How Do API Documentation and Static Typing Affect API Usability?

Stefan Endrikat
University of Duisburg-Essen
Germany
stefan.endrikat@stud.uni-due.de

Stefan Hanenberg
University of Duisburg-Essen
Germany
stefan.hanenberg@icb.uni-due.de

Romain Robbes
PLEIAD @ DCC
University of Chile
Chile
rrobbes@dcc.uchile.cl

Andreas Stefik
University of Las Vegas
United States
stefika@gmail.com

ABSTRACT

When developers use Application Programming Interfaces (APIs), they often rely on documentation to assist their tasks. In previous studies, we reported evidence indicating that static type systems acted as a form of implicit documentation, benefiting developer productivity. Such implicit documentation is easier to maintain, given it is enforced by the compiler, but previous experiments tested users without any explicit documentation. In this paper, we report on a controlled experiment and an exploratory study comparing the impact of using documentation and a static or dynamic type system on a development task. Results of our study both confirm previous findings and show that the benefits of static typing are strengthened with explicit documentation, but that this was not as strongly felt with dynamically typed languages.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Experimentation, Documentation, Human Factors

Keywords

API Usability, Documentation, Static Type Systems

1. INTRODUCTION

The software engineering community has developed a renewed interest in studies of Application Programming Interfaces (APIs) and their documentation, although the importance of good APIs is know for some time. In his landmark essay, Brooks states that "The most radical possible solution for constructing software is not to construct it at all. [3]" Brooks describes how APIs and frameworks are promising attacks on the essential complexities of software.

While most reasonable scholars would not deny the benefits of high quality APIs, the impact of documentation is less clear. For example, consider several claims on the benefits of documentation in the academic literature:

> One user mentioned that the equivalent of an empty API documentation with only the type name, (e.g., Doxygen), could save him hours of source code exploration.[5]
>
> The biggest hurdle when learning an API is the documentation. If the documentation for an API is good, it solves 99% of your problems. [25]

While on the surface, many might consider such claims reasonable if documentation is of truly excellent quality, studies show a more nuanced state-of-the-practice. For example, not all developers trust documentation, preferring to investigate the source code or to talk to colleagues. This is due in part to missing documentation or the suspicion that documentation may be out of date [26]. Further, an analysis of knowledge exchange sites (e.g., Stack Overflow) conducted by Parnin *et al.* showed that 87% of the classes in the Android APIs are covered by questions and answers coming on the Stack Overflow web site [20]. Indeed, the exact, quantifiable, benefits of documentation are in need of further investigation.

Proponents of statically typed programming languages further claim that static typing provides a form of *implicit documentation* embedded in the source code. For example, Pierce states that "The type declarations in procedure headers and module interfaces constitute a form of documentation, giving useful hints about behavior [22]." Similarly, Bruce claims that "[...] type annotations provide documentation on constructs that make it easier for programmers to determine how the constructs can or should be used [4]."

We have undertaken an empirical investigation into these and similar claims, finding in previous work some empirical evidence for their arguments. Specifically, controlled experiments comparing static and dynamic type systems have shown that, for some tasks, type annotations in statically typed code do lead to increases in productivity [14, 17]. However, while previous work appeared to support the use of static type systems, all previously reported experiments included *only the implicit documentation provided by the type system.*

Thus, to better evaluate the hypotheses presented by Pierce and Bruce, we conducted a randomized controlled trial with a 2x2 experimental design, and an exploratory study on possible causes,

ICSE 2014 Hyderabad, Andhra Pradesh, India

comparing developers using either static or dynamic typing and with or without accompanying documentation. Given this setup, we see at least two possible outcomes. First, it could be the case that adding documentation is an equalizing factor, affording equal or better productivity to those with dynamic typing. Second, type annotations and static checking could benefit programmers even further. The results of our experiment show evidence toward the second explanation—the addition of explicit documentation affords even greater productivity gains to developers using statically typed languages. Those with dynamic typing did not appear benefit as much from the addition of explicit documentation.

While the previous paragraph gives the primary finding of our paper, to try and garner possible causes for the behavior we observed, we conducted an exploratory analysis of the interaction logs of our developers while completing the tasks. Results from this analysis are not definitive, but hint that the number of file switches may be a leading indicator on why such differences occur. We also report on additional indicators from the logs.

Structure of the paper We first describe related work (Section 2): empirical studies on documentation and type systems. Then, we describe the design of the experiment in detail (Section 3). Next, we present the results of our experiments (Section 4). We then perform an exploratory study based on the interaction data we recorded (Section 5). We close the paper with a discussion (Section 6), possible threats (Section 7), and conclude (Section 8).

2. RELATED WORK

2.1 Studies of APIs and Documentation

Controlled experiments An early experiment by Tenny [28] compared two types of procedures (inlined or external procedures) and the occurrence of comments. The experiment showed an effect of comments on understandability of short programs, measured by a questionnaire. An experiment by Tryggeseth [30] found that documentation had a positive impact on the time necessary to complete a maintenance task in a C++ system; the subjects were approximately 20% faster. The solution was written in pseudocode. Unfortunately, the experimental design was not fully documented.

An experiment by Prechelt *et al.* [24] found that documenting design patterns in source code (in addition to the existing documentation), had a positive effect on development time and correctness. Entities forming a design pattern were more easily recognized and processed as a single "chunk" of data, rather than several entities. On the other hand, Ellis *et al.* [10] found conflicting evidence on the usefulness of the Factory design pattern.

Another experiment by Bandi *et al.* [2] explored the impact of three complexity metrics on maintenance time: Interaction Level, Interface Size, and Operation Argument Complexity. All three metrics were found useful to predict the maintenance time taken by several groups of students (93 students in total).

An experiment by Arisholm *et al.* [1] showed that there is a positive impact of UML documentation on task correctness, but not on time. The experiment had several tasks on the same systems and showed that subjects were more likely to succeed in later tasks when the documentation was present. A recent pilot experiment by Leotta *et al.* [15] seems to, unsurprisingly, confirm that outdated documentation is less useful than up-to-date documentation. However, a recent study by Petre [21] showed that usage of UML in the practice is limited, yielding the necessity to investigate additional forms of documentation.

Dekel and Herbsleb evaluated the effectiveness of eMoose, a "knowledge-pushing" tool, that highlights critical parts of the documentation and makes them more obvious in the IDE [6]. A controlled experiment was conducted, where participants used eMoose in a variety of tasks where significantly more likely to solve the tasks than users from the control group (for four out of 5 tasks). A follow up video analysis study fleshed out how the documentation was used in more detail [7].

Exploratory studies Robillard and DeLine conducted a study on what makes APIs hard to learn [25], based on surveys and interviews; they found a variety of obstacles that developers faced, and documented the implications of them. In particular, developers prefer a continuous presentation of the API, focused on its relation to the problem domain (how a set of API elements are used together to solve a given task), as opposed to a hyperlinked documentation of individual API elements.

Dagenais and Robillard interviewed developers to find out what decisions were taken by developers with respect to documentation, and the consequence it had on the project [5]. Timely update to the documentation was found to improve code quality. Three types of documentation were observed: task-related documentation aimed at newcomers (getting started type of documentation), reference documentation documenting program elements, and conceptual documentation.

An exploratory study by Duala-Ekoko and Robillard [9] analysed video recordings of twenty participants working on a set of tasks for which they had to discover unknown APIs. The main information sources where the API documentation, and the web. They formulated a list of 20 questions the developers asked when solving the tasks, and made additional observations about the challenges developers faced, such as discovering types with non-obvious relationships, and difficulties related to the usage of exceptions.

Roehm *et al.* conducted a study of program comprehension in industry, based on an observational study of 28 programmers from seven companies [26]. It highlighted important facts pertaining to documentation: source code was more trusted than documentation due to possible mismatches, or the code not being covered; direct communication was preferred over the documentation for similar reasons. On the other hand, understanding the rationale behind the code (its purpose and how to use it) was found to be very difficult ("exhausting") when the code is not documented.

Maalej and Robillard developed a taxonomy of the types and patterns of knowledge commonly found in APIs [16], based on two large APIs, the JDK and .NET. The types of knowledge found in the APIs were: functionality, concepts, directive, purposes, non-functional requirements, control flow, structure, patterns, examples, environment issues, and references. They also found a surprisingly high amount of content (40 to 50%) that brought little value.

Parnin *et al.* investigated the recent phenomenon of crowd documentation, as measured by the questions about API elements asked and answered on the Stack Overflow website [20]. They found that over 35000 developers contributed in crowd-documenting Android, covering 87% of the classes. Additionally, there was a strong correlation between usage of API elements and the amount of discussions on Stack Overflow.

2.2 Type Systems as Documentation

Several empirical studies have been completed on type systems. We focus our discussion here on those that are related to the impact of documentation in relation to type systems. Perhaps the earliest known controlled study on type systems was from Gannon. This early experiment showed an increase of reliability for subjects using a statically-typed language [11].

An experiment by Prechelt and Tichy [23] compared two versions of C, one which the types of arguments of procedure calls are checked (ANSI C), and another (K&R C), in which the types are not. Both programs came with documentation and type annotations. One task showed differences between the two treatments, another did not. Thus the experiment gave doubt to the idea that type annotations are useful as documentation only.

In terms of our own work, several previous studies investigated the impact of using static and dynamic type systems on the usage of undocumented APIs. We investigated a variety of maintenance tasks, including identifying classes in an undocumented API [14]. Through these studies, we observed that as the number of classes that needed to be identified grew, the advantage of the static type system grew as well. Other tasks related to fixing type errors and semantic errors were investigated, with an advantage found when fixing type errors, but none found when fixing semantic errors.

In a second experiment [17], we focused on class identification tasks in an undocumented API. We also defined several tasks of varying complexity, and observed a similar tendency: the larger or more complex the types to identify were, the greater the advantage was found towards the statically typed language. We complemented the study with an exploratory story of the participants' interaction log, which shed some light in our findings: as tasks got more complex, developers using the statically typed language tended to open less files and switched between them less often.

In a subsequent experiment [12], we investigated the usefulness of generic types compared to raw types under similar conditions. The experiment showed that generic types helped for using undocumented APIs, but not for type error fixing tasks or extension tasks.

3. EXPERIMENT DESCRIPTION

3.1 Initial considerations

The goal of the experiment is to compare the effects of documentation and type system (and their potential interaction). Several decisions need to be made on: the research questions, the kind of documentation, the programming language, the programming tasks, and the measurements. Similarly to our previous experiments, we measure the effect of documentation and type system on the time developers take to complete a programming task.

Number and type of subjects The goal was to define and execute an experiment with a relatively small number of participants: we planned for 20 to 30 participants. Additionally, the time for each participant should be relatively small. Unfortunately programming tasks require on the order of hours to be completed. We settled on a maximum duration of five hours.

The low number of partipants also means that the deviation among participants should be as low as possible. Otherwise, there is the danger that the measurements are hidden by confounding factors. These constraint directly influence the choices of documentation, programming language, programming environment and programming task. This amount of participants is common in Software Engineering. The experiment of Pankratius [19] had 13 participants, Dekel's [6] had 26, and Wettel's [32] had 42. Our previous experiments on type systems all had between 20 and 30 participants. As in most software engineering experiments, subjects are students (see [29] for details).

Type of documentation. Obviously the type and quality of documentation chosen strongly influences the results; giving misleading or outdated documentation would negatively impact the results [15]. Since the space of possibilities is large, we sought a format of documentation that would be commonly encountered and useful.

The first criteria is length: the documentation must not be too exhaustive, otherwise most of the development time would be spent in reading the documentation with the risk that the participants would not finish the task. Another risk is that participants do not read the documentation carefully if it is too long, as seen in Dekel's study [7]. Additionally, there must be the possibility that participants search in wrong parts of the document for a given solution—a situation that frequently occurs in web searches. So the size of the documentation must be carefully balanced.

Although there are many different documentation formats (such as JavaDocs, UML diagrams, etc.) we settled on free-form text which gives a short description of the API and source code examples that show how to use the API. This matches the needs expressed in Robillard's study [25]: linear documentation instead of small interlinked pieces, and examples showing how various API elements are used together. This is also similar to the format of questions and answers in Stack Overflow, strongly suggesting this is the type of documentation developers favor. We did not investigate UML documentation as it was already investigated by Arisholm *et al.* [1], and discarded JavaDoc-type documentation (hyperlinked documents matching the program structure) as it did not match developer preferences in Robillard's study.

A challenge is to give documentation which is not too close to the programming task. Otherwise, the task is just to type text. The examples given were not an exact match with the task to achieve.

Choice of programming language We needed a language familiar to participants—else the time required to teach and learn it would consume most of the experiment time. At the same time we needed a language supporting both static and dynamic typing, to reduce the differences between the experimental treatments.

Since most of the participants come from the University of Duisburg-Essen, where Java is being taught right from the first semester, we decided to use the programming language Dart. Dart can be used similarly to Java, but has optional types annotations.

Choice of IDE Similarly to previous experiments, we use a very simplified IDE which consists only of an editor with syntax highlighting, a treeview for the files within the software project and buttons for running test cases. This reduces the risk that IDE proficiency affects the results.

Choice of Tasks A first factor is size: excluding warmup tasks, experiments in the literature range from one single task [19], to ten tasks or more [32]. Since documentation takes time to read, short tasks would give an unfair advantage. Likewise, the effect of static type systems is more visible on larger tasks [17, 14]. Contrary to our previous experiments, this experiment uses a single task (plus warmup), whose expected duration is several hours. This also prevented us to use a within-subject design, as the task would be too long to be repeated.

The type of task to perform has an important effect on the results. We can not expect static type systems or documentation to have the same effect regardless of the task; we observed this in a previous experiment [14]. Hence we need a task that would not favor too much one or the other treatments (task or documentation), and would fit in the API usability scenario. We chose a task where participants use the API in a way that objects need to be configured and passed to the API. Previous experiments have shown that static type system benefit in such a situation, but we expect documentation to also benefit. To reduce confounding factors, no other API is necessary, and the task can be solved without using loops or conditionals (to reduce confounding factors).

We considered other tasks in [14]: Type error fixing tasks, where we found an advantage of the type system, and would not expect an advantage of documentation; and semantic error fixing tasks, where we found no advantage of the type system, but would expect an advantage of the documentation, as in Dekel's study [6].

3.2 Research questions

Our independent variables being documentation and type system, and our dependent variable being time, our first two research questions are obvious:

- **RQ1**: When a free–text form documentation is given, do developers actually use it?
- **RQ2**: Is there a difference in development time for a given programming task between statically and dynamically typed programming languages, with or without a given free–text form documentation?

One potential confounding factor in measuring development time is, that potentially developers spend a lot of time only reading documentation. Hence, we are also interested in the results of RQ2 without taking the reading time into account:

- **RQ3**: Assuming that developers have already read and understood the documentation, does that influence the potential difference in development time (from RQ2)?

3.3 Programming Task and API

The domain for the programming task was a delivery service: different kinds of packages from a vendor (with multiple stores) need to be delivered to a customer. The domain was chosen as it is easily understandable by the participants.

The programming task is to deliver a product (a kitchen) to a customer under some constraints: the shortest way should be taken, i.e. a store that is as close as possible to the target address and a carrier should be used that is as close as possible to this store. To solve this task, a method is given to the participants whose header contains key objects for the customer and the product objects.

```
void task(Connections datalinks,
     PersonalIdentification customer,
     KitchenKey kitchen) {

  Delivery d = Delivery.createDelivery();
  Product prod = datalinks.getCatalogs().
    getHomeAidsCatalog().
    getProductById(kitchen);
  d.setDeliveryObject(prod);

  Customer c = datalinks.getLibraries().
    getCustomerLibrary().getClient(customer);
  Contact ct = Contact.createContact();
  ct.setPerson(c);
  Address ca = // get address via connections ...

  d.setDestination(ct);

  Store s = datalinks.getManagers()...
    getNearestStoreForProduct(prod, ca);
  d.setSource(s);
  Shipper shipper =
    ...getNearestCarrierForProduct(prod, s);
  d.setTransporter(...);
  // ...set some additional properties ...
  return d;
}
```

Figure 1: Example solution for the task (statically typed)

The challenge for developers is to locate all the API elements necessary to properly configure a delivery object. Figure 1 shows an extract from a possible solution (some code is omitted for space reasons). For the dynamically typed code all type annotations (especially in the header) would be removed. The figure also shows how similar Dart is to Java, and also that neither loops nor conditions are required to solve the task.

The API's source code was available to the participants. It follows the design decisions of previous experiments (see [17, 14]). First, the methods to be used by the developers require parameters and return objects that are only basic data types or object from the API itself (it does not use standard library classes to reduce confounding factors due to expertise in the standard APIs). Second, the API is rather small (about 2000 lines of code), but is large enough that developers can not completely read the code within the experiment. For instance, it is larger than the systems used in the experiment by Arisholm, which had 293 and 338 lines of code *et al.* [1]. Finally, the API does not contain additional documentation (no JavaDocs, no comments, etc.) in order to control the documentation variable in the experiment more strictly.

Finally, developers were equipped with test cases. When all the tests pass, the programming task has been solved. The source code of the tests was not available to the subjects.

3.4 Documentation

As motivated in section 3.1, we decided to put text documents into the project, i.e. separate ASCII files, that describe of a certain part of the API and provide the corresponding code examples. The names of these ASCII files give a hint about their content. For example, a file named delivery.txt describes the meaning of delivery objects in general and provides code snippets creating and instantiating a delivery object and related objects (illustrating how several API elements are used together, as recommended in [25]). The documentation also contains parts of the API which were not relevant for the programming task. For instance, the documentation contains code snippets that show how a certain kind of carrier can be requested for the delivery, while the programming task requires to select a carrier depending on the distance from the store.

Figure 2 gives an impression of the documentation delivered to the participants.[1].

Each document starts with a general description of its content and then gives a more detailed description of specific topics, including code snippets. For example, the code in Figure 2 shows how to create and initialize a contact object – which is required for solving the programming task (see Figure 1). As another example, the last code snippet in 2 describes the access to a certain object from the furniture catalog by its name – code which is not used in the solution. But the last line of code shows how a product is actually stored in the delivery object – which is used in the solution.

We distributed the relevant and the irrelevant information in the documentation randomly in order to give subjects not the ability to learn where to look into the documentation (without actually reading it).

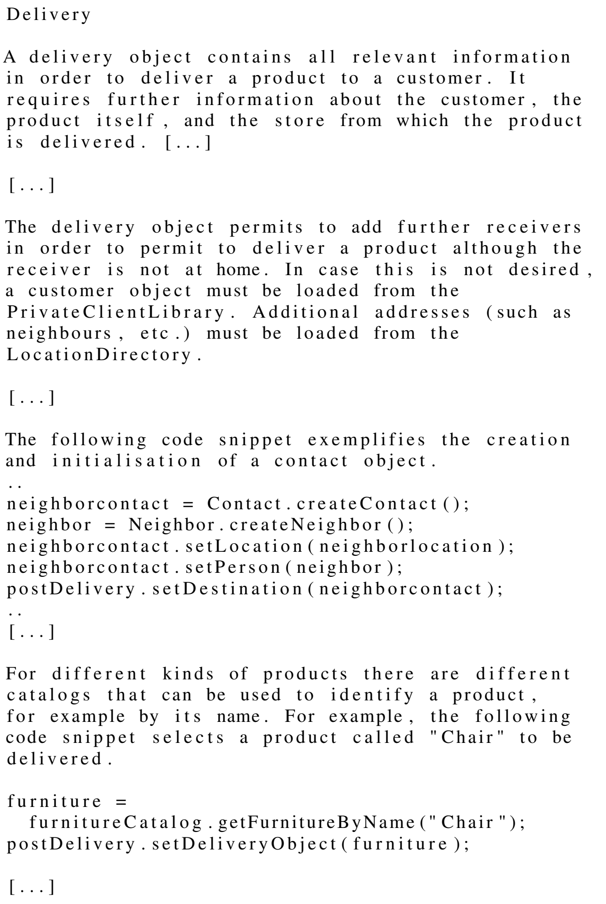
Altogether, we provided six of such documentation files, each with a length between one and six kilobytes.

3.5 Measurements

The experiment performed a number of different measurements, where it should be mentioned that the development time is the main measurement for the experiment.

- **Development Time (DT, seconds)**: The main measurement in the experiment is the development time (DT), i.e. the time

[1] The documentation given to the participants was written in German—this is an abridged translation of the documentation file on delivery



```
Delivery

A delivery object contains all relevant information
in order to deliver a product to a customer. It
requires further information about the customer, the
product itself, and the store from which the product
is delivered. [...]

[...]

The delivery object permits to add further receivers
in order to permit to deliver a product although the
receiver is not at home. In case this is not desired,
a customer object must be loaded from the
PrivateClientLibrary. Additional addresses (such as
neighbours, etc.) must be loaded from the
LocationDirectory.

[...]

The following code snippet exemplifies the creation
and initialisation of a contact object.
..
neighborcontact = Contact.createContact();
neighbor = Neighbor.createNeighbor();
neighborcontact.setLocation(neighborlocation);
neighborcontact.setPerson(neighbor);
postDelivery.setDestination(neighborcontact);
..
[...]

For different kinds of products there are different
catalogs that can be used to identify a product,
for example by its name. For example, the following
code snippet selects a product called "Chair" to be
delivered.

furniture =
  furnitureCatalog.getFurnitureByName("Chair");
postDelivery.setDeliveryObject(furniture);

[...]
```

Figure 2: Small extract from the documentation delivery.txt given to subjects that had access to documentation

required to solve the programming tasks. The development time is measured from the moment when subjects were given the programming task until the moment when the programming task is fulfilled (i.e. until all test cases passed).

- **Number of File switches (FS):** The number of file switches measures how often developers switch between different files during development.[2]

- **Number of documentation switches (DS):** We measured how often developers (those ones to whom documentation has been delivered) switched into the documentation. The number is an indicator for whether documentation has been actually used.

- **Documentation reading time (DRT):** We measured the time that developers (those ones to whom documentation has been delivered) spent on the documentation, i.e. it is the sum of time spans between switching from a documentation file within the IDE. We give this time measurement the name documentation reading time, although this time (probably) additionally includes the time required to find the relevant places in the documentation, to understand the documentation and to think about the relationship between the given documentation the relationship to the given programming problem.

- **Coding time (CT = DT – DRT):** While the development time is the complete time required to solve the programming problem, i.e. including the time required for studying the documentation, the coding time is the development time minus the documentation reading time. The intention of this measurement is to test, whether – under the assumption that it is not necessary to study the documentation at all – the coding time is (finally) much less than the ordinary development time

3.6 Experimental Design

We designed the experiment as a 2x2 randomized between-subject factorial design (see [33]) that analyzes the effect of the independent variables type system (with the treatments static and dynamic type system) and documentation (with the treatments with and without documentation) on the dependent variable development time (respectively coding time). This experimental design consists of four groups, each one represents a combination of a treatment of both independent variables. All subjects are randomly assigned to one of the four groups in a way that finally each group has approximately the same number of participants.

This 2x2 design has the potential problem that for a small sample size, groups may be unbalanced (i.e. good developers are more frequently in one group than in a different group) and there may be a high deviations among subjects (i.e. a large difference between the best and worst developers). These issues may hide the effects in the experiment—in such cases the independent variables have a non-significant effect on the dependent variable. Because of that, our previous experiments consisted only of 2 groups [14, 17, 12]. However, in constrast to our previous studies, this experiment has a single, large programming task, which allows to detect the effects more easily. We conducted a pilot experiment in which we tested the task on some subjects, which convinced us that the main effects are already visible in this setting.

Table 1 shows the four different groups we have in the experiment.

4. EXPERIMENT RESULTS

The experiment has been performed at the University of Duisburg–Essen and the University of Koblenz–Landau. In total, 25 students were used as participants in the analysis (two subjects were removed from the data[3], one subject cancelled the experiment). The data was collected in multiple sessions, depending on student schedules. Students were randomly assigned to one of the four groups in order to reduce the potential for accidental bias, the reasons for which are explained thoughtfully by Vogt (see [31]). Figure 1 describes the names of the groups and the number of subjects. Almost all groups have the same number of subjects (each has six subjects) with the exception of group 1 which has seven. Group 1 has both documentation and static type system, group 2 had no documentation and static typing. The other groups had the same configuration of documentation with dynamic typing.

[2] The motivation for this measurement is that previous experiments that measured differences between static and dynamic type systems showed that the number of file switches reveal the same results as the development time, see section 2.2.

[3] For one subject, the machine crashed which made the experiment data for this subject unreadable, one subject was a non–native german speaker and said, that he did not completely understand the documentation.

	Documentation	
	Yes	No
Static TS	1(7)	2(6)
Dynamic TS	3(6)	4(6)

Table 1: Experimental groups: groupNumber (numberOfSubjects)

4.1 Empirical Data

Table 2 shows the raw anonymized data for the experiment: The first row shows each participant number, the second the experimental group. Continuing rows show development time (DT), file switches (FS), number of switches to documentation (DS), time spent reading documentation (DRT) and coding time (CT).

Three participants (8, 20, and 23) were unable to complete the development task within our time constraints. All of the participants that were unable to complete the tasks happened to be in the group without documentation. Two of these (20 and 23) used the dynamically typed language, while one other (13) was given the statically typed language.

In order to ease the reading of the raw measurements, Figure 3 illustrates the development times via boxplots, which reveals the following possible trends, which will be evaluated more carefully work with standard statistical assessments:

1. The raw median amount of time taken to complete the task appears to increase from groups 1–4.
2. The raw difference between static and dynamic typing, with documentation (groups 1 and 3), appears to be large.
3. The raw difference between static typing without documentation, and dynamic typing with documentation, does not appear to be large.
4. The raw differences between static typing without documentation and dynamic typing either with or without documentation, appear to be rather small, although this is potentially mitigated by the fact that 2 of six participants in group 4 did not complete the task.

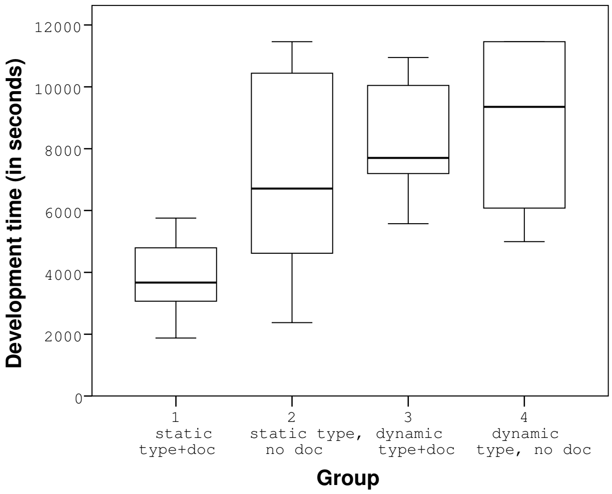


Figure 3: Boxplot for raw development time measurements

4.2 RQ1: Documentation usage

There are many ways in which an analysis of our first research question can be quantified, although which is the “right” one is not obvious. In this experiment, we analyzed the use of documentation by having participants use a single monitor and tracking both switches between code and documentation and time spent on each. From our perspective, there are several other techniques that could potentially be used, for example, eye tracking technology on single or multiple monitors, or even, to give a very different approach, subjective surveys about the user’s experience with the documentation, could potentially draw different kinds of insight into RQ1. While many approaches may have merit, we chose our approach because it is objective and easily trackable.

Given our methodology, we first analyzed, whether the number of file switches to documentation files (for developers to whom documentation was delivered) is significantly larger than zero (no documentation). Table 4 shows the results. In this table, we present the Shapiro–Wilk test, which was not significant (p>.12), i.e. a normal distribution can reasonably be assumed. Thus, we conducted a traditional parametric 1-sample t-Test (a comparison against zero), which was found to be significant (p<.001). The 95% confidence interval is between 54 and 109 with the standard alpha of .05. Performing the same study on time reveals that developers spent between 724 and 1345 seconds (i.e. 12 – 22 minutes) on the documentation and that this difference was also significant (p < .001).

	Shapiro–Wilk	T–Test	95% Confidence Interval	
DS	.12	.001	54	109
DRT	.537	.001	724	1345
DRT/DT	.99	.001	.15	.23

Table 4: Results for research question 1

Another way to assess whether documentation was used is to evaluate how long it was displayed on the screen. We tracked this using the ratio DRT/DT, effectively the time spent on reading documentation divided by the total time spent on developing as a whole. The resulting T-Test reveals that their usage of the API was greater than zero (p < .001). More specifically, the traditional 95% confidence interval indicates developers spent between 15% and 23% of the development time on documentation (DRT / DT).

4.3 RQ2: Development time

The experiment is built as a standard 2x2 between-subject design with two factors, namely type system and documentation. The standard analysis technique for this type of design is a two-factor ANOVA with the aforementioned fixed factors and the dependent variable development time. While this technique is standard, one important assumption of the two-factor ANOVA is the homogeneity of variances, commonly tested with a Levene Test. In our case, the Levene Test was significant (p<.034), implying that our data exhibits heteroscedasticity. In other words, this test confirms what is visually obvious from our boxplots—different experimental groups had different amounts of variance. Given this observation, we double checked our results using non-parametric statistics. Results from these tests confirm our findings and we present only the parametric results here.

Thus, testing the results as a two-factor ANOVA, we find a significant effect for language (p=.007), an effect that approaches significance for documentation (p=.075), and a non-significant effect for the interaction (.211). The overall difference, with 95% confidence, between the static and dynamic typing factors was between

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Group	1	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3	3	4	4	4	4	4	4
DT	3668	3870	3042	3092	1877	5756	5715	11460*	5056	10439	4618	2374	8363	7468	7196	10947	5574	10046	7931	11460*	6078	9025	11460*	9678	4994
FS	253	189	367	170	210	519	487	1120	359	709	396	320	796	672	730	674	505	734	372	920	419	680	834	616	411
DS	35	47	94	41	43	45	30	0	0	0	0	0	0	175	130	132	93	103	91	0	0	0	0	0	0
DRT	553	741	947	781	435	588	338	0	0	0	0	0	0	1788	1295	1901	1214	1360	1534	0	0	0	0	0	0
CT	3115	3129	2095	2311	1442	5168	5377	11460*	5056	10439	4618	2374	8363	5680	5901	9046	4360	8686	6397	11460*	6078	9025	11460*	9678	4994

Table 2: Measured development times (DT, time in seconds), file switches (FS), documentation switches (DS), documentation reading time (DRT), and coding time (CT), * = programming task not finished in time

	DT				FS				DS*				DRT*				CT			
Group	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
MIN	1877	2374	5574	4994	170	320	372	411	30	-	91	-	338	-	1214	-	1442	2374	4360	4994
MAX	5756	11460	10947	11460	519	1120	734	920	94	-	175	-	947	-	1901	-	5377	11460	9046	11460
Avg	3860	7052	8194	8783	314	617	615	647	48	-	121	-	626	-	1515	-	3234	7052	6678	8783
Median	3668	6710	7700	9352	253	553	673	648	43	-	117	-	588	-	1447	-	3115	6710	6149	9352
Std.Dev.	1430	3589	1973	2715	145	315	145	209	21	-	32	-	211	-	278	-	1512	3589	1827	2715

Table 3: Descriptive statistics for raw measurements (groups 2 and 4 do not have values for DS and DRT, because they do not have documentation)

912 and 5353, i.e. between 15 and 89 minutes. In this case, static typing had the advantage. With respect to documentation, given that the effect only approached significance, we urge the reader to use caution in the validity of the result. While we report the result for completeness, we make no direct claims as to whether documentation had a positive impact [4]. With that said, overall, the 95% confidence interval for the difference between the groups with and without documentation was between -391 and 4496. Thus, the group with documentation can arguably be said to have a small advantage. Table 5 shows the results of running a T–Test (again, to double check the two–factor ANOVA with the significant Levene–Test), with comparable results.

	Type System		**Documentation**	
	static	dynamic	with	without
Shapiro–Wilk (p-values)	.08	.45	.63	.17
T-Test (p-values)	.008		.098	
95% Conf. Interval	912	5353	-391	4496
Benefit	statically typed		(*with documentation*)*	

Table 5: Results for research question 2 – the variable documentation is only close to significant (p<.1)

4.4 RQ3: Coding time

In order to evaluate the amount of time each group spent programming, we analyzed the data in a third way, by removing the amount of time the documentation groups spent analyzing documentation. Said in a different way, in our third question, we subtracted out the time spent looking at documentation from the total development time for the documentation group. From this point of view, the group that never received documentation would have the same times.

We perform this analysis to differentiate between two scenarios. First, developers with documentation might browse it, but ultimately spend the same amount of time on programming as the group without documentation. On the other hand, users with documentation may spend less time looking at the code, if the documentation helps them to understand the code more quickly. If the former was the case, it might imply that documentation was only marginally helpful, while if the latter was the case, documentation could be seen as helping developers. Given that our previous result from RQ2 showed that documentation only approached significance, separating the concerns in this way might help us to understand what happened in the experiment.

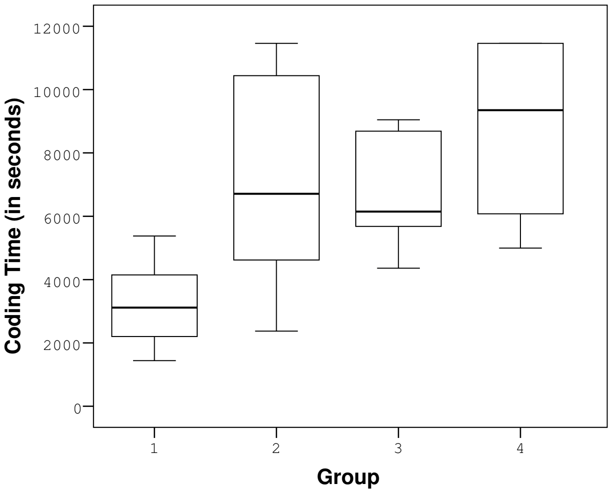


Figure 4: Boxplot for coding time measurements

We reran the experimental analysis with this new view of the data. The Levene-Test is also significant, hence we employ the

[4] On the other hand, discarding a p-value close to, but exceeding the traditional significance level (alpha = 0.05), may result in a Type II error, in this case, assuming that there is no difference when there actually is. We also note that the alpha level is arbitrary; some studies use an alpha level of 0.1, others an alpha level of 0.01.

same procedure as before. In this case, the type system was significant (p=.018), documentation was significant (.008), and the interaction effect continued to be non-significant (.426). For typing, the 95% confidence intervals of the difference between the groups was between 323 and 5082 seconds (5–85 minutes). The static typing group again had the advantage. For the documentation main effect, the 95% confidence interval for the difference was between 810 and 5386 seconds (14–90 minutes), with more time needed for the group without documentation. Figure 6 shows the T-Test that has been applied in order to double–check the two-factor ANOVA, with comparable results.

For a more intuitive representation of the differences, Figure 4 shows the boxplots for the different coding times for each group.

	Type System		**Documentation**	
	static	dynamic	with	without
Shapiro–Wilk (p-values)	.06	.25	.54	.19
T-Test (p-values)	.03		.01	
95% Conf. Interval	323	5082	810	5386
Benefit	statically typed		with documentation	

Table 6: Results for research question 3

5. EXPLORATORY STUDY

Our purpose in this section is to document additional observations we had about our data, by analyzing interactions with the development environment [17]. Given the time necessary to complete the tasks, we elected not to require additional time from the subjects to gather qualitative data; hence we base these observations on the interaction data only. While we hope they help the reader garner more information toward possible causes for our observed effects (e.g., why did documentation help those with static typing?), the reader should interpret this section as a collection of possible clues, not definitive answers.

First, our previous studies indicated that the number of file switches "might" be related to observed differences in productivity and we wanted to recheck this hypothesis here. Thus, we ran another two-factor ANOVA with the number of file switches as the dependent variable, and type system and documentation as independent variables (Table 7, top). Both type system (p = .067) and documentation (p=.064) approach significance, with small effect sizes (η_p^2=.151 and η_p^2=.155, respectively), with the interaction non-significant (p = .129). Table 7 also shows results for the number of file switches, excluding the number of switches to documentation (code switches).

		F	p-value	partial η^2
FS	**Type System**	3.73	.067	.151
	Documentation	3.86	.064	.155
	TS * Doc	2.50	.129	.106
CS	**Type System**	2.316	.143	.099
	Documentation	8.344	.007	.298
	TS * Doc	1.362	.256	.061

Table 7: Univariate ANOVA on file switches (FS) and code switches (CS)

Besides metrics related to switches between files, which remain at least suggestive that this metric might be related to the observed differences with static/dynamic typing with or without documentation, we also analyzed additional time metrics during the sessions. For this, we have the following broad observations:

- **Documentation.** Participants with documentation and static typing (group 1) and dynamic typing (group 3) spent a similar proportion of time reading documentation (medians of 19 and 21%). Both groups used the documentation less in the second half of the session. However, participants in the static group showed a much stronger drop in documentation usage in the second half of the session. This result could indicate that the API *might* be learned more quickly by this group. Whether this learning occurs is unclear, but it could be worthy of more study.

- **Source files.** Participants in all groups spent between 75% and 97% of their time in source files which were relevant to the task. In short, participants quickly realize whether a file is helpful, regardless of group.

- **Test runs.** Similar to our previous experiments, we found that participants using dynamically typed languages conducted more test runs. Documentation did not appear to have an impact.

Again, while we offer these more informal observations to try to give the reader some insight into potential causes, we do so with an air of caution. Like in all experiments, we had experimental controls only for the independent variables discussed in this work. As such, while these results might be suggestive that file switches, and potentially learning effects related to static typing, might be related to our observations, more study is needed to be certain.

6. DISCUSSION

Our experiment comparing the impact of API documentation and static type systems had the following results:

- Subjects who had access to the documentation accessed between 54 and 109 times the documentation and spent there between 12 and 22 minutes (respectively between 15% and 22% of the development time). From that we conclude that the provided documentation was actually used.

- The presence of a static type system had a significant positive effect on development time: Subjects using the statically typed language required between 15 and 89 minutes less time for solving the task.

- The presence of documentation had only a close to significant effect on development time.

- Both, the static type system as well as documentation, reduced pure coding time significantly. The differences between the static and the dynamic type system as well as with and without documentation were comparable (approximately between 10–90 minutes).

A possible explanation of the results pro static type system is that develpers with a static type system (i.e. where the parameter and return types are explicitely declared in the API) have a benefit because they directly see in the code what types are required and requested. Hence, instead of looking into different code fragements in order to find out what kind of data they needs use in the API (which requires additional time), developers are directly guided to the corresponding types. This effect is so strong that it exists independent of whether or not documentation is available. For us the important implication of this is, that previous experiments, which did not take documentation into account, were not falsified —the large benefit of static type systems even existed in the presence of documentation.

The interpretation of the pro documentation effect is slightly harder. First, it looks as if the documentation effect is not that strong as assumed by the experimenters—because the effect on development time is only close to significant. However, as soon as the reading time is subtracted from the development times, the existence of documentation becomes a significant advantage. Hence, it looks like the reading time in the experiment was (still) too large and influenced the experiment results in an undesired way. We believe that future experiments should choose either a larger programming task or a more precise documentation in order to measure the documentation effect.

Additionally, three subjects in the experiment were not able to complete the programming tasks—none of these subjects had documentation available. This could potentially indicate that documentation may increase the correctness of the tasks. In addition, since these subjects were considered to have finished just as they ran out of time, we may have advantaged the subjects without documentation by underestimating their task completion time.

Our exploratory study allowed us to gain additional insights. As in previous experiments, a possible explanation for the differences we see is that both documentation and static typing cause less system browsing, which could result in less time spent.

The second insight found in our exploratory study is that the API documentation tends to be browsed less in the second half of the session. This effect is much more pronounced in static typing users. This might be interpreted as subjects assimilating the API over time. If true, this could indicate that longer task would benefit more from API documentation. Of course, only a partial replication with a longer task would help us ascertain this fact. The experiment of Arisholm [1] had a related finding: their experiment had several task, and they hypothesize that learning effects led to increased performance (in terms of correctness) in the last task.

7. THREATS TO VALIDITY

The overarching results aside, like any scientific experiment, it is not possible for any one study to fully flesh out all potential confounding variables in regards to a particular research topic. At best, empirical work provides a snapshot of carefully controlled variables which, if done well, can help other researchers find answers to deeply difficult questions over time. Our study is no exception and we focus here on what we felt were the most likely validity threats.

Type of documentation. First, the quality and shape of software documentation can vary considerably. In the study of Roehm *et al.* [26], several industrial developers do not trust documentation and prefer to rely on source code. Dart's home page (https://www.dartlang.org) acknowledges the incompleteness of the documentation and points to Stack Overflow. Languages like Java and .NET have a much more extensive coverage, although one study by Maalej and Robillard showed that some API documentation contains bloat and is of questionable value (e.g., rehashes method names, includes trivial examples) [16].

In our case, we wrote correct documentation that was equivalent across groups, was grammatically understandable (in German), and that covered what we felt were the most important points in regards to learning an API. Our documentation contained examples and scenarios that we thought would be useful. In other words, we set out to write what seemed to us to be "good" documentation. We would be unsurprised if systems that have misleading, incorrect, or, more abstractly, poor documentation, might not benefit programmers. For example, a pilot study by Leotta et al. suggests that out of date, or as they authors say, misaligned, UML documents can impact programmer productivity [15].

Type of type system. We have chosen Dart's type system in the experiment which has a number of parallels to Java's type system: the types are declared in the code and identified by type names. It might be possible and likely that the measured effect for different kinds of type systems is different.

Usefulness of documentation. Besides the quality of the documentation, we must also consider whether the participants found it useful. We tracked carefully whether participants switched to documentation and have provided evidence showing as such in RQ1. However, we can not assess whether the documentation was useful, correctly used, or comprehended, beyond that.

Single task. Our choice of a single task limits the generalizability of our results. Previous results have shown that not all tasks lend themselves well to static typing, for example [14]. Given this, we would be unsurprised if different types of tasks impacted programmers differently. In this experiment, we chose a single task that previous studies show lends itself to static typing and evaluated the impact once documentation was introduced. As the showed, at least for this task, the effect was strengthened.

Choice of experimental design. In previous studies, we tracked participants over a set of tasks, conducting what is commonly termed a repeated measures experiment [14]. In such experiments, participants complete a set of tasks repeatedly, which likely leads to learning over time. Such studies have pros and cons, namely in that they provide a view of how a system can be learned (see e.g., Arisholm *et al.* for an example with UML [1]), but may not capture potential confounds related to *how* a system is learned. In contrast, single tasks provide a snapshot of developers as they use a tool once, but may not provide much insight into how a system can be used with continuous usage. Such pros and cons are well known in a variety of academic disciplines and are discussed thoughtfully by Vogt [31].

The role of expertise. Like previous experiments, we used students late in the academic pipeline for our experiment. While one study suggests that this class of user might be at least representative of professionals [13], recent research has shown, with some confidence, that novices have difficulty with type annotations in statically typed languages [8, 27]. Likewise, experts in a certain dynamic languages may not be as strongly affected by the lack of static typing or documentation, although to our knowledge this has not been studied formally. As such, the reader should not assume that our findings generalize to all levels of expertise.

Choice of environments. The fact that we restricted our participants to a simple IDE may not reflect the state of the practice. Eclipse allows very quick navigation between API elements and their documentation (including hovering to get documentation [7]), for JavaDoc-style documentation. However including a state-of-the art IDE adds a variety of confounding factors related to the expertise of subject with tools. Some subjects may not be aware of the existence of tools, usage data show that even highly-praised tools such as Eclipse's "open resource" tool are not used by 88% of developers [18].

Another issue is that we evaluated participants as they used a single monitor, flipping between documentation and computer code. We ultimately chose to use a single monitor because many individuals do program in this manner (e.g., on a laptop). If the experiment were re-run with dual monitors, participants might have had documentation on one monitor and code on the other, potentially minimizing time to flip between them.

8. CONCLUSION

API usability is an active field of research. There is evidence in the literature of two factors that contribute to it: the presence of documentation, and the presence of type annotations. We conducted a study comparing the impact of documentation and static type systems on API usability as measured by development time. The study had two components: a 2x2 randomized controlled trial, and an exploratory study of the subjects' interaction logs.

Overall, we found that both static type systems and documentation had a positive effect; however, the effect of static type systems was stronger, as it was not balanced by the need for additional reading that documentation exhibits. As a result, the effect of documentation was only close to significant.

However our exploratory study found interesting findings related to documentation: it appears that subjects use documentation more heavily at the start of their tasks, which means that the effect of documentation may be larger in longer tasks. Further experiments are needed to validate this tentative hypothesis.

Overall our results confirm and expand on previous findings. It appears that adding documentation to statically typed systems strengthens the effects we previously observed [17]. In a sense, static typing appears to be even more effective when documentation is added, as both helps appear to reinforce each other.

9. ACKOWLEDGEMENTS

We would like to thank the volenteers from the University of Duisburg–Essen and from the University of Koblenz–Landau for participating in the experiment. Last but not least, we especially thank Ralf Lämmel from the University of Koblenz–Landau for his kind support.

10. REFERENCES

[1] E. Arisholm, L. C. Briand, S. E. Hove, and Y. Labiche. The impact of uml documentation on software maintenance: An experimental evaluation. *IEEE Trans. Software Eng.*, 32(6):365–381, 2006.

[2] R. K. Bandi, V. K. Vaishnavi, and D. E. Turk. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Trans. Software Eng.*, 29(1):77–87, 2003.

[3] F. P. Brooks, Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[4] K. B. Bruce. *Foundations of object-oriented languages: types and semantics*. MIT Press, Cambridge, MA, USA, 2002.

[5] B. Dagenais and M. P. Robillard. Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of SIGSOFT FSE 2010*, pages 127–136, 2010.

[6] U. Dekel and J. D. Herbsleb. Improving api documentation usability with knowledge pushing. In *Proceedings of ICSE 2009*, pages 320–330, 2009.

[7] U. Dekel and J. D. Herbsleb. Reading the documentation of invoked api functions in program comprehension. In *Proceedings of ICPC 2009*, pages 168–177, 2009.

[8] P. Denny, A. Luxton-Reilly, and E. D. Tempero. All syntax errors are not equal. In *Proceedings of ITiCSE 2012*, pages 75–80, 2012.

[9] E. Duala-Ekoko and M. P. Robillard. Asking and answering questions about unfamiliar apis: An exploratory study. In *Proceedings of ICSE 2012*, pages 266–276, 2012.

[10] B. Ellis, J. Stylos, and B. A. Myers. The factory pattern in api design: A usability evaluation. In *Proceedings of ICSE 2007*, pages 302–312, 2007.

[11] J. D. Gannon. An experimental evaluation of data type conventions. *Commun. ACM*, 20(8):584–595, 1977.

[12] M. Hoppe and S. Hanenberg. Do developers benefit from generic types? an empirical comparison of generic and raw types in java. In *Proceedings of OOPSLA 2013*, page to appear, 2013.

[13] M. Höst, B. Regnell, and C. Wohlin. Using students as subjects-a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3):201–214, 2000.

[14] S. Kleinschmager, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. Do static type systems improve the maintainability of software systems? an empirical study. In *Proceedings of ICPC 2012*, pages 153–162, 2012.

[15] M. Leotta, F. Ricca, G. Antoniol, V. Garousi, J. Zhi, and G. Ruhe. A pilot experiment to quantify the effect of documentation accuracy on maintenance tasks. In *Proceedings of ICSM 2013*, page to appear, 2013.

[16] W. Maalej and M. P. Robillard. Patterns of knowledge in api reference documentation. *IEEE Trans. Software Eng.*, 39(9):1264–1282, 2013.

[17] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of OOPSLA 2012*, pages 683–702, 2012.

[18] E. R. Murphy-Hill, R. Jiresal, and G. C. Murphy. Improving software developers' fluency by recommending development environment commands. In *Proceedings of SIGSOFT FSE 2012*, page 42, 2012.

[19] V. Pankratius, F. Schmidt, and G. Garreton. Combining functional and imperative programming for multicore software: An empirical study evaluating scala and java. In *Proceedings of ICSE 2012*, pages 123–133, 2012.

[20] C. Parnin, C. Treude, L. Grammel, and M.-A. Storey. Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. Technical Report GIT-CS-12-05, Georgia Institute of Technology, May 2012.

[21] M. Petre. Uml in practice. In *Proceedings of ICSE 2013*, pages 722–731, 2013.

[22] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[23] L. Prechelt and W. F. Tichy. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Trans. Softw. Eng.*, 24(4):302–312, 1998.

[24] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Trans. Software Eng.*, 28(6):595–606, 2002.

[25] M. P. Robillard and R. DeLine. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[26] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of ICSE 2012*, pages 255–265, 2012.

[27] A. Stefik and S. Siebert. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (To Appear)*, 2013.

[28] T. Tenny. Program readability: Procedures versus comments. *IEEE Trans. Software Eng.*, 14(9):1271–1279, 1988.

[29] W. F. Tichy. Hints for reviewing empirical work in software engineering. *Empirical Software Engineering*, 5(4):309–312,

2000.

[30] E. Tryggeseth. Report from an experiment: Impact of documentation on maintenance. *Empirical Software Engineering*, 2(2):201–207, 1997.

[31] W. P. Vogt. *Quantitative Research Methods for Professionals in Education and Other Fields*. Allyn and Bacon, 2006.

[32] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: a controlled experiment. In *Proceedings of ICSE 2011*, pages 551–560, 2011.

[33] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell. *Experimentation in Software Engineering*. Springer, 2012.