Static Type Systems (Sometimes) have a Positive Impact on the Usability of Undocumented Software: An Empirical Evaluation

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

Romain Robbes, Éric Tanter
University of Chile,
Computer Science Department,
Santiago de Chile, Chile
rrobbes@dcc.uchile.cl
etanter@dcc.uchile.cl

Andreas Stefik
Southern Illinois University
Edwardsville,
Department of Computer Science,
Edwardsville, IL
astefik@siue.edu

Abstract

Static and dynamic type systems (as well as more recently gradual type systems) are an important research topic in programming language design. Although the study of such systems plays a major role in research, relatively little is known about the impact of type systems on software development. Perhaps one of the more common arguments for static type systems is that they require developers to annotate their code with type names, which is thus claimed to improve the documentation of software. In contrast, one common argument against static type systems is that they decrease flexibility, which may make them harder to use. While positions such as these, both for and against static type systems, have been documented in the literature, there is little rigorous empirical evidence for or against either position. In this paper, we introduce a controlled experiment where 27 subjects performed programming tasks on an undocumented API with a static type system (which required type annotations) as well as a dynamic type system (which does not). Our results show that for some types of tasks, programmers were afforded faster task completion times using a static type system, while for others, the opposite held. In this work, we document the empirical evidence that led us to this conclusion and conduct an exploratory study to try and theorize why.

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

For decades, type systems (cf. [4, 18]) have played an essential role in software education, research, and industry. While a large number of programming languages with industrial relevance include a static type system (e.g., Java, C++, Haskell, or Scala), a number of programming languages—especially those used in web technologies—include a dynamic type system (such as Ruby, PHP, JavaScript, or Smalltalk). Thus, a lingering question is whether one system or another, either static or dynamic, has a larger benefit for the humans that use them.

Although there is an ongoing debate about the pros and cons of static type systems where different authors strongly argue for or against static type systems, it is rare that such arguments are backed by empirical observations. In contrast, such claims are often backed by personal experience, speculation, or anecdote.

To help summarize the debate both for and against static type systems, typical arguments against include reasoning such as:

- **Type systems are inflexible:** not all valid programs can be handled by a given type systems. As such programmers often use workarounds, such as type casts, to bypass the limitations of the type system.
- For the same reason, **type systems impede rapid prototyping**, where, for instance, an object passed as an argument may not comply with a given type, but will work in practice if it implements an acceptable subset of the protocol it should respond to. Inserting type annotations and making objects conform to the expected type distract the programmer from the task at hand.

On the other hand, typical arguments in favor of static type systems often include:

- **Type systems improve the program structure.** Type systems help in the design of clear and well-structured programs. Logical errors can be detected by any compiler that enforces a type system. (see [1, p. 8]).

- **Type systems act as a form of documentation.** Quoting Pierce: "A static type system provides the reader of code with an implicit documentation. Because a static type system enforces type declarations for variables, methods parameters and return types, it implicitly increases the documentation factor by making the code speak for itself." [18, p. 5]

Interestingly, more recent programming languages such as Dart[1] try to avoid this question by providing a gradual type system (see [25]) which permits developers to use static and dynamic typing in combination. While this approach appears to be a compromise at first glance, such a technique essentially pushes the decision onto developers themselves. While developers using a gradual type system have the freedom to choose the type system, they (still) suffer from the problem of determining in what situations each type system is appropriate.

In order to address the question of pros and cons of static and dynamic type systems, this paper focusses on the second of the above mentioned arguments—the static type system's potential benefit in documentation. We present an empirical study on 27 human subjects that we asked to perform five different programming tasks on an API which was not initially known to the subjects and which was documented only via its source code (without any additional documentation such as comments, programming examples, handbooks, etc.). While the expected outcome of this experiment was that the static type system has, at least, no negative impact on the usability of such APIs, our empirical observations revealed that for three of the five tasks, static type systems had a positive impact on development time–while for two programming tasks, we found that dynamic type systems afforded faster development times.

We start this paper with a discussion of related work in section 2. Then, section 3 describes the experimental design, the programming tasks given to the subjects, the execution of the experiment, and the threats to its validity. Section 4 describes the results of the experiment. As the results we observed did not conform to our initial expectations, we performed exploratory studies that try to explain the result of the experiment using further measurements (section 5). We discuss this work in section 6. Finally, we conclude the paper in section 7 and include additional empirical data in an appendix.

2. Related Work

In an experiment in the late 1970s, Gannon was able to show a positive impact of static type systems [8]. Thirty-eight subjects participated in a two-group experiment where each group had to solve a programming task twice, with a statically type checked language and a language without type checking. The study reveled an increase in programming reliability for subjects using the language with the static type checking. The programming languages used were artificially designed for the experiment.

By using the programming languages ANSI C and K&R C, Prechelt and Tichy studied the impact of static type checking on procedure arguments [21]; ANSI C is a language which performs type checking on arguments of procedure calls while K&R C does not. The experiment divided 34 subjects into four groups. Each group had to solve two programming tasks, one using ANSI C and one using K&R C. The groups differed with respect to the ordering of the tasks and with respect to what task had to be fulfilled with what technique. While the experiment tested several hypotheses with slightly mixed results, it measured with respect to development time for one of the two programming tasks a significant impact of the programming language in use—for one task subjects using the statically type checked ANSI C were faster in solving the programming task. For the other programming task, no significant difference was measured.

In a qualitative pilot-study on static type systems Daly et al. observed programmers who used a new type system for an existing language (a statically typed Ruby [6]). The authors concluded from their work that the benefits of static typing could not be shown, however, the study did not include quantitative analysis, making it difficult for other research teams to replicate their findings at other institutions.

Another study, which did not directly focus on static type systems compared seven different programming languages [19]. Although the focus was not explicitly on static or dynamic type systems, one finding was that programs written in scripting languages (Perl, Python, Rexx, or Tcl), took half— or less—time to write than equivalent programs written in C, C++, or Java. However, it should be emphasized that this experiment had considerable methodological flaws, including: 1) the development times for the scripting languages —as explicitly emphasized by the author— were not directly measured on subjects (which was the case for the languages C, C++, and Java), 2) the subjects were permitted to use tools of their choice (e.g., IDE, testing tools), and 3) the programs subjects wrote were not of equivalent length. As such, that study was not a controlled experiment and even it the results are ultimately correct is difficult to replicate. Despite the potential methodological issues, Prechelt concluded that humans writing in programming languages with a dynamic type system (the scripting languages) were afforded faster development times.

Our studies. The study presented here is part of a larger experiment series that analyzes the impact of static type systems on software development (see [12]). We performed four other experiments ([10, 15, 26, 27]).[2]

The study by Hanenberg studied the impact of statically and dynamically typed programming languages to implement two programming tasks within approximately 27 hours

[1] See http://www.dartlang.org/

[2] The work by Steinberg and Hanenberg is not yet published [26].

[10]. Forty-nine students were divided in two groups, one solving programming tasks using a statically typed language and the other a dynamically typed language. Subjects using the dynamically typed programming language had a significant positive time benefit for a smaller task, while no significant difference could be measured for a larger task. However, the experiment did not reveal a possible explanation for the benefit of the dynamically typed group for the smaller task. The object-oriented programming languages being used were artificially designed for the experiment.

In a different experiment, Stuchlik and Hanenberg analyzed to what extent type casts, which are a language feature required by languages with a static type system, influenced the development of simple programming tasks [27]. Twenty-one subjects divided into two groups, took part in a within-subjects design—participants complete two tasks, once with static typing and once with dynamic typing. It turned out that type casts do influence the development time of rather trivial programming tasks in a negative way, while code longer than eleven lines of code showed no significant difference. The programming languages being used were Java (for the statically typed language) and Groovy (where Groovy was only used as a dynamically typed Java without using Groovy's additional language features).

A further experiment performed 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 [26]. The study was based on 31 subjects (again based on a two-group within-subject design). The result of the experiment was that static type systems have a positive impact on the time required to fix type errors (in comparison to the time for fixing equivalent no-such-method exceptions). With respect to the time required to fix semantic errors, the experiment did not reveal any significant differences between the statically and dynamically typed languages. Again, the programming languages Java and Groovy were used.

Finally, Kleinschmager et al. performed an experiment on 33 subjects which combined repetitions of previous experiments (with Java and Groovy as languages) [15]. Among other tested hypotheses, it was confirmed that fixing type errors in a statically typed language is faster than corresponding fixes in a dynamically typed language, while no differences for fixing semantic errors was measured. As we see from our literature review, there is not yet an empirically rigorous consensus amongst the few studies we are aware of. As such, further experiments—like this one—are needed.

3. Experiment Description

We begin by stating the underlying hypotheses of our experiment and then discuss initial considerations when running such studies. After introducing issues such as which programming languages, programming environments, and APIs we used, we describe our programming tasks in detail. Then, we state our expectations about the performance of the developers in the two groups, the experimental design, and why we chose that design. Finally, as all experiments have limitations, we discuss the threats to validity.

3.1 Hypotheses

Our experiment tests the belief that a static type system helps developers use an undocumented API. By "helping", we mean that a static type system either requires less effort in comparison to a dynamic type system in order to fulfill the same programming tasks, or, with the same effort, that more programming tasks can be fulfilled with the aid of static type systems. The second perspective differs from the first one in that the effort is fixed, while it is variable in the first one. In our experiment, we decided to use development time as a variable, as it is easier to measure the time required to fulfill a certain task than determining how much work has been accomplished within a given amount of time; Development time is a common measurement for effort in pure programming experiments (see [8, 13, 19–21, 24, 28]).

Hence, the first hypothesis to be tested is:

- **Null Hypothesis 1**: The development time for completing a programming tasks in an undocumented API is equivalent when using either a static type system or a dynamic type system.

Given that our first null hypothesis only takes into account the design of an undocumented API as a whole, it is desirable to formulate a second null hypothesis that potential confounding factors into account. For example, it seems reasonable that given a larger undocumented API, static type information may help the user more than a similar task using a smaller API, where using the API may be more obvious because there are less classes or methods[3].

Accordingly, we formulate a second null hypothesis as follows:

- **Null Hypothesis 2**: There is no difference in respect to development time between static and dynamic type systems, despite the number and complexity of type declarations in an undocumented API.

Note that the second hypothesis only takes a different number of classes and types into account but does not try to explain the relationship between the number of types and development time. Finally, both hypotheses focus on the development time of programming tasks. Hence, if either of these hypotheses can be rejected through standard statistical inference techniques, we may gain insight into the relative benefits or consequences of static and dynamic typing.

[3] According to programming languages such as Java or C++ we assume a close connection between class hierarchy and type hierarchy. Furthermore, we assume here that the dynamically typed API does not contain any type annotations.

3.2 Initial considerations for the experiment

In order to build an experiment that tests the previously defined hypotheses, it was necessary to find:

- Two comparable languages (with a static and a dynamic type system);
- An undocumented piece of software that should be used within the experiment;
- Appropriate programming tasks that should be fulfilled by the subjects; and
- A reasonable experimental design.

At the time the experiment was planned, we knew that volunteers from the University of Duisburg-Essen, Germany, and the University of Chile would participate as subjects in the experiment. In order to obtain participation from these volunteers, we agreed that that the experiment should last for no more than one day. Further, we estimated upfront that the number of subjects would be around thirty. Hence, the experimental setup and the experimental design should take into account that the expected effect size should be measurable for a rather small sample size.

3.3 Programming languages and environment

The broad goal for our experiment was to compare two different languages—one with a static type system (which requires type annotations in the code) and one with a dynamic type system (without any type annotations). In order to address the problem that different languages have different language features and that consequently differences in an experiment cannot be reduced to the factor type system, it is necessary to find two very similar languages.

According to previous experiments (see [26, 27]) we decided to use Java and Groovy [16]. While Groovy has a number of language features in addition to Java, it can also be used as a "dynamically typed Java": all type declarations of variables, fields and parameters can be annotated with the keyword `def` (without referring to the corresponding nominal type). In this case, Groovy does not perform any static type checking on these elements but dynamically performs the type check at runtime. Consequently, a pure reduction of Groovy on the language features of Java permitted us to have two similar languages which only differ with respect to the type system.

One further argument for using Java and Groovy was that the subjects already had some programming skills in Java. Consequently, it was not necessary to perform any exhaustive training for the subjects. This implies that we did not intent to introduce Groovy as a new programming language. Instead, we only introduced Groovy as "a Java version where the declarations of variables, parameters, etc. only required the keyword `def`."

To maximize the similarity of the two experimental settings, it was necessary to exclude another language feature: *Method overloading*. When the overloaded and overloading methods have the same number of parameters, parameter types must be explicit in order to distinguish between the methods. Hence, if no static types are available, method overloading cannot be used. An alternative would have been to use in the dynamically typed version methods with different names, where the caller of the method is responsible for choosing the right one. However, it was unclear whether this alternative would introduce other experimental confounds, as the two APIs would be different. Thus we decided to ignore method overloading in the experiment.

For the programming environment used by the subjects, our intention was (again) to provide a comparable situation for both Java and Groovy. While at first glance IDEs such as Eclipse provide tool support for Java as well as Groovy, Java tool support in Eclipse is much more mature than that of Groovy. Consequently, using Eclipse would probably have been an advantage for Java developers and would have confounded the measurements: the intention of the experiment was to measure the impact of the language feature type system, not the maturity level of tool support. Hence, we decided to use a plain text editor, as has been done in previous experiments [15, 26]). This custom editor permitted the user to compile programs, run applications, and execute test cases. This simple IDE also logged data for the experiment such as when subjects were about to run their code, or when they fulfilled a given programming task.

3.4 Experimental design—two groups within-subject design

The experiment was designed as a 2-group within-subject design (see [13, 20, 28] for a more detailed introduction into such kinds of experiments) where the subjects were randomly assigned to one of both groups. The groups should be balanced with respect to the number of subjects, i.e. both groups should have a comparable number of subjects. Each group solved the programming tasks in both languages, i.e. in Java as well as in Groovy. Both groups differ with respect to the ordering of both languages: While the first group (group A) solved all programming tasks in Groovy and then solved all programming tasks in Java, the other group (group B) did it the other way around. The main motivation for this kind of design is, that the problem of deviation among different subjects is reduced (see section A.1 for a more detailed discussion).

Obviously, learning effects are a potential problem in experimental designs that have subjects complete more than one task. Once a programming task is solved using language A, it becomes easier to solve the same programming task with language B (if both languages are similar). Consequently, it seems obvious—at first—that a within-subject measurement, where each subject solves a programming task twice cannot lead to valid results. However, apart from the fact that this experimental design has been already successfully applied (see for example [27]), there are several

valid statistical approaches for fairly analyzing data in such studies. Here we just roughly discuss the design – a more detailed description and discussion can be found in the appendix (section A.1).

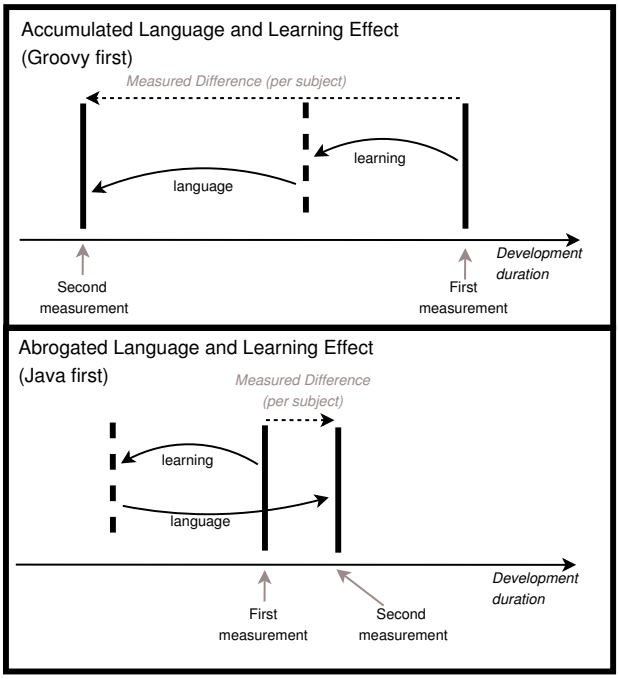

Figure 1. Impact of within-subject design on experiment results (taken with adaptations from [27])

Figure 1 illustrates the assumed effect on both groups – the first measurement and then the second measurement which includes the learning effect as well as the language effect. As the figure shows, the second measurement is still able to reveal the effect of the language, if the learning effect is smaller than the language effect (in the lower part of Figure 1 the second measurement is still larger then the first measurement). It is even acceptable if the learning effect is similar to the language effect (which means that no difference between the first and the second measurement is found for group B)– but the design fails, if the learning effect is larger than the language effect (see section A.1 for a more detailed discussion).

3.5 Description of the APIs

API Design Considerations. Our aim was to provide two versions of an undocumented API—one Java version and one Groovy version—as similar as possible. The only difference should be that the pure Java version makes use of static types while the Groovy version does not. Similarly, we the API should be simple enough to understand how to use it in a relatively small amount of time. However, the API must not be so trivial that all subjects immediately finish the tasks (a ceiling effect). Since the time difference between static and dynamic type systems potentially depends on the searching for the right types or reading source code in different files, the source code should be large enough that a difference could be measured.

Finally, we needed to take the previously described within-subject design into account. Since this design potentially suffers from learning effects, it is desirable to consider some precautions against it. For example, we would generally expect that subjects using the API a second time would be better at doing so. As such, our statistical procedures must take this into account by conducting comparisons across first time users of both static and dynamic typing, in addition to secondary use.

Structurally identical APIs. With these factors in mind, we provided two structurally identical APIs, where the second one derived from the first by renaming classes, methods, variables, etc. This was done in a manner that ensured the two APIs, despite having the same complexity, seemed to address two entirely different problems. We designed an API consisting of 35 classes, in four packages, having 1279 lines of code. One version of the API handled the implementation of UI tree views, the other version handled SQL queries (as trees of SQL elements) and results. The tree view API was built first and from it we derived the SQL API.

3.6 Description of programming tasks

The programming tasks should directly reflect on the hypotheses as introduced in section 3.1. In order to do so, we decided to design programming tasks that differ with respect to whether the API is statically or dynamically typed (hypothesis 1). They also should differ with respect to the complexity of type declarations needed in order to fulfill the programming tasks (hypothesis 2).

Task complexity. Since we are interested in the effect of typing on APIs, we wish to measure the difference with respect to typing and not the possible influence of type systems on different language constructs, or on the complexity of the code to write. While complexity measurements such as lines of code, Halstead, or cyclomatic complexity are often used [7], we decided to use a different approach. In order to maximize the effect of using the API, and reduce the effect of the complexity of the programming task itself, we kept the complexity of the code snippets subjects have to write are similar. The programs that use the APIs should only consists of single methods which do not make use of non-trivial constructs such as loops or conditions. Instead, the intention was to use only assignments, variable reads, method calls and object creation operators. The complexity in the tasks only stems from the number of types to identify and the relationships between the tasks.

Task completion. We instructed the subjects to move to the next task only when they successfully completed the current task. For each task, we defined several tests that verify that the task has been indeed completed. These were Unit Tests run, run each time the code snippet is compiled successfully. Only when all the tests passed were the subjects

allowed to go to the next task. The subjects did not have access to the tests.

Warm-up task. We included a simple warm-up task so that the subjects would have time to familiarize themselves with the development environment, the testing process to determine correctness, and the Groovy programming language. We asked that subjects solve three vary basic programming tasks (e.g., summing a list of numbers). This task was not included in the analysis and is not described below.

Task description. In the following section, we describe the programming tasks from the following perspectives: (1) what the expected solution of the task is; (2) what the characteristics of the programming tasks are; and (3) how many unique types are required in order to fulfill the programming tasks. We show the solution from the first API—UI tree views—the subsequent ones have similar solutions.

3.6.1 Task 1 (easy, one class to identify)

In the first programming task, developers were asked to return an instance of a certain class (`AbstractTreeFactory`). In order to do so, they were told the name of the abstract superclass, but not the name of the concrete subclass (where only one such concrete subclass exists in the project).

Hence, on both cases (dynamically and statically typed) it was necessary to identify one type (which corresponds to only one class `AbstractTreeViewFactory`) in the project. The code consists of a single line, which returns the object being created (see Figure 2).

3.6.2 Task 2 (easy, three classes to identify)

In the second programming task, developers were required to create an initialized `TreeView` object with the name `sampletree`. This object is created by invoking a method in the previously described factory. Additionally, an initializer needed to be passed to the factory. This initializer itself required a `Configuration` object which contains the title of the tree (an instance of class `Title`); altogether, subjects had to identify `Initializer`, `Configuration` and `Title`.

```
public static AbstractTreeViewFactory task1() {
  return new TreeViewFactory();
}

public static TreeView task2() {
  AbstractTreeViewFactory stv;
  Initializer init = new Initializer();
  Configuration conf = new Configuration();
  Title t = new Title();
  t.setName("sampletree");
  conf.setTitle(t);
  init.setConfig(conf);
  stv = new TreeViewFactory(init);
  return stv.createTree();
}
```

Figure 2. Example solutions for programming tasks 1 & 2

3.6.3 Task 3 (medium, three classes to identify)

The third task required developers to create a transformer of the tree view. A corresponding class `Transformer` was given within the API which performs such a transformation. However, it was necessary to create first a graph from the tree view and pass it to the transformer. Then, a walker must be created and passed to the transformer. Finally, a start node for the transformation (the tree's root node) is provided; it needs to be extracted by converting it into a `TreeViewNode` (by sending a corresponding message to the graph object). Figure 3 illustrates a possible solution for the task. Altogether, the types `Graph`, `Walker` and `TreeViewNode` needed to be identified.

3.6.4 Task 4 (difficult, three classes to identify)

The fourth task required developers to add a new node (with name `sampletreenode`) to a graph which can be accomplished by parameterizing an initializer correctly. The problem with this task is that the design of the initializer object is not trivial. The initializer must be parameterized with instances of an `IdentifiedSequence`. This sequence contains a tree node identifier (the node's name) and a sequence of pairs consisting of an identifier and an `IdentifiedSequence`—each child of a tree node itself is identified by a corresponding identifier. In that way, it is possible to parameterize the initializer already with a tree of identifiers.

Again, three classes need to be identified: `IdentifiedSequence`, `TreeNodeIdentifier`, and `Pair`. Due to the underlying recursive definition of the items, we suspected this code to be rather difficult to understand (if no static type system directly reveals the underlying design of the API).

3.6.5 Task 5 (easy, six classes to identify)

For the final task, a menu with a corresponding command should be created for the tree view. A command, represented by a String, is passed to the method. It should be added to a corresponding `Command` object (which needs to be created). The command object needs to be passed to the menu. Further objects (`LayoutPolicy`, `BackgroundImage`, `Font`) should also be passed to the menu object in order to fully specify the command.

The task required six classes to be identified: `Menu`, `MenuItem`, `SingleCommand`, `LayoutPolicy`, `BackgroundImage`, `Font`. We suspected that this task might be simpler than others, since the relations between the types is much more straightforward than the preceding task.

3.7 Assumed Disadvantage of Dynamic Type Systems

The programming tasks were designed in a way that we assumed that for all tasks (except task 1) the static type system would show a measurable positive impact; we explain the assumed reasons for this and the assumed behavior of developers in the following. We use "DT developers" to describe

```
public static Transformer<TreeViewNode>
                    task3(TreeView tv) {
  GraphFactory gf = new GraphFactory();
  Graph <TreeViewNode,Edge<TreeViewNode>> g =
    gf.createGraphFromTreeView(tv);
  Transformer<TreeViewNode> t =
    new Transformer<TreeViewNode>();
  t.setTransformableStructure(g);

  Walker w = new Walker();
  t.setWalker(w);
  TreeViewNode s =
    g.getNodeFromContent(tv.getRootNode());
  t.setStartNode(s);
  t.doTransformation();
  return t;
}

public static void task4(Initializer init) {
  IdentifiedSequence<TreeNodeIdentifier> z =
    new IdentifiedSequence<TreeNodeIdentifier>();
  TreeNodeIdentifier t = new TreeNodeIdentifier();
  t.setName("samplerootnode");

  Pair <TreeNodeIdentifier,
    IdentifiedSequence<
      TreeNodeIdentifier>> p =
    new Pair <TreeNodeIdentifier,
    IdentifiedSequence<TreeNodeIdentifier>>();

  p.setFirst(t);
  p.setSecond(new IdentifiedSequence<
      TreeNodeIdentifier>());
  z.add(p);
  init.setItems(z);
}
```

Figure 3. Example solutions for programming tasks 3 & 4

```
public static Menu task5(String cmd) {
  Menu m = new Menu();
  MenuItem mi = new MenuItem();
  mi.setTitle("samplecommand");
  m.add(mi);
  SingleCommand cc = new SingleCommand();
  mi.setCmd(cc);
  cc.setCommand(cmd);
  LayoutPolicy l = new LayoutPolicy();
  BackgroundImage bg = new BackgroundImage();
  Font f = new Font("Arial", 12, Font.Style.DEFAULT);
  m.setLayout(l);
  l.setBgImage(bg);
  l.setFont(f);
  return m;
}
```

Figure 4. Example solution for programming tasks 5

those developers that use the dynamically typed API, and "ST developers" for those using the statically typed API.

3.7.1 Task 1

For the first task, DT developers as well as ST developers have the same problem: they need to become aware that the class mentioned in the task description is abstract. In both cases, developers need to scan the code in order to find a concrete class that implements the mentioned one. As such, we expected no significant difference between the groups.

3.7.2 Task 2

For programming task 2 we suppose that DT developers have a disadvantage in comparison to ST developers. We assumed that developers first will have to investigate the class `TreeViewFactory` in order to determine how a new tree view can be created. DT developers then find a parameter named `init` in the constructor (which is then passed to an instance variable named `init`). DT developers should need additional time in order to determine that an `Initializer` instance is required here. Then, the type `Initializer` needs to be understood. While the ST developers directly can see in the code that a type `Configuration` is required, DT developers only see that there is a field named `config`. Again, DT developers might assume that the tree name should be assigned to `config`. Finally, while ST developers directly get the information that a `Configuration` object requires a `Title` object (which contains the `String` object representing the tree's name), DT developers need to find it out on their own—perhaps assuming that they can directly pass the `String` to the config object.

3.7.3 Task 3

Task 3 is slightly harder to solve than task 2. The main difference lies in the way the API can be read. In task 2, it was possible to study a class in order to determine what needs to be done next. In task 3, DT developers only know that they need a `Transformer` object, but no `TreeView` is permitted as parameter. Here, developers need to detect that a different class `GraphFactory` converts the tree view into a graph which then can be passed to the transformer. The same is true for the start node, which needs to be returned from the graph in order to be passed to the transformer. We assume that the relationships between these different tasks are harder to detect for DT developers, advantaging ST developers.

3.7.4 Task 4

Task four is the hardest task for the DT developers. We think that developers easily become aware that the items need to be set in the `Initializer` object. For the ST developers, the method directly reveals the required (generic) type which guides developers to type `IdentifiedSequence`. From the types, ST developers see that a `TreeNodeIdentifier` is required and that an object of type `Pair` needs to be added to the sequence. Hence, the static type system directly documents the recursive definition of `IdentifiedSequence`. In contrast, DT developers have to discover this either by reading the API's source code or by interpreting the error messages when they used the wrong classes in their code. We think that this is a very time consuming task and expect that DT developers will require more than ST developers.

3.7.5 Task 5

We suspected that Task 5 would be less difficult to understand (no recursive type definition, etc.), even though it had the largest number of classes that have to be identified. We assume that ST developers have an advantage over DT developers due to the number of identified classes (and not, as in task 4, the complexity of the type definitions).

3.8 Summary of the programming tasks

The programming tasks are trivial code, algorithmically speaking: no loops, conditions, or advanced language features are used. This is intentional; instead the tasks construct data structures. Each task requires the developer to identify the classes to instantiate and to pass them to other classes.

However, the tasks have different levels of difficulty. While tasks 1, 2, and 5, are relatively trivial programming tasks, tasks 3 and 4 are slightly more complex. For task 3, the conversion of some objects need to be understood and task 4 requires the understanding of a recursive data structure. Both tasks make use of generic types in Java, but from our point of view, task 4 is more complex, as the the recursive data definition is directly documented by the generic types for ST developers. In sum, we designed our tasks such that there was a variety of different complexity levels. In doing so, we attempted to provide insight into how this characteristic might influence our goal of learning the pros and cons of static and dynamic type systems.

3.9 Experiment Execution / Subjects

We recruited 33 subjects for our experiment. All subjects were (undergraduate as well as graduate) students from the the University of Duisburg Essen, Germany or the University of Chile – all of the students were trained as part of their studies in the programming language Java. Three subjects did not complete the programming tasks and abandoned the experiment (two starting with Groovy first, one starting with Java first); their results were excluded from the analysis.

Another subject was removed from the experiment, because after watching the measurements, it was revealed that the student spent a very large amount of time in reading the complete source code while working on task 2 and then solved the tasks 3 and 4 quickly – the subject confirmed in a following interview that he worked like that. We removed the subject because we considered our measurement method (time until a programming task was completed) to be insufficient in this situation. This is because it was not possible to determine how time should be now considered for the different tasks. For similar reasons, we removed two further subjects from the experiment, because they abandoned one task, switched then to another one and then came back to the original one. We finally considered 27 subject for the analysis – fifteen from Duisburg-Essen and twelve from Chile.

3.10 Threats to validity

As in any experiment, there is a number of threats to validity. Most of the threats are in common with previous experiments [10, 27] (such as chosen subjects, background of subjects, etc.) and will not be discussed here.

Internal Threat: Experimental Design. As already mentioned before, a general threat is that the underlying design assumes that the (possible) main effect (the effect of the type system) is larger than the possible learning effect in the within-subject measurement. If, however, the learning effect is very large, it potentially invalidates the main effect.

Internal Threat: Measurement. The chosen programming tasks largely depend on the underlying system. It might be the case that the underlying system is very simple so that the effect "undocumented API" does not play any role. We tried to avoid this problem by using an implementation which is from our subjective point of view complex enough – although we cannot argue exactly what "complex enough" means. For example, we may have chosen tasks that are either too simple or too hard. While this is true, we have carefully documented the tasks we used so that other research groups can reproduce them, modify them, or at least compare them to future work.

External Threat: Dynamically Typed API. The underlying system is created by creating a statically typed system first and then by removing the type annotations (i.e. replacing the type annotations with the keyword def and by removing interfaces and implements declarations). Consequently, the system does not make use of any possible features of dynamically typed languages. It could be possible that the existence of the dynamic type system would have a large impact of the overall system's architecture. From our point of view, the dynamically type system still represents a reasonable design—we do not think that the dynamic type system would have had any impact of the resulting structure.

External Threat: Artificiality of the tasks. At first glance, some tasks may seem to be intentionally complex, to the point of being artificial. However, we have found similar examples in the field. For instance, using the XML DOM API to write a document to a file, one must instantiate a `DocumentBuilderFactory`—catch various exceptions in the process—, then create a `Transformer` (through a second factory), and finally get a `NodeList` of elements of interest wrapped inside a `NodeSource` that is passed to the `Transformer`. Similar tasks to ours may happen in the field; we do not regard the tasks we designed as artificial.[4]

4. Experiment Results

This section illustrates the analysis of the experiment data. In order to ease the reading of the paper, we decided to shift parts of the analysis to the appendix. This choice was

[4] The full example is available at:
http://docs.oracle.com/javaee/1.4/tutorial/doc/JAXPXSLT4.html

made to focus on the most important parts of the analysis—which is otherwise quite systematic and long—and on the results themselves. In this section, we focus only on the time taken to solve the tasks. As the subjects did switch tasks only when they successfully passed a task, we do not need to analyze the correctness of the tasks; the solutions are by definition correct. Subjects unable to complete some tasks were discarded from the analysis (see section 3.9).

Structure of the results. We first give an overview of the results with descriptive statistics of the data (Section 4.1). We then conduct a *between-subject* analysis, which concludes that the results are significantly influenced by the tasks (Section 4.2). Given the variability of performance between subjects and tasks, the between-subject analysis does not find a main effect of the type systems. This is however the goal of the following *within-subject* analysis, which ascertains which language (and by extension type system) is better performing on a task-by-task basis (Section 4.3). We continue the analysis to conclude whether the number and complexity of the types to identify has a relationship with the difference we observe between the tasks (Section 4.4). We finally wrap up the results with a discussion motivating the need for a further exploratory analysis (Section 4.5).

4.1 Measurements and descriptive statistics

We start with a high-level analysis of the measurements of task completion time. A first view on the boxplot (see Figure 5) representing the measured data (see appendix A.2) seems to support the following statements: while for task one, four and five there is a tendency that the Groovy solutions required more time, task three (and to a certain extent, task two) required more time in Java.

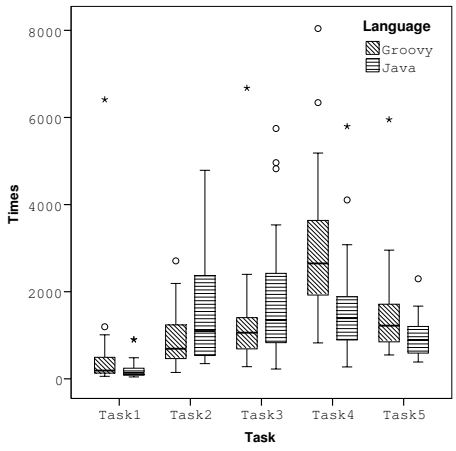

Figure 5. Boxplot for all measured data (combined round 1 and round 2) grouped by tasks

However, the raw data, the descriptive statistics and the visual representation only give a rough overview of the measurements. For instance, the boxplot does not consider that each subject is measured twice. To understand whether there is a significant difference, it is necessary to apply appropriate statistical tests (see [9, 22] for an introduction).

4.2 Repeated measures ANOVA

We start our analysis by running two repeated measures ANOVAs. The first was run on tasks 1-5 for the groups that used static or dynamic typing for the first time. The second ANOVA compared the groups that had "switched" from static to dynamic, or vice versa, for tasks 1-5 again. This analysis does not benefit from the within-subject measurement of each individual task; i.e., it cannot detect the effect of the static or dynamic type system on each subject for a given task. As explained above, this analysis is also sensible to the variability of the performance between subjects; it can, however, detect differences in performance due to the tasks themselves. Figures 6 and 7 show the boxplots for the first—respectively second—round where we group the results by the programming tasks.

We perform a repeated measures ANOVA on programming tasks and programming language. This analysis considers programming task and programming language as independent variables while we have the development times as dependent variable. The different programming tasks are considered within-subject, i.e. each individual's difference in development times for different tasks is considered.

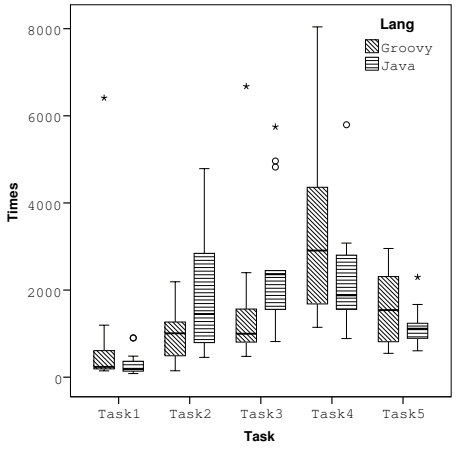

Figure 6. Boxplot for first round (no repeated measurement of same tasks)

We first observe that, in both the first and the second round, the dependent variable programming time is different for the different programming tasks—the ANOVA reveals a significant impact of the factor programming task for the first round ($p<0.001$[5], partial η^2=.354) as well as for the second round ($p<0.001$, partial η^2=.396). An interesting observation here is, that the partial η^2 values are very similar (which indicates that the impact of the task on the variance of the corresponding development time is similar). As a first

[5] The Greenhouse-Geisser was applied since the sphericity test turned out to be significant with $p<0.02$.

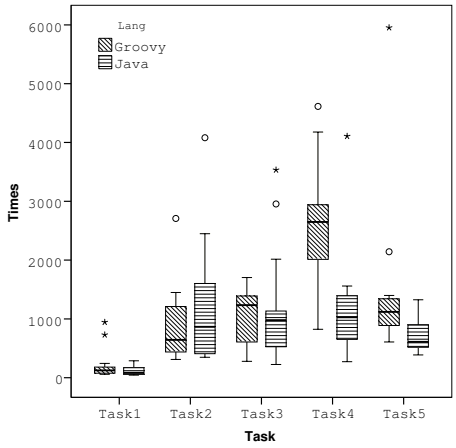

Figure 7. Boxplot for second round (no repeated measurement of same tasks)

conclusion, we can say that the individual task has a strong influence on the development time; this finding is in accordance with the existing literature [10, 21, 27].

A second observation is that there is a significant interaction between the factor task and programming language (in the first round p<0.025, partial η^2=.167; in the second round p<0.001, partial η^2=.195). This corresponds to our expectations: the experimental design assumes that the impact of the type system is different for the different tasks, since their complexity varies. With regard to the factor programming language, the first round did not reveal a significant impact (not even any tendency, since p>0.90!) while the second round reveals a close to significant difference (p<0.06). Consider what this means in plain English.

At first glance, it appears that we cannot reject our Null Hypothesis 1. Since no significant effect has been demonstrated, it seems that the impact of the programming language—if any—is so small that it is hidden by the deviation amongst tasks, developers, or any other confounds we have not considered here. This is however a weakness of this particular analysis, in cases—such as programming experiments—where the between-subject variability is high. This is the case here: as shown in the appendix, the standard deviation of the task completion time is comparable to the median—a common occurrence in programming experiments. Hence, the between-subject analysis is not able to take into account the (possible) effect of the programming language on each individual; as argued in our experimental design, we go on in the analysis with a within-subject analysis.

4.3 Analysis of groups and tasks: Null hypothesis 1

We now study the tasks and the groups in isolation, and combine the results of the analyses afterwards. Such a separated analysis means that we study, for each task, the differences between the Java and the Groovy development times for group A (Groovy first), and group B (Java first). According to section 3.4 we then combine the results of the tasks where one group reveals a significant difference while the other one either shows the same or no significant difference. In case both groups show contradicting differences for one task, it is not possible to interpret the result for this task.

The complete analysis of the separate groups is in appendix A.3. The combination of the analyses shows a significant difference for each task. For no task, the significance test revealed the same results, i.e., for no task did group A have the same positive impact of one language as group B. Additionally, no contradiction between the results appeared, i.e., for no task did one group have a different significant result than the other group.

Table 1 shows the results of the analysis for null hypothesis 1—that development time for completing a programming tasks in an undocumented API is equivalent when using either a static or a dynamic type system—and the corresponding p-values (see section A.3). For tasks 1, 4, and 5 we find a positive impact of Java; however, we find a positive impact of Groovy for tasks 2 and 3. Consequently, we reject the null hypothesis in all cases, but for tasks 2 and 3, the dominating language is contrary to our expectations.

Task	**Task 1**	**Task 2**	**Task 3**	**Task 4**	**Task 5**
P-values					
Group A	0.00	0.93	0.45	0.03	0.00
Group B	0.35	0.01	0.01	0.34	0.86
Dominating language	Java	Groovy	Groovy	Java	Java

Table 1. Experiment results for hypothesis 1 – Combination of both analysis for both groups

4.4 Analysis of complexity: Null hypothesis 2

Given the results from our task by task analysis, we fail to reject our second Null hypothesis:

- For the "easy" tasks (task one, two, and five) we find that the dynamic type system had a positive impact for task two, while it had a negative impact for tasks one and five. This contradicts our hypothesis that the number of types to identify is the main factor, since task two, where the dynamic type system prevailed, required to identify more types than task one, and less than task five.
- For tasks three and four—with the same number of types to be identified but with a different complexity—we observed another result that contradicts our hypothesis, since the dynamic type system prevailed in a task of "medium" difficulty (and did not prevail in all the "easy" tasks).

Consequently, a pure reduction of the programming tasks to the number and the difficulty of the types to be identified cannot be a main effect on the difference between static and dynamic typing; there are clearly other factors at work.

4.5 Discussion

The results show that the chosen experimental design was appropriate for the research question and the given task: the learning effect did compensate the main effect for the group starting with Java (no significant differences for task 1, 4 and 5) but it did not lead to contradictory results (such as one group showing positive effects for one language, while the other shows a positive effect for the other).

However, the study revealed unexpected results: While there was a significant positive impact of the static type system for tasks one, four, and five, there was a significant positive impact of the dynamic type system on tasks two and three. This result is surprising in several ways.

First, it is unclear why there should be a significant difference for task one: the task was only to instantiate a class—no parameters needed to be sent, nor anything that seems to be related to the question of type system seems to be relevant here. The subjects (which were mainly familiar with Java) could have been slightly surprised by seeing the keyword `def` in the code snippet. Following this argumentation leads to the conclusion that the Java solutions had a benefit in comparison to the Groovy solutions, possibly due to the subjects being more accustomed to Java than Groovy, despite the presence of the warmup task.

We were not surprised to see a positive impact of Java in tasks four and five: For task four, the static type system explicitely documents the recursive type definition and gives in that way developers a benefit in comparison to the dynamically typed version where this information was missing. For task five, a large set of types were necessary in order to initialize the required `Menu` object. While developers having the annotations of the static type system could directly see from the class definitions which objects needed to be instantiated, developers with the dynamic type system need to find out on their own which classes possibly match.

However, tasks two and three revealed a completely different picture: in both cases the developers with the dynamic type system were faster. Since our IDE gathers more information than the completion time of the tasks, we conducted an exploratory stury to better understand this result.

5. Exploratory Study

For some tasks the impact of the static or dynamic type system is contrary to our intuitions; we investigated whether this contrary effect might be explained by other factors.

A possible influencing factor is the number of builds and test runs that were performed during the implementation of each task. Assuming that people with the dynamically typed solutions need to explore on their own what kind of objects are required by the API, they would require more compilations and test runs in order to situate themselves.

Another influencing factor is the number of files developers were watching while working on a solution. People working on the dynamically typed solutions probably have to spend more time on different class definitions in order to find out what types might be expected by the corresponding API. A third—and related—data point is the number of file switches that occur during the session.

These data points are recorded by our IDE; we study them in turn. First, we analyze how often people built and ran the code. Second, we analyze the number of files that were consulted for each task. Finally, we analyze how much developers have navigated between files.

As we did previously, we shifted parts of the analysis to the appendix in order to ease the reading of the paper (see appendix A.4, A.5, and A.6).

5.1 Number of Builds and Test Runs

The numbers of builds and test runs we count here is not only the number of test runs on code that compiles, but the number of times a developer pressed the start-button from the IDE; this involved compiling and running the test cases. For statically typed solutions this test run also implies the type check, which potentially fails. The potential difference between such test runs could mean that people working on the statically typed solutions gain more information about the expected types than the people using the dynamically typed solutions. In contrast, the users of the dynamic type system need to explore on their own which types are required by the API, possibly leading to more run-time errors.

Task	Task 1	Task 2	Task 3	Task 4	Task 5
P-Values					
Group A	0.23	0.70	0.03	0.01	0.81
Group B	0.23	0.03	0.78	0.48	0.68
Less test runs	-	Groovy	Java	Java	-

Table 2. Wilcoxon tests for number of test runs

We apply the same analysis as in the previous section by studying the tasks and groups in separation (see appendix A.4). The result is in Table 2, which shows the language which had less builds and test runs. While the result is similar to the comparison of the development times for tasks two and four, no significant difference was found for tasks one and five. The result for task three is the opposite of the development time analysis: although task three required less test runs using Java, the development time was still higher.

At first glance, it seems that the number of test runs do not provide an appropriate explanation for the measured development time; we will elaborate on this in the general discussion.

5.2 Number of Watched Files

Next, we analyze the number of files being viewed by the developer. The number of watched files might be an indicator of the quantity of source code one has to read before one solves the task. A reason for differences would be that developers using a dynamic type systems are more likely to look into sources which are not related to the current task. If

one assumes that dynamically typed language do not directly guide developers to those types which are needed by the API, many different files are opened by the developer—the more files are opened, the larger the number of files which are not related to a given task.

Performing a separate analysis for the tasks and the language (see appendix A.5) reveals the results which are summarized in Figure 3: for all tasks (with the exception of task two) the number of watched files is higher for the dynamically typed solutions; for task two, it is the opposite.

Task	Task 1	Task 2	Task 3	Task 4	Task 5
P-Values					
Group A	0.02	0.35	0.05	0.01	0.00
Group B	0.67	0.01	0.85	0.04	0.05
Less watched files	Java	Groovy	Java	Java	Java

Table 3. Wilcoxon tests for number of watched files

Hence, it looks like that the number of watched files seems to provide similar results as the development time measurement, with the exception of task 3, where Groovy users watched more files than Java users, despite taking less time overall.

5.3 Number of File Switches

The number of file switches determines how often a developer switches between different files; the first opening of a file is already considered as a file switch. If for a given task a developer has only one file switch, then this means that he has solved the tasked without switching to another file. We can see the number of file switches as a measure of the amount of exploration that a developer has to perform in order to solve a task. The underlying assumption from the task design was that the use of the dynamically typed language would cause the developer to switch more often between different files as he or she needs to consult more code in order to decide which types are needed.

The difference between the number of file switches and the (previously analyzed) number of watched files is, that in the previous analysis each file that is opened more than once is only counted as one watched file.

Performing a separate analysis for the tasks and the language reveals the results shown in Table 4 (see appendix A.6 for the details of the analysis).

Task	Task 1	Task 2	Task 3	Task 4	Task 5
P-Values					
Group A	0.0	0.78	0.55	0.01	0.00
Group B	0.48	0.01	0.02	0.17	0.36
Less switches	Java	Groovy	Groovy	Java	Java

Table 4. Wilcoxon tests for number of file switches

The result of this analysis directly corresponds to the analysis of the development times (again, see Table 1): For task one, four and five the developers require less file switches for the statically typed solutions; they require more file switches for task two and three. What's even more interesting, the p-values for the different groups are quite similar.

Hence, the number of file switches seem to be a plausible indicator for the resulting development time. We will investigate the reasons for this in the discussion.

6. Discussion

We start with a summary of our findings in the main experiment and the exploratory analysis. Table 5 reports on all results of the measurements we investigated.

Aspect	Task 1	Task 2	Task 3	Task 4	Task 5
Less Development Time	**Java**	**Groovy**	**Groovy**	Java	Java
Less Builds/Runs	—	**Groovy**	Java	Java	**—**
Less Files watched	**Java**	**Groovy**	Java	Java	Java
Less File switches	**Java**	**Groovy**	**Groovy**	Java	Java
Our Expectations	—	Java	Java	Java	Java

Table 5. Summary of measurements we performed, and our expectations before carrying out the experiment. Unexpected results are shown in bold.

A priori expectations. Initially, we expected to see developers using the static type system (Java users) perform the tasks faster, with the exception of task one, where a negligible difference would be seen. Similarly, we expected subjects using the static type sytem to hold an advantage for tasks two to five, in all other metrics: number of builds and runs—indicator of ad-hoc explorations and trial-and-error; number of file switches—indicator of the amount of exploration; and number of files watched—indicator of quantity of code read to finish the task.

Results. As shown above, we found some surprising results: task 1 shows an unexpected advantage to Java; task 2, an unexpected and consistent advantage to Groovy; task 3, a time advantage to Groovy, reflected in file switches, but not in build and runs and files watched. On the other hand, task 4 yields consistently expected results, and task 5 yields expected results (except in builds and runs where there is no clear advantage to Java). We continue with a task-based discussion of the possible reasons for the results we observe.

Task 1. We measured a positive effect of the static type system although one would expect that typing hardly plays any role for such a simple task. A possible explantion would be the background of the subjects, who all had experience with Java, but not with Groovy. We included a warm-up task to alleviate this effect, but it still could be present.

Task 2. In all cases the results were better for the dynamically typed solutions (time, build and runs, file switches, and files watched). This is even more surprising if one accepts the argument put forward for task one—that the subjects had, because of their background, some advantage in Java.

A possible reason for the unexpected result is that the task is simple enough that types hinders more than they help.

The design of the API—having an `Initializer` object and additionally having a `Configuration` object which receives a string—may be intuitive; people can simply use it correctly without committing any obvious error.

In case this is not quite sufficient, the message-not-understood exception for the dynamically typed solutions may easily guide developers to the correct class; the error specifies which message is not understood, hinting at the correct class. In contrast, the subjects using the statically typed solution may be enticed to browse the classes they know they will use, in order to get more familiar with their API, even if the full API of these classes will not be used.

Hence, instead of trying to understand the class definition in detail, trial-and-error seems like a reasonable way of solving the task—and in simple cases, it may be efficient for developers to behave that way, instead of consistently reading class definitions.

Task 3. Contrary to task two, Groovy developers indeed watched more files than Java developers, even if they spent less time to solve the task. The Java developers also had less build and runs, and less files watched.

This finding is consistent with the interpretation of task two: Java developers spend more time reading API classes but read less API classes. They read these files with more attention, covering more of the API than Groovy developers. In contrast, Groovy developers seem to "jump around the system", more frequently compiling and running the system, and browsing more files in the process (including files less relevant to the task).

Hence, it is possible that the same "trial and error", and partial program comprehension approach that worked for task 2 still works for task 3. We already see the limits of the approach, as the "slow but deliberate" approach used in the Java solutions ends up requiring less builds and runs—less runtime errors—, and less investigation of unrelated files.

The task may also be simple enough that subjects easily know where to start: From the task definition, subjects were already aware that some kind of transformation is needed; the initialization of the tranformer object, and the construction of the graph may be intuitive enough for the "trial and error" approach to work.

On the other hand, Java users may have been confused by the presence of the generic types. Although they had the benefit that they can read directly from the Transformer's class definition that a graph object is required, they possibly spent more time on the definition of the generic types without a significant benefit.

Task 4. The argument that complex types reveal more about the structure of the program, but are harder to interpret would also explain the difference (pro Java) for task four.

Generic types were also required; while the generic type definitions reveals an important characteristic—the recursive type definitions—which directly help to understand the design of the underlying class. this was not the case in task three. In task three, it might have been helpful to see that an object of type `Graph` is required, but the additional type parameters may have reduced—even negated—the possible positive impact of the static type system.

In contrast, in task four, the recursive type definition is hard to understand without type information. Groovy users hence needed to read more files, more file switches, more builds and runs, and more time overall. According to our theory, the trial-and-error approach, appropriate for simple types, clearly shows its limit for complex types.

Task 5. Task five is also interesting in constrast to task two. Task five required more types to be identified, but the types to identify were of a similar complexity. Thus one could conclude that the more types need to be found, the more the statically typed solution is advantageous; less file browing is necessary, and less file switches as well. There are no differences in builds and runs, however. This could be due to the simplicity of the type definitions themselves (in comparison to task four).

Summary. Analyzing the tasks through the various metrics, we built a working theory of why we saw the results we observed: The difference in task one could be due to the experience of the subjects with Java.

For tasks two onwards, simple type definition may be easier to understand through trial-and-error than through static typing. Static typing encourages subjects to consult the class definition that are mentioned, whereas the users of the dynamic type system employ a more partial and less systematic program comprehension process. This approach shows its limits for more complex tasks, when more types, or more complex types are identified. These need more browing, file switches, and program runs (especially for complex type definitions), than the statically typed version.

This is only a working theory; it needs to be confirmed by other controlled experiments, and qualitative and quantitative program comprehension studies. Further, we are currently not able to formulate this theory more precisely: Although we suspect that, for example, task two and four differ with respect to their complexity, and although we think that the type system in task four documents better the design of the underlying classes, we cannot describe this in a way that we could determine to what extent this documentation is better. Likewise, the apparent correlation between file switches and development time warrants further investigation.

However, this working theory is a first step towards formulating a theory that describes differences in developer performance in statically and dynamically typed programs.

7. Conclusion

This paper described an experiment comparing static and dynamic type systems for programming tasks in an undocumented API. We gave 27 subjects five programming tasks and found that the type systems had a significant impact on the development time: for three of five tasks me measured a

positive impact of the static type system, for two tasks we measured a positive impact of the dynamic type system.

Based on the previous discussion, our overall conclusions for the use of static/dynamic type systems in undocumented APIs are the following:

1. **There is no simple answer to the static vs. dynamic typing question**: The question of whether or not static types are helpful for developers cannot be generally answered without taking the programming tasks into account. In fact, this is completely consistent with the results of previous experiments, such as Prechelt and Tichy's [21], or our own experiments[10, 27].
2. **The type system has an influence on the development time**: The choice of the static or dynamic type system had an influence on the development time for all programming tasks in the experiment. Again, this is consistent with previous experiments (such as [11, 21, 27]).
3. **Dynamic type systems potentially reduce development times for easier tasks**: Although we are currently not aware of how to determine exactly what "easy" and "hard" means, it seems that if a dynamic type systems has a positive impact, this is rather the case for easier tasks (which is consistent with the experiments described in [10, 27]). Although there was one counter example in the experiment (task 1), we think that the result for this task is rather a consequence of the chosen subjects' low familiarity with the dynamic language, Groovy (despite the presence of a warmup task).
4. **Static type systems reduce development times if**

   (a) the type annotations explicitely document design decisions, or

   (b) the number of classes in the programming tasks are relatively high.

   We argued for a) based on the fourth programming task (also in comparison to task two and three) and for b) based on the fifth programming task (in comparison to taks two and three). However, we also showed that a pure reduction of the number of classes is not valid (see section 4.4).

Comparison to related work. The experiment by Kleinschmager et al. [15] should also be taken into account; there, for no single task a positive impact of the dynamic type system could be shown. Comparing the tasks of the experiment presented here and the one in [15], we think that (again) the difference lies in the complexity of the programming tasks. While the programming tasks here require to instantiate an configure objects, the programming tasks in [15] required more interactions between the objects. We think that it is necessary in future experiments not only to think about complexity from the perspective of "number of unknown artefacts that should be used" but also from the perspective of "complexity of the required interaction between the developer and the API classes".

Future work. While there is already some strong evidence for our first two conclusions (as they agree with several previous experiments in the literature), we think that much more research is required in order to give more evidence for the conclusions three and four. We plan to do so in subsequent studies. Rather than closing the issues, we see this experiment as starting point that opens a number of questions for following experiments.

Our discussions (and even the original research question) were strongly related to the idea that the static type system has a positive impact on the documentation. In fact, this is related to the kind of static type system as provided by programming languages—such as Java—where the type system also implies the existence of corresponding type annotations in the code.

In a follow up experiment, we will study if the positive impact of the static type system, as shown in the programming tasks four and five, can also be achieved only with type annotations in the code—without static type checks. This experiment is currently planned and will be executed in the very near future. A related experiment would be to evaluate the benefits of a statically typed language using type inference instead of annotations (such as Haskell or ML).

Another question is to what extent the type system plays a role if the API is well documented. A good documentation may reduce the effect of the (static or dynamic) type system on the developer performance. Works such as [14, 23] provide additional insights about other possible influencing factors for the use of APIs, which might have a larger effect than the type system of the underlying programming language.

Strong empirical evidence is needed to back the arguments in favor of static or dynamic type systems. From that perspective, we consider this experiment as a valuable contribution. We also think that much more experimentation is needed in order to strengthen the evidence—the current state of experimentation in the area of programming languages, and type systems in particular, is weak. We hope that more experiments and replications will be performed by the programming language community at large.

Acknowledgments

The authors would like to thank the volunteers from the University of Duisburg-Essen and from the University of Chile that participated in the experiment.

References

[1] BIRD, R., AND WADLER, P. *An introduction to functional programming*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988.

[2] BORTZ, J. *Statistik: für Human- und Sozialwissenschaftler*, 6., vollst. überarb. u. aktualisierte aufl. ed. Springer, September 2005.

[3] BROOKS, R. E. Studying programmer behavior experimentally: the problems of proper methodology. *Commun. ACM 23* (April 1980), 207–213.

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

[5] CURTIS, B. Five paradigms in the psychology of programming. In *Handbook of Human-Computer Interaction*, M. Helander, Ed. Elsevier (North-Holland), 1988, pp. 87–106.

[6] DALY, M. T., SAZAWAL, V., AND FOSTER, J. S. Work in progress: an empirical study of static typing in ruby. *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU),Orlando, Florida, October 2009* (2009).

[7] FENTON, N. E., AND PFLEEGER, S. L. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 1998.

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

[9] GRAVETTER, F., AND WALLNAU, L. *Statistics for the Behavioral Sciences*. Cengage Learning, 2008.

[10] HANENBERG, S. 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* (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 22–35.

[11] HANENBERG, S. Faith, hope, and love: An essay on software science's neglect of human factors. In *to appear in Proceedings of Onward/SPLASH* (October 2010).

[12] HANENBERG, S. A chronological experience report from an initial experiment series on static type systems. In *2nd Workshop on Empirical Evaluation of Software Composition Techniques (ESCOT)* (Lancaster, UK, 2011).

[13] JURISTO, N., AND MORENO, A. M. *Basics of Software Engineering Experimentation*. Springer, 2001.

[14] KAWRYKOW, D., AND ROBILLARD, M. P. Improving api usage through automatic detection of redundant code. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009* (2009), IEEE Computer Society, pp. 111–122.

[15] KLEINSCHMAGER, S., HANENBERG, S., ROBBES, R., TANTER, É., AND STEFIK, A. Do static type system improve the maintainability of software systems? an empirical study. In *Accepted for publication in: International Conference on Program Comprehension, ICPC'12* (Passau, German, 2012).

[16] KOENIG, D., GLOVER, A., KING, P., LAFORGE, G., AND SKEET, J. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[17] MCCONNELL, S. What does 10x mean? Measuring variations in programmer productivity. In *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, Eds., O'Reilly Series. O'Reilly Media, 2010, pp. 567–575.

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

[19] PRECHELT, L. An empirical comparison of seven programming languages, ieee computer (33). *Computer 33* (2000), 23–29.

[20] PRECHELT, L. *Kontrollierte Experimente in der Softwaretechnik*. Springer, Berlin, March 2001.

[21] PRECHELT, L., AND TICHY, W. F. A controlled experiment to assess the benefits of procedure argument type checking. *IEEE Trans. Softw. Eng. 24*, 4 (1998), 302–312.

[22] RICE, J. A. *Mathematical Statistics and Data Analysis*. Duxbury Press, Apr. 2001.

[23] ROBILLARD, M. P. What makes apis hard to learn? answers from developers. *IEEE Software 26*, 6 (2009), 27–34.

[24] SHNEIDERMAN, B. *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop Publishers, August 1980.

[25] SIEK, J. G., AND TAHA, W. Gradual typing for objects. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings* (2007), vol. 4609 of *Lecture Notes in Computer Science*, Springer, pp. 2–27.

[26] STEINBERG, M., AND HANENBERG, S. What is the impact of static type systems on debugging type errors and semantic errors? an empirical study of differences in debugging time using statically and dynamically typed languages - unpublished work in progress.

[27] STUCHLIK, A., AND HANENBERG, S. 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* (Portland, Oregon, USA, 2011), DLS '11, ACM, pp. 97–106.

[28] WOHLIN, C., RUNESON, P., HÖST, M., OHLSSON, M. C., REGNELL, B., AND WESSLÉN, A. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

A. Appendix

A.1 Experimental Design

A general question when designing an experiment is whether it should be done as a between-subject (without repeated measurement) or a within-subject (with repeated measurement on each subject) experiment. The main reason for choosing a within-subject design is that programming tasks are typically quite complex. The subjects do not only have to manage the intellectual effort of the programming tasks, they must additionally be able to handle a development environment such as an editor, understand compiler or runtime exceptions, etc. We are far from ideal situations, such as measuring reaction times to simple stimuli. As a consequence, programming tasks have a large deviation (see for example [5, 17] for a more detailed discussion on this) that potentiallys hide the main effect that should be studied in the experiment. This does not happens because there is no main effect; this happens because the individual differences among subjects are much larger than the main ef-

fect. Within-subject design help to overcome this problem (see [3])—although the potential problem of learning effects needs to be considered.

We assumed for the experiment a (positive) impact of the static type system on the measured results, but we also assumed a positive impact of the learning effect. Consequently, the repeated measurement for each subject contains both, the learning effect as well as the type system effect.

Figure 1 in section 3.4 illustrates the implications of the within-subject design – whose measurement implies the learning as well as the language effect – in more detail: the upper and the lower part of the figure represents two (very similar) subjects in the two different groups. The figure contains the first as well as the second measurement. For group one (Groovy first) this means that the first measurement is the development time for the Groovy solution, while for the second group (Java first) the first measurement is the development time required for the Java solution. The second measurement (and hence the difference between the first and the second measurement) represents the development time required for the second language. The second measurement also contains the learning effect, which in both cases reduces the development time of the first measurement.

Under the assumption that the static type system reduces the development time, the second measurement for a subject in the group starting with Groovy must be much smaller than the first measurement, because learning effect as well as the language effect reduce the development time. Under the assumption that the learning effect is smaller than the language effect, the second measurement for a subject in the group starting with Java is higher than the first measurement. Hence, if for each subject in the sample the first and the second measurements are similar to the ones shown in Figure 1, it can be concluded that the development time using the static type system in Java requires less time than the development time using the dynamic type system in Groovy.

Keep in mind that the resulting analysis is performed on a sample; it is not necessary that the assumptions hold for each subject in the sample, they should hold only in the average.

The experimental design would fail if the learning effect is much larger than the effect of the type system. In such a situation, the experiment would reveal for both groups a significant decrease of development times. Figure 8 illustrates this potential problem: In both cases the language effect is rather small in comparison to the learning effect, i.e. the learning effect dominates the language effect. Consequently, for the subjects in both groups the second measurement would be lower than the first measurement. In such a situation the experiment would not reveal anything with respect to type systems – in both situations the second measurement is lower than the first measurement. Because of the above mentioned large deviations, the differences between the first and the second measurements will probably not be significant. Hence, it would only be possible to conclude that

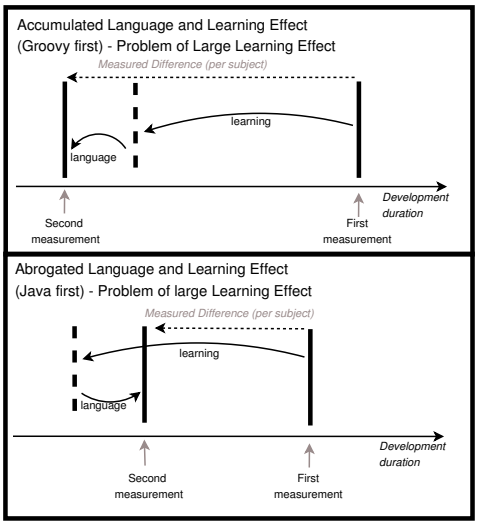

Figure 8. Potential problem in the experimental design: large learning effect

subjects who perform a programming tasks for the second time, are quicker in the second round – which is irrelevant for the research question followed by the experiment.

Importantly, the design does not require the learning effect to be much smaller than the language effect. It is valid if both effects have a comparable size. In such a situation, the group with the accumulated effects will show (significant) differences between the first and the second measurement while the group with the abrogated effects does not.

A.2 Measured Development Time and Descriptive Statistics

Table 6 contains all measured data for all 27 subjects in the experiment. In addition to the raw data we also include the differences for each subject and each task in the table (Groovy times - Java times), i.e. a negative value means that the subject required more time using Java than using Groovy. Table 7 shows the descriptive statistics for the measurements. It turns out, that the standard deviation for the different tasks is quite large (in most cases comparable to the median) – a phenomenon that can be often found in programming experiments – which from our point of view strengthens the experimental design based in a within-subject comparison (see sections 3.4, A.1 and 3.10).

For task one 19 subjects required more time for the Groovy solution than for the Java solution. For task two and three, just 10 subjects required more time for the Groovy solution, for task four 22 subjects required more time for the Groovy solution, and finally 20 subjects required for task 5 more time for Groovy than for Java. For task one, four and five a majority required more time using the dynamic type system while for task two and three it is the opposite.

		Task 1			Task2			Task3			Task4			Task5			Sums		
No	Group	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff
1	B	897	124	-773	793	1336	543	2362	1390	-972	3079	2225	-854	1113	941	-172	8244	6016	-2228
2	B	179	58	-121	1152	373	-779	817	1703	886	1307	2414	1107	949	1342	393	4404	5890	1486
3	B	175	944	769	2868	1210	-1658	2447	1271	-1176	1887	1834	-53	1669	1399	-270	9046	6658	-2388
4	B	196	110	-86	1451	644	-807	4823	607	-4216	899	1554	655	1148	1220	72	8517	4135	-4382
5	B	89	60	-29	596	691	95	2191	278	-1913	1877	824	-1053	605	1126	521	5358	2979	-2379
6	B	366	121	-245	2842	2708	-134	1348	1234	-114	1761	2942	1181	2296	1120	-1176	8613	8125	-488
7	B	122	139	17	1021	310	-711	2444	805	-1639	886	2942	2056	893	887	-6	5366	5083	-283
8	B	82	69	-13	454	633	179	845	1419	574	2817	3113	296	890	607	-283	5088	5841	753
9	A	86	448	362	2429	1977	-452	1108	924	-184	1054	2238	1184	1326	1484	158	6003	7071	1068
10	A	58	228	170	348	492	144	634	518	-116	455	1671	1216	644	1025	381	2139	3934	1795
11	A	105	144	39	461	345	-116	621	805	184	382	1144	762	532	548	16	2101	2986	885
12	A	45	225	180	1380	951	-429	1134	1604	470	652	2625	1973	577	1774	1197	3788	7179	3391
13	A	63	539	476	404	358	-46	917	477	-440	738	2473	1735	387	2705	2318	2509	6552	4043
14	A	83	248	165	944	2190	1246	515	1247	732	1514	4359	2845	862	1652	790	3918	9696	5778
15	A	62	177	115	374	1266	892	338	515	177	697	3714	3017	435	759	324	1906	6431	4525
16	B	139	181	42	648	362	-286	1558	328	-1230	1557	4179	2622	988	615	-373	4890	5665	775
17	B	904	76	-828	4644	437	-4207	4963	1378	-3585	5795	2663	-3132	1296	1078	-218	17602	5632	-11970
18	B	483	730	247	4787	608	-4179	5747	768	-4979	2801	4614	1813	1238	5953	4715	15056	12673	-2383
19	B	183	243	60	1673	683	-990	2344	500	-1844	1848	2012	164	709	812	103	6757	4250	-2507
20	B	255	155	-100	2298	1449	-849	2398	1472	-926	2144	2649	505	1236	2142	906	8331	7867	-464
21	A	44	144	100	1097	1775	678	225	1059	834	272	8042	7770	407	2955	2548	2045	13975	11930
22	A	273	1195	922	1602	847	-755	527	866	339	1392	5185	3793	1170	2310	1140	4964	10403	5439
23	A	107	6411	6304	4081	147	-3934	1088	1388	300	1001	1682	681	897	1596	699	7174	11224	4050
24	A	169	190	21	408	1060	652	1040	936	-104	4107	1555	-2552	517	813	296	6241	4554	-1687
25	A	234	1011	777	2449	665	-1784	2955	1564	-1391	1296	3558	2262	1235	2612	1377	8169	9410	1241
26	A	288	610	322	789	1180	391	2015	6676	4661	1559	6341	4782	567	1387	820	5218	16194	10976
27	A	75	216	141	473	1059	586	3534	2399	-1135	1292	3190	1898	645	735	90	6019	7599	1580

Table 6. Development times for all 27 subjects; times in seconds, differences are Groovy times - Java times; Group A = Groovy first; Group B = Java first

	Task 1			Task2			Task3			Task4			Task5		
No	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff
min	44,0	58,0	-828,0	348,0	147,0	-4207,0	225,0	278,0	-4979,0	272,0	824,0	-3132,0	387,0	548,0	-1176,0
max	904,0	6411,0	6304,0	4787,0	2708,0	1246,0	5747,0	6676,0	4661,0	5795,0	8042,0	7770,0	2296,0	5953,0	4715,0
mean	213,4	548,0	334,6	1572,8	953,9	-618,9	1886,6	1264,1	-622,5	1669,2	3027,5	1358,3	934,5	1540,6	606,1
median	139,0	190,0	100,0	1097,0	691,0	-286,0	1348,0	1059,0	-184,0	1392,0	2649,0	1184,0	893,0	1220,0	324,0
std. dev.	224,1	1212,8	1252,7	1314,5	634,6	1452,5	1473,2	1189,4	1834,0	1215,4	1622,4	2153,1	431,9	1106,5	1143,1

Table 7. Descriptive statistics for development time (time in seconds for all but standard deviation)

A.3 Within-Subject Analysis

Figure 9 illustrates the boxplot for the within-subject measurement of group A (the group that started first with the dynamically typed tasks). The differences to the between-subject measurement from Figure *10* are obvious: tasks four and five reveal differences, task one and three show potential differences and task two probably does not reveal any difference in the development time of Groovy and Java. Furthermore, the differences correspond to the expectations: in all cases the median of the Groovy development times is larger than the median of the Java development times. As argued in section 3.4 this difference consists not only of the difference between the type systems but also of the learning effect.

The significance tests confirm the previous impressions concerning differences for the five tasks. Because the data is now based on a within-subject measurement, the Java and Groovy development times should not be tested for normality separately. Instead, the differences should be checked (see [2]). Table 8 shows the results for the Shapiro-Wilk-test in order to check the normality assumption, and the corresponding p-values for the Wilcoxon-test and t-test. For task 1, 4 and 5, the differences are significant

Figure 10 is the boxplot for the group B. For task 1 and 2, there is a potential benefit for Groovy; for task 3, the Groovy benefit is obvious; for task 4 and 5, a potential advantage to Java. We perform the same analysis as before: we check the

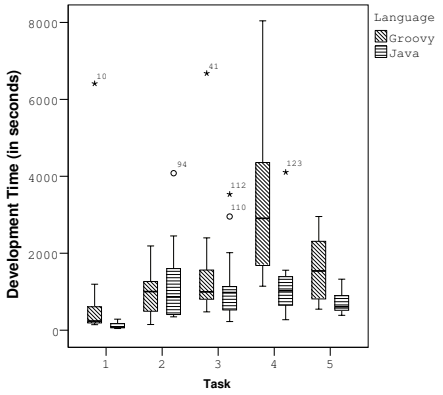

Figure 9. Boxplot for the within-subject comparison of Group A (Groovy first)

Task	Task 1	Task 2	Task 3	Task 4	Task 5
Shapiro-Wilk p-value	0.00	0.01	0.00	0.29	0.05
Applied test	Wilcoxon	Wilcoxon	Wilcoxon	t-Test	Wilcoxon
p-value	0.00	0.93	0.45	0.03	0.00
dominating language	Java	-	-	Java	Java

Table 8. Significance tests for the within-subject comparison of Group A (Groovy first)

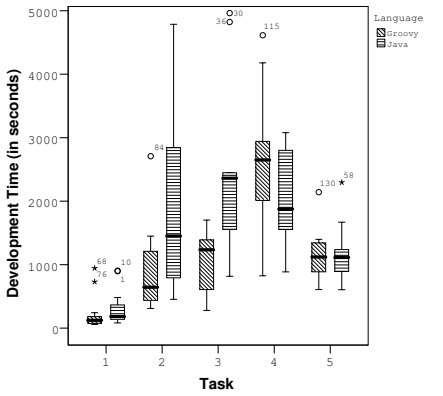

Figure 10. Boxplot for within-subject comparison of Group B (Java first)

differences in measurements for normality, then we perform either the Wilcoxon-test or the t-test (Table 9).

Task	Task 1	Task 2	Task 3	Task 4	Task 5
Shapiro-Wilk p-value	0.08	0.00	0.43	0.60	0.00
Applied test	t-Test	Wilcoxon	t-Test	t-Test	Wilcoxon
p-value	0.35	0.01	0.01	0.34	0.86
dominating language	-	Groovy	Groovy	-	-

Table 9. Significance tests for the within-subject comparison of Group B (Java first)

A.4 Test Runs

Table 10 contains the descriptive statistics for the number of test runs, and Figure 11 shows the boxplot. It seems as if the boxplot (which does not take into account that each subject is measured twice) already indicates that there is no general tendency with respect to the number of test-runs: for task one, three, and four there seems to be a tendency that the number of test runs is higher for the dynamically typed solutions. For task two and five it is unclear whether there is a difference between the number of test runs for the dynamically typed or the statically typed solutions.

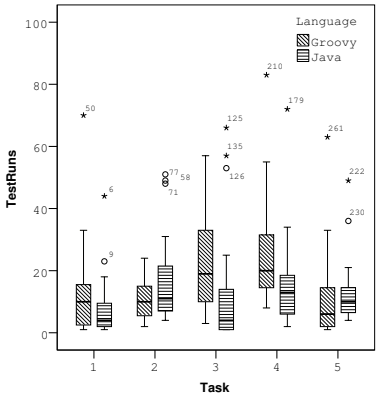

Figure 11. Boxplot for test runs

Performing a repeated measures ANOVA on the test runs reveals a first interesting characteristic. For the first round, with p<.01 and η^2=.142 there is a significant impact of the programming tasks. However, the interaction between the factor task and programming language is only close to significant (p<.07 and η^2=.092). Again, the factor programming language is not significant (p>.99). This is similar to the results for the measured development time. However, the second round reveals different results. Neither the factor programming task nor the interaction between task and programming language are significant (p=.101, respectively p=0.188). Instead, the factor programming language turns out to be significant (p<.01, η^2=.30). A non-parametric test shows significant differences for tasks 2, 3, and 4 (Table 11).

Group A (Groovy first)

Task	Task 1	Task 2	Task 3	Task 4	Task 5
p-value	0.23	0.70	0.03	0.01	0.81
less test runs	-	-	Java	Java	-

Group B (Java first)

p-value	0.23	0.03	0.78	0.48	0.68
less test runs	-	Groovy	-	-	-

Table 11. Wilcoxon tests for number of test runs

A.5 Watched Files

The repeated measures ANOVA on the number of watched files reveals for the second round a significant factor of pro-

No	Task 1			Task2			Task3			Task4			Task5		
	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff
min	1	1	0	4	2	-2	1	3	2	2	8	6	4	1	-3
max	44	70	26	51	24	-27	66	57	-9	72	83	11	49	63	14
mean	7.67	12.15	4.48	16.78	10.63	-6.15	12.33	20.81	8.48	15.67	25.89	10.22	12.48	11.48	-1.00
median	4	10	6	11	10	-1	4	19	15	13	20	7	10	6	-4
std. dev.	9.17	13.99	4.82	13.96	6.04	-7.92	18.14	14.64	-3.51	14.37	16.68	2.31	9.80	14.25	4.45

Table 10. Descriptive statistics for number of test runs

gramming task (p=.0 and η^2=.31), a significant interaction between the programming task and the group (p<.02 and η^2=.116) and non-significant factor group (p=.19). For the second round, the factor programming task is significant (p=.0 and η^2=.367) the interaction is significant (p<.01 and η^2=.176) as well as the factor group (p=.0 and η^2=.35). The results of the Wilcoxon-test for both groups is in Table 12.

Group A (Groovy first)

Task	Task 1	Task 2	Task 3	Task 4	Task 5
p-value	0.02	0.35	0.05	0.01	0.00
more files	Groovy	-	Groovy	Groovy	Groovy

Group B (Java first)

p-value	0.67	0.01	0.85	0.04	0.05
more files	-	Java	-	Groovy	Groovy

Table 12. Wilcoxon tests for number of watched files

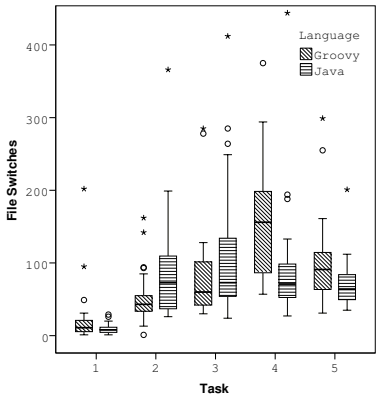

Figure 12. Boxplot for file switches

A.6 File Switches

Table 13 contains the descriptive statistics for the number of test runs, and Figure 12 shows the boxplot. The boxplot shows that the number of file switches is comparable to the development time.

The repeated measures ANOVA reveals for the first round a significant factor of programming task (p=.0 and η^2=.36), significant interaction between the programming task and the group (p<.01 and η^2=.173) and a non-significant factor group (p>.75). For the second round, the factor programming task is significant (p=.0 and η^2=.412) the interaction is significant (p<.01 and η^2=.157) and the factor group is close to significant (p<.06). The Wilcoxon-test on both groups is shown in Table 14.

Group A (Groovy first)

Task	Task 1	Task 2	Task 3	Task 4	Task 5
p-value	0.0	0.78	0.55	0.01	0.00
more switches	Groovy	-	-	Groovy	Groovy

Group B (Java first)

p-value	0.48	0.01	0.02	0.17	0.36
more switches	-	Java	Java	-	-

Table 14. Wilcoxon tests for number of file switches

	Task 1			Task2			Task3			Task4			Task5		
No	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff	Java	Groovy	Diff
min	1	1		26	1	-25	24	30	6	27	57	30	35	31	-4
max	29	202	173	366	162	-204	412	285	-127	444	375	-69	201	299	98
mean	9.67	23.63	13.96	90.33	52.96	-37.37	113.33	80.41	-32.93	91.59	156.37	64.78	71.74	102.07	30.33
median	8	11	3	73	43	-30	73	60	-13	71	156	85	64	91	27
std. dev.	7.13	40.36	33.23	75.43	35.94	-39.49	93.14	65.10	-28.04	82.28	76.41	-5.86	32.81	60.69	27.88

Table 13. Descriptive statistics for number of file switches