solve the second task with the second language. Then, the first task is solved by the second language, and finally the second task is solved with the first language – we did not let participants switch the language after the second task in order to no confuse them. Figure 1 illustrates the resulting 2-group within-subject experiment.

3.3 Threats to validity

The original experiment description already reported on a large number of threats to validity [12, 14], that we do not repeat here (because they have actually not changed). Instead we only discuss the possible threats that are caused by our changes to the experiment.

IDE maturity (external threat): The IDE being used in the experiment represents an IDE that is currently used in industry. However, taking into account that the programming language Java is available since approximately 20 year (while Groovy is approximately only 10 years old) and that it is highly probable that many more developers use Java instead of Groovy, it is also obvious that much more effort was spent on IDE support for Java than for Groovy. Consequently, nowadays Groovy IDEs have the tendency to be rather weak compared to the Java support available in the same or a comparable IDE. Hence, it is possible that the possibly measured difference between static and dynamic languages is only caused by the difference in the maturity of the IDEs.

Dynamic IDE functionality (external threat): Another closely related threat is the functionality provided by current IDEs for dynamic programming languages such as Groovy. Although in principle it would be possible that the IDE does a lot of code inferencing in order to improve code completion and other features of the IDE, it turns out that this is – at least currently – hardly the case. Even nowadays, IDEs for dynamic languages have troubles to in-

fer for example the return type of a method call – even though the corresponding method just consists of a single statement such as `return new A()`, which obviously returns type A. Hence, it is possible and maybe even likely that in the future the functionality of dynamic IDEs will significantly improve – which in that case is not covered by the here introduced experiment.

Documentation (external threat): While we introduced with JavaDocs one kind of possible documentation, it is unclear what the “right” way of documenting software is, i.e. it is unclear whether our choice really increases the external validity of the experiment, or whether it just introduces a different external threat. However, our argument here is that at least JavaDocs is currently an applied technique – independent of how large of a benefit is provided by it.

Experiment design (internal threat): We intentionally changed the design in the experiment in order to reduce the potential larger carry-over effect of the first experiment. However, it is possible, that this goal was actually not achieved by the change in the experiment design. And more important, in case the here described experiment has a different result than the previous experiment, this could be just an implication of the change in the experimental design.

4. MEASUREMENTS AND ANALYSIS

The experiment was executed at the University of Duisburg-Essen, Germany. The participants were bachelor students as well as master students. The students were not selected based on their educational background, but simply because of their availability (but making sure upfront, that the students already passed programming lectures). Altogether, 23 subjects participated in the experiment.

Because of its similarity and comparability to the previous experiment by Kleinschmager et al. (see [14]), we illustrate the measurements and the results in a comparable way: We start with a description of the raw measurements and corresponding graphical illustrations. Afterwards, we run for each round (i.e. when the subjects processed the tasks for the first time, respectively for the second time) as well as for the individual tasks corresponding significance tests.

4.1 Measurements and Descriptive Statistics

Table 2 contains all primary measurements from the experiment, namely the development times. In addition to the raw data, the differences between Java and Groovy for each task, as well as the complete development times, i.e. programming task 1 and 2 altogether, are shown—assuming that both programming tasks represent just one atomic task.

Just by watching the raw data it is possible to make a number of interesting statements.

- First, not a single subject required for the first programming task more time using Java/Eclipse than using Groovy/Eclipse - and the differences are between 1.5 minutes (92 seconds) and 50 minutes (3033 seconds)
- Second, only two of 23 subjects required for the second task more time using Java/Eclipse than Groovy/Eclipse. In both cases, the subjects began task 2 with Java, i.e. it seems plausible that this is the result of some learning effects for these subjects.
- Third, if the development times are summed up, each required more time using Groovy/Eclipse compared to Java/Eclipse for both tasks. The differences between these sums

Table 2: Measured development times (time in seconds) and descriptive statistics. Column Start describes the technique each subject (row) started with for the first task (J = Java/Eclipse, G = Groovy/Eclipse) – the second task started with the other technique

Subject	Start	Development time (in seconds)								
		Task1			Task2			Sum		
		Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff
1	J	952	1045	-92	670	1810	-1140	1623	2855	-1232
2	J	1148	1447	-299	493	1236	-743	1642	2683	-1041
3	J	1075	2708	-1633	590	1274	-685	1665	3983	-2318
4	J	1076	2128	-1052	502	1102	-600	1578	3230	-1652
5	J	1422	1677	-255	440	1723	-1284	1861	3400	-1539
6	J	1101	2164	-1063	733	2471	-1738	1834	4635	-2801
7	J	1003	1321	-318	656	1562	-906	1659	2883	-1224
8	J	814	2075	-1261	900	1406	-506	1714	3480	-1766
9	J	1264	1629	-365	680	1590	-910	1944	3219	-1275
10	J	1346	1651	-305	655	1625	-970	2001	3276	-1275
11	J	1475	1982	-507	523	1723	-1200	1998	3705	-1706
12	J	2115	3730	-1615	560	2067	-1508	2674	5797	-3123
13	G	717	2758	-2041	747	1353	-606	1464	4111	-2647
14	G	539	2742	-2203	418	1242	-824	957	3984	-3026
15	G	326	1367	-1042	285	571	-286	611	1939	-1328
16	G	327	1012	-685	382	560	-178	710	1572	-862
17	G	1262	3576	-2314	1228	2878	-1649	2490	6454	-3963
18	G	1507	3913	-2405	1096	2089	-994	2603	6002	-3399
19	G	922	3324	-2402	876	1340	-464	1798	4665	-2866
20	G	1723	3834	-2111	1398	2476	-1078	3121	6310	-3189
21	G	467	2438	-1970	533	770	-236	1001	3207	-2206
22	G	1093	3708	-2615	1309	844	465	2402	4552	-2150
23	G	649	3682	-3033	1093	716	377	1742	4398	-2656

Min	326	1012	-3033	285	560	-1738	611	1572	-3963
Max	2115	3913	-92	1398	2878	465	3121	6454	-862
Mean	1058	2431	-1373	729	1497	-768	1787	3928	-2141
Median	1076	2164	-1261	656	1406	-824	1742	3705	-2150
Std. Dev.	443	977	913	309	618	564	615	1302	876

were between 14 minutes (862 seconds) and 66 minutes (3936 seconds).

The boxplot (see Figure 2) shows all measured data for both programming tasks in a single boxplot, i.e. although the data is collected within-subject, the data is shown as if the data is measured between-subject. It appears relatively clear that for both programming tasks the development times using Groovy/Eclipse is higher than Java/Eclipse.

4.2 Repeated Measures ANOVAs

The previous section described only descriptive statistics and does not consider that the data was measured twice. From the sceptical point of view, it seems at least possible that the within-subject measurement influenced the measurements in an undesired way, i.e. that it appears as if the Groovy/Eclipse has a negative effect on development time, but that this is caused by the possible carry-over effect. Hence, it is worthwhile to compare the between-subject data, too.

Figure 3 illustrates a boxplot for the between-subject measurements when the tasks were processed first, i.e. the measured data does not contain any carry-over effect so far. The boxplot seems to imply that the results are comparable to the within-subject measurements as illustrated in Figure 2. In contrast to the within-subject measurements, it looks like the differences for task 2 are not that large without the within-subject measurement.

Figure 4 illustrates the boxplot for the second round, i.e. after the subjects already fulfilled the programming tasks with one language. This is the measurement after the carry-over effect took

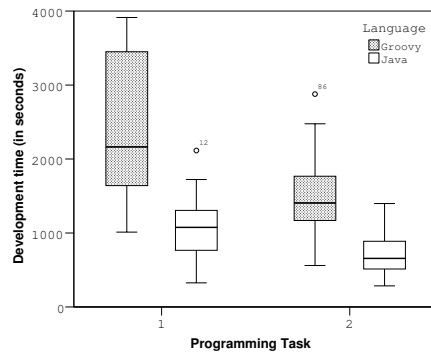


Figure 2: Boxplot for all measurements (within-subject measurement treated as between-subject measurement)

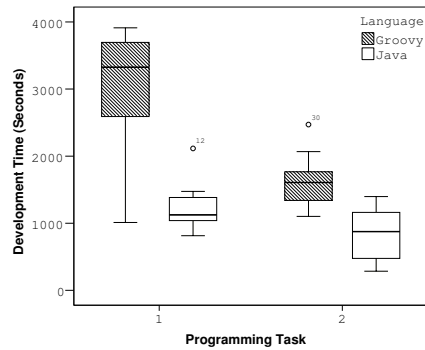


Figure 3: Boxplot for first round (between-subject measurement, when tasks were processed first)

place. Again, the boxplot looks comparable to the previous ones, i.e. the within-subject and the between-subject measurements for the first round.

In order to test, whether these visual impressions match the results of the experiment, we run a repeated measures ANOVA for the first, as well as for the second round.⁵

Two languages per subject (within-subject variable task and between subject-variable group): The results for the first round correspond to the impressions from the boxplots. First, the within-subject variable programming task is a significant factor (with $p < .001$, $\eta_p^2 = .72$). The between-subject variable group is significant, too ($p = .044$, $\eta_p^2 = .18$). Even more important, the interaction between programming task and group is significant ($p < .001$, $\eta_p^2 = .85$).

The interaction diagram in Figure 5 illustrates the detected interaction of the analysis. The group starting with Groovy required more time than the group starting with Java. When both groups switched programming language, the Groovy starter group (group 2, which now worked with Java) took less time than the group starting originally with Java (group 1, which now worked with Groovy).

The diagram 6 is another interaction diagram, where the x-axis shows the group and each task is represented in a separate line. Again, we see the same situation: While the development time for task 1 is less than task 2 for group 1 (Java starters), it is the opposite for the second task (again, Java requires less time). Hence, neither the task nor the group can be interpreted separately from each other.

⁵Note, that in contrast to previous experiments, each round is not equivalent to the programming language. Instead, group 1 starts task 1 with Java, but switches to Groovy for task 2, the other group vice versa.

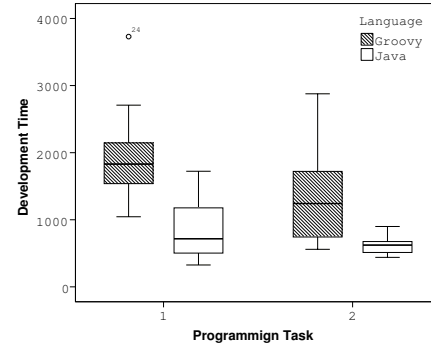


Figure 4: Boxplot for second round (between-subject measurement, when tasks were processed second)

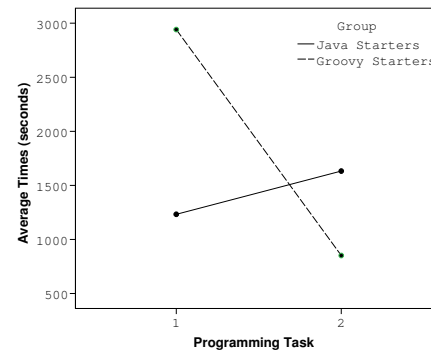


Figure 5: Interaction diagram for round 1 on programming task (Java starters process task 1 with Java, then switch for task 2 to Groovy, Groovy starters vice versa)

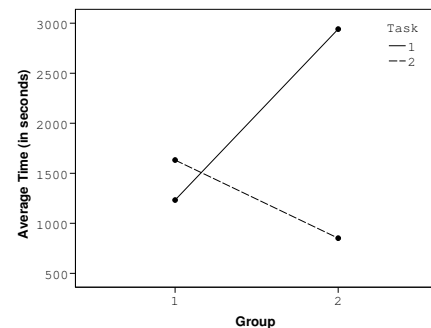


Figure 6: Interaction diagram for round 1 on group (Java starters process task with Java, then switch to Groovy, Groovy starters vice versa)

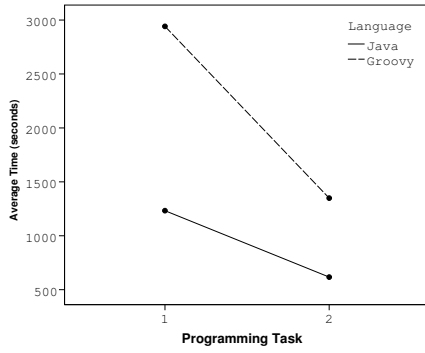


Figure 7: Interaction diagram for the initial language for each group (programming task)

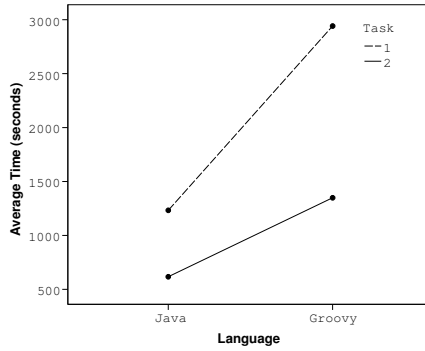


Figure 8: Interaction diagram for the 2nd language for each group (programming language)

For the second round, we observe comparable results. The within-subject variable programming task is significant ($p=.003$, $\eta_p^2=.34$) and the interaction is significant ($p<.001$, $\eta_p^2=.7$). However, group is not significant ($p=.34$).

One language per subject (within-subject variable language, between subject variable task): In order to analyze the data for the same language and different programming tasks, we need to compose data from both rounds: taking the Java results for group 1 of task 1 in round 1 and the Java results for group 1 of task 2 in round 2 and vice versa for group 2. Again, we perform the analysis in two steps: First, for the programming language each group started with, and then for the second language for each group. Hence, the resulting analysis is on the within-subject variable programming task and the between-subject variable programming language.

For the starting language, we receive the following results. The within-subject variable programming task is significant ($p<.001$, $\eta_p^2=.76$), the interaction between both variables is significant, too ($p=.001$, $\eta_p^2=.4$), and the variable programming language is also significant ($p<.001$, $\eta_p^2=.56$).

Again, it is worthwhile to take a look at the interaction diagrams. Figure 7 shows the interaction diagram where the x-axis illustrates the programming tasks and each line represent the programming language the subjects were working with. The development time for the subjects working with Java is, for a given programming task, less than the development time for Groovy. If we take a look at the interaction diagram that shows on the x-axis the language (see Figure 8), we see that the programming task 2 is always higher than programming task 1 for a given programming language.

Task	Task 1	Task 2
p-value	<.001	<.001
95% confidence interval (Java-Groovy)	[-1768 -- -979] s	[-1012 -- -524] s

Table 3: T-Test for within-subject comparison

For the second programming language, the results are slightly different: neither task nor the interaction is significant ($p=.15$, respectively $p=.19$), but the programming language is significant ($p<.001$, $\eta_p^2=.56$).

4.3 Task-specific analysis

The previous analysis determined that programming task is a significant factor—which is an argument why both programming tasks might not be analyzed at the same time.

Furthermore, the original design of the experiment was a within-subject design with the potential benefit that we get for each developer both measurements at the same time and an assumed counter-balance effect for the experiment. The initial goal being to run on each individual programming task a corresponding test, giving for each task a more precise approximation of the actual differences between the programming environments Java/Eclipse and Groovy/Eclipse.

However, the potential problem of such an analysis is, that the carry-over effect is too strong, i.e. that the participants in group 1 (starting with Java) have potentially a larger learning or novelty effect (the same could be true for group 2). Because of that, we test first, whether such an effect was measured, i.e. we run once again a repeated measures ANOVA with the within-subject variable programming language and the between-subject variable group.

The results of both tests are, that neither for the first task nor for the second task the variable group is significant ($p=.23$ with, respectively $p=.89$), giving us confidence that no significant difference between both groups exists and a combined analysis for both groups can be executed.

The intention is to run a paired sample t-Test for both languages. In order to do so, we test the differences of both languages for normality. In both cases, the Kolmogorov-Smirnov-Test is non-significant ($p=.2$). Hence, the hypothesis that both differences are normally distributed cannot be rejected. This matches the t-Test's assumptions.

Running the paired sample t-Test reveals the results as illustrated in Table 3: on both cases, there is a significant difference between Java and Groovy and the confidence interval of the difference is between 16 and 29 minutes (979 – 1768 seconds) for task 1 and between 9 and 17 minutes (524 – 1012 seconds) for task 2.

4.4 Results so far

We tested first (“Two languages per subject”), whether there is a significant interaction between task and group: since the participants in both groups switched the language, a language effect requires such an interaction. Such an interaction was detected, which supports the idea that there is a difference between Java/Eclipse and Groovy/Eclipse—although in the second round, the group itself did not turn out significant. The test was performed on the data of each round in separation, hence no learning effect can have influenced the result.

Second, we tested (“One language per subject”) whether one single language for each group was significant. Since the participants in each group solved each task with the same language (although doing another task in between), this is an indicator for a difference in the language, too. For each round, we received a sig-

nificant variable programming language.

Then, we estimated the differences in development time based on a t-Test (checking the test’s assumptions beforehand). It turned out that the programming environment Java/Eclipse revealed a clear positive effect in comparison to Groovy/Eclipse. The differences ranged between 16–29 minutes for task 1, and 9–17 minutes for task 2. Although these may not seem like impressive numbers, they should be compared to the time participants required to solve the programming tasks. For task 1 the median was 18 minutes for Java (and 36 minutes for Groovy), for task 2 the median for Java was 11 minutes (23 minutes for Groovy). Taking these medians into account, this implies that the development time for Groovy/Eclipse was about twice as much as the development time for Java/Eclipse.

5. EXPLORATORY STUDY

The results of the previous section are already impressive and show that the differences measured in previous experiments still hold—even if an IDE, JavaDocs, and parameter names with hints about the expected types are available. However, it is desirable not only to understand that there is a difference between the development times, but possible reasons why such differences were measured.

In the experiment by Kleinschmager et al. we observed that the number of file switches, i.e. whenever a developer switched into a different class file, concorded with the measurement of development time. Hence, it was concluded that these file switches might be responsible for the measured differences.

In the presence of IDEs the situation might be potentially different. When features such as code completion are available, there is no need for developers to switch into different files—instead, this could be achieved by browsing the code completion window (at least in cases where the IDE for the dynamically typed language shows relevant proposals). However, at the same time, it seems likely that people using the dynamically typed language use other means to explore the code, such as the IDE’s search functionality.

Because of this, we perform an exploratory study not only on the file switches (as being done in the previous study), but also on the number of searches being used within the IDE, the frequency of code completion usage, and the time each participant spent in the code completion menu. The raw measurements and descriptive statistics for these numbers are illustrated in Figure 4.

In this section, we follow the approach of section 4.3. We check first that the variable group is not a significant factor. Then, we combine the within–subject measurements, and run either a t-Test (in case the differences are normally distributed) or a Wilcoxon-Test (in case the differences are not normally distributed).

5.1 File switches

For the file switches, the variable group does not turn out to be significant in a repeated measures ANOVA, neither for the first round ($p=.1$) nor for the second round ($p=.08$). Hence, we assume that the variable group is not an influencing factor and that we do not need to differ between the groups in the analysis. Additionally, we test whether the differences between Java/Eclipse and Groovy/Eclipse are normal distributed. We receive in both cases $p=.2$ from the Kolmogorov-Smirnov-Test, hence we assume a normal distribution and run the paired sample t-Test.

Task	Task 1	Task 2
p-value	.03	.002
95% confidence interval (Java-Groovy)	[-50 – -21]	[-37 – -13]

Table 5: Two-sided paired sample T-Test for number of file switches

The results of the paired sample t-Test are shown in Figure 5: in both cases the differences between Java/Eclipse and Groovy/Eclipse are significant. The 95% confidence intervals for the Java-Groovy differences show that the Java/Eclipse developers switches less often between the files: between 21 to 50 (task 1), and 13 to 37 (task 2) less file switches than the Groovy/Eclipse developers. Hence, the results for the file switches are comparable to the results in the previous experiment: both the development time and the file switches lead to the same result (less effort for Java/Eclipse developers), and it seems at least possible that the file switches are a possible explanation for the measured differences in development time.

5.2 Search counts

For the number of search counts, the variable group is not significant for the first or for the second round ($p=.81$, respectively $p=.6$). However, the Kolmogorov-Smirnov test for normality does not permit to assume a normal distribution ($p=.02$ in the first round, $p=.001$ in the second round); the assumptions for the parametric t-Test do not hold. Instead, we apply a non-parametric Wilcoxon test.

Task	Task 1	Task 2
p-value	=.001	<.001
Positive Ranks (Groovy-Java)	18	17

Table 6: Two-sided Wilcoxon-Test of search counts

The results for the Wilcoxon test are illustrated in Figure 6: while the test does reveal any confidence interval, the p-values show (in combination with the positive ranks) that the Groovy/Eclipse group performed significantly more searches using the IDE. To picture how large this difference is, we look into the measurements and descriptive statistics in Figure 4: while only six subjects in the Java/Eclipse groups used the IDE’s search functionality for task one (hence the median of 0), only two subject in the Groovy/Eclipse group did not the search for functionality (median of 3). A similar result can be seen for the second task (median 0 vs. median 3). Hence, it can be concluded that the Groovy/Eclipse group used significantly more often the search capabilities of the IDE, which is another indicator that their code exploration was more intensive.

5.3 Number of code completions

The number of code completions, i.e. how often the subjects opened the code completion menu is another possible indicator that participant who used Groovy/Eclipse spent time on exploring the code. In both rounds, the group is not significant (it is significant in the first round with $p=.06$, while in the second round the p-value is more definitive with $p=.13$). Although we are aware that the first value is not that far from being significant, we still assume here that we can combine both groups. Here, both differences between Java/Eclipse and Groovy/Eclipse can be assumed to be normally distributed (the KS-Test being .2 in both cases).

Table 4: Measurements for exploratory study

Subject	Start	Number of File switches						Number of Searches						Number of Code Completions						Code Completion Time (in seconds)						
		Task1			Task2			Task1			Task2			Task1			Task2			Task1			Task2			
		Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java
1	J	19	33	-14	19	54	-35	1	6	-5	0	26	-26	30	24	6	20	34	-14	94	73	21	93	117	-23	
2	J	52	39	13	3	42	-39	9	6	3	1	10	-9	36	36	0	34	40	-6	122	332	-210	95	251	-156	
3	J	29	96	-67	5	56	-51	0	1	-1	0	0	0	19	23	-4	14	11	3	133	169	-36	91	49	42	
4	J	31	66	-35	9	35	-26	0	0	0	0	7	-7	15	14	1	19	16	3	170	87	83	46	40	7	
5	J	9	1	8	1	7	-6	0	1	-1	0	9	-9	24	37	-13	26	35	-9	280	450	-170	138	295	-157	
6	J	24	71	-47	11	65	-54	0	1	-1	2	3	-1	16	31	-15	30	38	-8	58	121	-64	96	178	-83	
7	J	40	31	9	32	63	-31	0	19	-19	0	21	-21	25	19	6	20	16	4	75	89	-14	79	74	5	
8	J	19	62	-43	15	46	-31	0	2	-2	0	10	-10	12	18	-6	20	21	-1	37	90	-53	58	56	2	
9	J	38	55	-17	38	44	-6	0	1	-1	0	2	-2	18	24	-6	22	16	6	152	107	45	71	65	6	
10	J	36	21	15	1	15	-14	0	3	-3	0	3	-3	21	25	-4	29	15	14	73	92	-19	73	41	31	
11	J	65	83	-18	1	79	-78	0	5	-5	2	23	-21	17	14	3	12	10	2	72	82	-11	63	39	25	
12	J	21	88	-67	1	27	-26	7	5	2	0	5	-5	42	35	7	25	40	-15	356	294	62	59	272	-214	
13	G	5	59	-54	5	17	-12	0	1	-1	0	0	0	9	26	-17	12	10	2	33	184	-151	31	34	-2	
14	G	32	77	-45	34	50	-16	0	0	0	0	0	0	24	27	-3	28	33	-5	69	207	-139	85	311	-226	
15	G	3	39	-36	1	7	-6	1	13	-12	1	2	-1	13	22	-9	16	25	-9	63	86	-23	55	123	-68	
16	G	3	65	-62	3	13	-10	0	4	-4	0	3	-3	14	11	3	16	18	-2	45	31	14	77	121	-44	
17	G	85	121	-36	85	125	-40	2	14	-12	0	44	-44	8	12	-4	11	13	-2	18	54	-36	21	42	-21	
18	G	57	86	-29	40	127	-87	0	11	-11	0	3	-3	28	29	-1	25	32	-7	93	204	-110	57	139	-82	
19	G	32	100	-68	33	77	-44	3	7	-4	0	0	0	8	25	-17	15	11	4	28	169	-141	52	53	-1	
20	G	55	59	-4	60	72	-12	0	3	-3	0	29	-29	7	13	-6	10	9	1	25	83	-58	37	25	12	
21	G	17	41	-24	11	9	2	0	8	-8	0	2	-2	21	38	-17	27	32	-5	49	245	-196	77	171	-95	
22	G	56	132	-76	79	53	26	0	1	-1	0	0	0	10	18	-8	18	16	2	33	128	-95	44	44	0	
23	G	42	168	-126	90	63	27	0	0	0	0	0	0	7	20	-13	10	10	0	13	88	-75	30	34	-4	
Min		3	1	-126	1	7	-87	0	0	-19	0	0	-44	7	11	-17	10	9	-15	13	31	-210	21	25	-226	
Max		85	168	15	90	127	27	9	19	3	2	44	0	42	38	7	34	40	14	356	450	83	138	311	42	
Mean		33	69	-36	25	50	-25	1	5	-4	0	9	-9	18	24	-5	20	22	-2	91	151	-60	66	112	-45	
Median		32	65	-36	11	50	-26	0	3	-2	0	3	-3	17	24	-4	20	16	-1	69	107	-53	63	65	-4	
Std. Dev.		21	38	34	29	33	28	2	5	5	1	12	12	9	8	8	7	11	7	84	101	81	27	92	77	

Task	Task 1	Task 2
p-value	.21	.002
95% confidence interval (Java-Groovy)	—	[-95 -- -25]

Task	Task 1	Task 2
p-value	=.004	=.041
Positive Ranks (Groovy-Java)	18	14

Table 7: Two-sided paired sample T-Test for number of file switches

The t-Test illustrates here an interesting result, because only for the second task, the number of code completions are significant ($p=.002$) and the Groovy developers used between 25 and 95 more times the code completion than the Java/Eclipse group. For the first task, it is noteworthy that that p-value is not even close to significant ($p=.21$, see Figure 7).

5.4 Code Completion Times

While the previous measurement can be considered as rather disappointing, because the Groovy/Eclipse developers did not use in both tasks the code completion more often than the Java/Eclipse developers, it seems still interesting to see whether there is a difference in the time the developers spent in the code completion menu.

Repeating our previous analysis reveals that the between-subject variable group is not significant in the repeated measures ANOVA (neither for the first nor for the second task; $p=.08$ and $p=.28$ respectively). However, the differences in time is not in both cases normally distributed: while the KS-Test reveals a non-significant result for task 1 ($p=.2$), it is significant for task 2 ($p=.002$). We decided to run the non-parametric test on both programming tasks.

Table 8: Two-sided Wilcoxon-Test of search times

Figure 8 contains the results of the two-sided Wilcoxon test. For both programming tasks, we observe a significant difference ($p=.004$, and $p=.041$). In order to understand how large these differences are, we take again a close look into the measurements and descriptive statistics in Figure 4: in both cases the difference in the medians is about one minute (60 seconds). Again, this is an indicator that the participants spent more time on exploring the code when using Groovy/Eclipse in comparison to Java/Eclipse.

6. SUMMARY

Section 4 determined first that there was a significant difference between Java/Eclipse and Groovy/Eclipse in development time. This was done by repeated measure ANOVAs that detected a significant interaction between task and group which gave confidence that the chosen design worked out. Then, we were able to approximate the size of the differences between Java/Eclipse and Groovy/Eclipse by reporting the confidence intervals which are the result of the t-Test (whose assumptions had been tested upfront) and we showed that the differences are quite large (between 9 and 29 minutes – where the working time for the statically typed solutions was approximately 18 and 11 minutes).

Then, we repeated the exploratory analysis as done in the previ-

ous experiment by showing that again the number of file switches is an indicator for the differences between the statically and the dynamically typed solutions – the number of file switches was originally interpreted as an indicator for the effort in code exploration and the here introduced experiment seems to strengthen this argument. Additionally, we showed that the new IDE features also give to a certain extent such indicators – the number of searches in the IDE as well as the time spent on code completion reveal the same significant difference as the number of file switches – as well as the development time.

However, it should be made explicit, that the exploratory study should not be used too extensively for the argumentation, because it is still just an exploratory study – it’s possibly influencing factors were not controlled.

7. DISCUSSION AND CONCLUSION

This paper described an experiment that compares the use of static and dynamic programming languages for using unknown APIs; an experiment, that is part of a larger experiment series that consists currently of eight controlled experiments that study different facets of type systems.

While such an experiment has been already performed by Kleinschmager et al., and while the previous experiment showed a clear, significant positive effect of static type systems in comparison to dynamic type systems, the here described experiment now removed a number of problems that threatened the validity of the previous experiment: a serious IDE with industrial relevance (Eclipse) has been used by the subjects in the experiment. Additionally, the experiment used a documentation that is widely used in the Java world (JavaDocs) and (in contrast to the experiment by Kleinschmager et al.) the experiment did not artificially set any constraints on parameter names in the API. By removing these threats from the experiment, the experiment reacts on criticism against the previous experiment: that the experiment’s result cannot be generalized to industrial setting. Similarly to the previous experiments, this experiment uses Java and Groovy as representatives for statically and dynamically typed languages.

It turned out that the experiment’s results did not contradict the previous experiment’s results: again, a positive effect of the statically typed programming language Java was measured. This might be an indicator that one often found critique against a number of experiments from the experiment series may not be as strong as previously assumed: the use of a simplified IDE in an experiment did not reduce the experiment’s validity.

In addition to the replication of the previous results, this experiment reveals additional insights into possible explanations for the measured time differences. While the original experiment mainly concentrated on file switches as an indicator for the differences between the statically and dynamically typed languages, this experiment indicates that the use of additional code navigation and exploration tools within the IDE can be used, too. The use of the search functionality as well as time developers spent in the code completion window revealed comparable results in comparison to development time: the measurements imply a positive effect of the statically typed language. Having said this, we also have seen that just the number of opened code completion windows are no such indicator. Additionally, we have seen that the absolute time spent in the code completion window can hardly be used to explain the difference in development time: the absolute differences were relatively small (with the median of one every minute). However, these measurements are only indicators for the developer’s navigation and exploration that takes place in some other actions (which might be just reading code), where the actual loss of time takes

place.

Our conclusion of the paper can be summarized as follows:

1. This paper gives (again) evidence that static type systems help using unknown APIs.
2. The statement from 1., for which already experimental evidence existed before, still holds, even if IDEs (with code completion), JavaDocs, and natural parameter names are used within the experiment.
3. With the number of searches and the code completion time, two additional metrics were found which (in addition to the number of file switches) indicate the difference in time and which might be indicators for code exploration.

This experiment additionally contributes to the area of experimental research in a different way: it contributes to the endless debate about the benefit of experimental results gathered from small-scale experiments that largely restrict developers, because (another) situation is shown where the results of such restricted experiments were replicated in a broader environment.

Of course, this work is not the final answer to the discussion if and when static type systems have a positive impact on development time: the chosen IDE in the experiment for the dynamically typed language could be much improved. However, it is unclear whether it is possible to improve the IDE up to the point where one could no longer observe a significant benefit of static type systems on API — something that should be studied in the future.

8. REFERENCES

- [1] Andrew Bragdon, Steven P. Reiss, Robert C. Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeptura, and Joseph J. LaViola Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of ICSE 2010*, pages 455–464, 2010.
- [2] Kim B. Bruce. *Foundations of object-oriented languages: types and semantics*. MIT Press, Cambridge, MA, USA, 2002.
- [3] Mark T. Daly, Vibha Sazawal, and Jeffrey S. Foster. Work in progress: an empirical study of static typing in ruby. *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU), Orlando, October 2009*, 2009.
- [4] Uri Dekel and James D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proceedings of ICSE 2009*, pages 320–330, 2009.
- [5] Uri Dekel and James D. Herbsleb. Reading the documentation of invoked API functions in program comprehension. In *Proceedings of ICPC 2009*, pages 168–177, 2009.
- [6] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger canvas: Industrial experience with the code bubbles paradigm. In *Proceedings of ICSE 2012*, pages 1064–1073, 2012.
- [7] Robert DeLine and Kael Rowan. Code canvas: zooming towards better development environments. In *Proceedings of ICSE 2010*, pages 207–210, 2010.
- [8] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. How do API documentation and static typing affect API usability? In *Proceedings of the ICSE 2014 (accepted for publication)*, ICSE ’14, 2014.

- [9] J. D. Gannon. An experimental evaluation of data type conventions. *Commun. ACM*, 20(8):584–595, 1977.
- [10] Stefan Hanenberg. Doubts about the positive impact of static type systems on programming tasks in single developer projects - an empirical study. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, LNCS 6183, pages 300–303. Springer, 2010.
- [11] Stefan Hanenberg. 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, pages 22–35, New York, NY, USA, 2010. ACM.
- [12] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. Springer, 2013 – published online in dec 2013.
- [13] Michael Hoppe and Stefan Hanenberg. Do developers benefit from generic types?: An empirical comparison of generic and raw types in java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 457–474, New York, NY, USA, 2013. ACM.
- [14] Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. 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*, pages 153–162, 2012.
- [15] Walid Maalej and Martin P. Robillard. Patterns of knowledge in api reference documentation. *IEEE Trans. Software Eng.*, 39(9):1264–1282, 2013.
- [16] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 683–702. ACM, 2012.
- [17] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.
- [18] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of ICSE 2012*, pages 69–79, 2012.
- [19] Fernando Olivero, Michele Lanza, Marco D’Ambros, and Romain Robbes. Enabling program comprehension through a visual object-focused development environment. In *Proceedings of VL/HCC 2011*, pages 127–134, 2011.
- [20] Cyrus Omar, YoungSeok Yoon, Thomas D. LaToza, and Brad A. Myers. Active code completion. In *Proceedings of ICSE 2012*, pages 859–869, 2012.
- [21] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.
- [22] Lutz Prechelt and Walter F. Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Trans. Softw. Eng.*, 24(4):302–312, 1998.
- [23] Romain Robbes and Michele Lanza. Improving code completion with program history. *Autom. Softw. Eng.*, 17(2):181–212, 2010.
- [24] Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.
- [25] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *Proceedings of ICSE 2012*, pages 255–265, 2012.
- [26] Samuel Spiza and Stefan Hanenberg. Type names without static type checking already improve the usability of APIs - as long as the type names are correct: An empirical study. In *Proceedings of the Modularity 2014 (accepted for publication)*, AOSD '14, 2014.
- [27] Marvin Steinberg and Stefan Hanenberg. 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.
- [28] Andreas Stuchlik and Stefan Hanenberg. Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 97–106, Portland, Oregon, USA, 2011. ACM.
- [29] Iyad Zayour and Hassan Hajjdiab. How much integrated development environments (IDEs) improve productivity? *JSW*, 8(10):2425–2431, 2013.