An Experiment About Static and Dynamic Type Systems

Doubts About the Positive Impact of Static Type Systems on Development Time

Stefan Hanenberg

Institute for Computer Science and Business Information Systems
University of Duisburg-Essen
Schützenbahn 70, D-45117 Essen, Germany
stefan.hanenberg@icb.uni-due.de

Abstract

Although static type systems are an essential part in teaching and research in software engineering and computer science, there is hardly any knowledge about what the impact of static type systems on the development time or the resulting quality for a piece of software is. On the one hand there are authors that state that static type systems decrease an application's complexity and hence its development time (which means that the quality must be improved since developers have more time left in their projects). On the other hand there are authors that argue that static type systems increase development time (and hence decrease the code quality) since they restrict developers to express themselves in a desired way. This paper presents an empirical study with 49 subjects that studies the impact of a static type system for the development of a parser over 27 hours working time. In the experiments the existence of the static type system has neither a positive nor a negative impact on an application's development time (under the conditions of the experiment).

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms Experimentation, Human Factors, Languages.

Keywords *Type Systems, Programming Languages, Empirical Study, Dynamically Typed Languages*

1. Introduction

Static type systems (see for example [1, 19]) are one of the major topics in research, teaching as well as in industry. In research, new type systems appear frequently either for existing programming languages (such as for example the introduction of Generics in Java) or new programming languages are constructed that provide a new static type system.

In teaching, students are educated in the formal notation of static type systems as well as in proofs on static type systems (see for example [1, 19]). In industry, type systems become important for different reasons. Possibly, a programming language in use evolves by introducing a new static type system. If this new static type system should be applied, developers need to be educated, which causes additional costs. Maybe existing libraries or products should be adapted to match the new type system which also causes additional costs. Finally, additional tools might be required due to the new type system (such as tools that measure the current state of the software product) which potentially also cause additional costs.

An example for such a static type system that evolves in a programming language with industrial relevance is Java: since the introduction of Generics in Java there is still a huge number of applications and APIs available that do not use Generics. Hence, for industry a main question is whether it is rewarding to educate developers to use the new static type system and whether there will be a return on investment for migrating existing non-generic APIs to the new static type system.

In general, for industry it is important to determine whether such an investment is reasonable, i.e. whether the expected benefit of static type systems represents some future revenues. This means, it is necessary to understand what the advantages and maybe additional costs of using a static type system are.

In industry, it is also observable that dynamically typed programming languages such as PHP or Ruby become more and more important for specific domains such as website engineering. Also, dynamically typed programming languages such as Tcl or Perl are still used in software development. For future developments of such languages it seems valid to ask, whether new releases of such languages should provide a static type system – assuming that a static type system has a positive impact on software development.

However, while static type systems are well-studied from the perspective of theoretical computer science, there is hardly any knowledge about whether a static type system plays a relevant role in the practical application of a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA/SPLASH'10 October 17–21, 2010, Reno/Tahoe, Nevada, USA.
Copyright © 2010 ACM 978-1-4503-0203-6/10/10…$10.00.

programming language. There is even a number of researchers that advocate the use of dynamically typed programming languages instead of statically typed ones (see for example [26]) – and who emphasize the tradition of dynamically typed programming languages such as Smalltalk, Lisp, etc..

In literature, typical arguments for static type systems (see e.g. [2]) are for example:

Static type systems...

- capture a large fraction of recurring programming errors
- have methodological advantages for code development
- reduce the complexity of programming languages
- improve the development and maintenance in security areas

On the other hand common arguments against static type systems can be found for example in [6, 18, 26]:

- Static type systems unnecessarily restrict the developers
- No-such-method exceptions which are caused at run-time because of missing type-checks do not occur that often
- No-such-method exceptions mainly occur because of null-pointer exceptions (which also occur in typed programming languages)

The arguments for and against static type systems seem to be valid but contradict each other. For industry, a common problem is that it is unclear which arguments should be trusted.

This paper contributes to this question by an experiment whose focus is on static type systems and software development, i.e. we address the question whether a type system has a positive impact on the development of a piece of software.

Thereto, we introduce an experiment which measures at two different points whether either the development time of a rather easy piece of software was improved by a static type system, or whether the resulting quality of a large piece of software was decreased.

This paper shows that within the experiment the use of the statically typed programming language did not have a significant positive impact on the software project – while there was a significant difference in the development time for the smaller piece of software (for a subproject of the whole one, the dynamically typed developers significantly performed better), there was no significant difference between the statically and dynamically typed solutions with respect to the resulting quality of the whole project.

Section 2 briefly discusses related work in the area of empirical studies on type systems. Section 3 discusses shortly the background of the experiment by explaining first the logic of empirical experimentation and second the initial considerations for the experiment. Section 4 introduces the experiment. Sections 5 and 6 analyze the data. Finally, section 7 summarizes and concludes this paper.

2. Related Work

We are aware only of three works which are directly in the area of empirical evaluations of usability of static type systems apart from our works.

The first one by Prechelt and Tichy [23] concentrates on the impact of static type checking in procedure arguments. Prechelt and Tichy checked the impact of static typing by letting subjects perform a programming task in a language with static type checking as well as in a language without static type checking with the result that there is a significant positive impact on the developer's productivity for those developers that used the static type checker. The main difference between the here introduced experiment and the one by Prechelt and Tichy is, that the here introduced experiment considers the development of an application from scratch, while the authors of [23] mainly address the use of a certain API. The amount of time required by the task studied in [23] (which is approximately 5 hours) is comparable to the first point of measurement analyzed here. It is noteworthy, that the results of the experiment in [23] contradict the results of the here introduced experiment. Concerning the languages the difference between the here introduced experiment and the one in [23] is, that in [23] two C dialects were used, where both of them required type declarations in the code (where only one language performed the static type check) – the here introduced experiment uses a language with and without type declarations.

The second experiment we are aware of is the one performed by Gannon [7]. There, also a controlled experiment was performed that compares the use of a statically typed vs. dynamically typed programming language. Again, the result of the study in [7] reveals a positive impact of static typing.

The third experiment is a qualitative pilot-study on type systems [5] where programmers were observed that used a new type system for an existing language. Although the study is just a qualitative one, the authors seem to suggest that at least in the specific setting of the experiment the benefit of the type system could not directly be shown.

In [13] we performed a small study (where the required development time for all tasks per subject was less than 2 hours) on the impact of static typing using the programming languages Java and Groovy, where Groovy was applied only as a dynamically typed Java without the additional language features provided by Groovy. In fact, the study in [13] was performed after the study described in this paper and the outcome of the work described in this paper directly influenced the experimental design of [13]. The tasks in [13] were designed in a way, that the tasks differed with respect to the number of expected type casts that needed to be performed in order to fulfill the programming tasks. The outcome of the experiment in [13] was, that subjects required less development time in order to fulfill tasks which have a small number of type casts. However, for bigger tasks (with even more expected type casts) no difference in the development time was measured.

One further experiment by Prechelt [22] could also be considered as an experiment that compares statically typed and dynamically typed programming languages. However, the main focus of the experiment is not the typing issue but the question whether or not the language is a script or compiler language. In that experiment, the programming effort for developers using the (untyped) languages Tcl, REXX, Python and Perl was rather lower than the effort for C++, Java and C. However, while this experiment seemed to be an argument for dynamically typed

languages, it is unclear whether the static type system was an influencing factor for this difference.

Based on the previously described experiment by Prechelt, Gat performed an experiment [8] in order to determine differences in development times, execution times, etc. between Java, C++ and Lisp. With respect to programming effort, it turned out that the effort for Lisp programmers was less than for C++ and Java programmers. Again (corresponding to the previous paragraph) it is unclear whether this difference was caused by the different type systems – or whether it was caused by different language semantics, IDEs, etc..

In [14] Hudak and Jones describe an experiment that also measures the impact of different programming languages. While in correspondance to [8] Lisp seems to have a positive impact on development time, it also seems that programming in (the statically typed programming language) Haskell requires less effort than for example Ada.

Finally, in [12] a first impression of the here described experiment is given (without any detailed experiment description or any detailed analysis).

3. Background and Motivation

While the use of empirical methods has some relevance and background in the area of software engineering (cf. for example [15, 21, 27]) its application in the area of programming language design is not often practiced. Due to this, this section gives first a short overview of the logic of experimentation. Afterwards, initial considerations for designing an experiment are described.

3.1 Logic of Empirical Experimentation

In order to understand the possible insights of the here presented experiment, it is necessary to understand the logic of the underlying empirical experimentation.

There is a number of different approaches to achieve empirical insights via experimentation (see for example the introduction in [15] and the discussion in [10]) such as benchmarks for measuring the performance of languages or simulation for generating possible scenarios for user input, etc.. A number of these approaches follow the idea of being capable to generalize from an experiment's results. The author would like to emphasize that this is not the case for the here described experiment.

The experiment in this paper is built to test a hypothesis: starting from the existing experiments with a quantitative analysis that were already described in the related work section (see [23, 7] with the corresponding description in section 2) the experiment's hypothesis is, that statically typed programming languages have a positive impact on the development time for a programming task, i.e. using a statically typed language instead of a dynamically typed one saves development time for a programming task. Using this hypothesis an experiment was designed and performed in order to check whether this hypothesis (still) holds. In case the hypothesis turns out to be confirmed by the experiment, the experiment can be considered as one more indicator that the initial hypothesis is true. In case the experiment rejects the hypothesis, a situation is found where the hypothesis does not hold – consequently, the general statement of the original hypothesis needs to be reconsidered and additional research and experimentation is required in order to determine more precisely under which circumstances static type systems save development time.

It is important to note that it is not possible to draw a general statement from an experiment. If for example the experiment confirms the hypothesis, it is still no general proof of the hypothesis. In the same way, if the outcome of the experiment is completely different than the original hypothesis (i.e. if the experiment would show that dynamically typed languages reduce development time) it is still no general proof that dynamically typed languages save development time. It is also not the case that a rejected hypothesis proves the falsity of previous experiments such as the ones in [23, 7]: a single experiment is just another test of the original hypothesis[1].

Although it would be desirable to have a large set of experiments to check a single hypothesis, there is a reason why typically no larger sets of experiments are available (see also [25, 10]): performing a single experiment as the one described in this paper requires a large set of resources. The experiment of this paper required approximately 2000 hours working time of subjects in a controlled setting (time for teaching subjects and time required by subjects to solve a programming task). Furthermore, the preparation of the experiment (experimental design, implementing a programming language for the experiment, etc.) required some additional months. Due to a typical limit of resources, it is rather not the case that a larger number of experiments with a size comparable to the one described in this paper is available (especially not by a single working group) – although is would be desirable in order to have a broader spectrum of empirical knowledge (see [10]).

3.2 Initial Considerations for an Experiment

The starting point of the experiment is the hypothesis underlying the work in [23, 7]: the application of static type systems reduces the development time of programming tasks.

Here, especially the study in [23] has quite a mature experimental design where the subjects solved programming tasks in a programming language with static type checks as well as in a programming language without such static checks. However, two main points were unclear for the author of the here introduced paper. First, is is unclear how far the fact that existing languages were used in the experiment influenced the results. It is possible that different qualities of development environments (IDEs, debuggers) as well as different qualities in the documentation of the programming languages influenced the results. Seconds, it is unclear whether it is possible that subjects which are already familiar with a statically type-checked version of a language can just switch to the language without such static type checks. Here, it is questionable whether "having already static types in mind" already influences the developers. This objection can be supported by the fact that both languages in the

[1] A more detailed motivation for, and discussion of empirical studies and their implications can be found in [10].

experiment in [23] required type declarations in the code – hence, it seems as if the programmers' mental models with respect to type systems were the same.

Hence, a first consideration for a new experiment is to provide a new programming language with the same tool support and the same documentation to subjects. Next, instead of trying to teach both versions to the same set of subjects, the idea is to teach subjects only one of both languages, i.e. subjects which should solve the programming tasks using the dynamically typed language should not even know the statically typed language (and the other way around). A resulting problem of this approach is, that the experimental design is quite limited (because the same subjects are not tested twice) and as a consequence, the results possibly depend on the different qualities of the subjects in the two groups (statically and dynamically typed groups). In order to reduce this problem the size of both groups should be high.

One further problem is that a programming task description that is not precise enough has the problem that different subjects understand the tasks in different ways. In order to reduce the problem, it seems desirable to provide the task description in a formal way which does not permit to leave any space for misinterpretation. Consequently, it cannot be argued that differences in the solutions are the result of possible misunderstanding of the programming tasks.

The next consideration is related to the way how it should be determined how development time differs. Here, different forces seem to have a direct impact. First, time and quality seem to be two facets of the same problem: the more time developers have, the more time they can spent on design decisions, code improvements, testing, etc. Hence, it seems intuitive that more time also increases the quality of the resulting software. Second, the term quality consists of different facets such as maintainability, readability, extensibility etc. as well as the number of implemented functional requirements.

Concerning the problem of dependent time and quality, it is desirable to design an experiment in a way that one of both variables is fixed. While it is clear how to fix time, it is not obvious how to fix the quality of the solutions.

The problem with maintainability, readability and extensibility is that no stable metrics are available for them which makes it hard to come to an objective decision about how large the difference of the quality between one program and another one is. A possible solution could be to perform code reviews. However, code reviews have the problem of " subjective code reviewers" - a problem that only can be addressed by having a larger number of independent code reviewers. A larger set of code reviews for each program requires a statistical analysis for each single program in order to determine " the average review of the program" before having the possibility to compare different programs. This also has some potential risks in cases where different reviewers have completely different opinions about the quality of a program. From the practical perspective, having a large number of code reviews (with the previously explained potential risk) was considered to be not adequate for the experiment. Due to this, a decision for the experiment was to do no code reviews, but to test only the number of implemented functional requirements via test cases.

Hence, since the way how time and quality should be measured is determined, it needs to be determined which of the variables should be fixed. The idea of the experiment was to have both measurements in the same experiment. For the whole experiment, a fixed amount of time should be given to the subjects in order to determine how many test cases the subjects were able to fulfill within the time. Additionally, for a subtask of the experiment, a set of test cases should be used in order to determine how long subjects needed to fulfill these test cases.

It is important to note that if we are interested in measuring differences in the number of passed test cases (by fixing development time) then we have to make sure that the programming task given to the subjects cannot be solved by the subjects completely – otherwise no difference in the number of passed test cases can be observed. Hence, the task must be hard enough or (from the other perspective) the time to solve such a task must be small enough.

4. Experiment

The introduced study is a relatively large study with 49 subjects and required for each subject approximately 27 hours pure development time. Including the teaching time for each subject, between 43 and 45 hours working time was required per subject.

While the recording of the experiment permits a detailed analysis after the experiment took place, the focus of the experiment is on the development for two tasks: a parser and a scanner.

Section 4.1 gives an overview of the whole experiment. Section 4.2 briefly describes the programming language and its IDE used in the experiment. Section 4.3 describes the experiment setting, section 4.4 describes the measurement and 4.5 briefly discusses the validity of the experiment.

4.1 Experiment Overview and Design

In the experiment, 49 subjects (undergraduate students, selected using convenience sampling [27]) were asked to write a simplified Java parser (for Mini Java). The subjects had already passed fundamental Java programming courses as well as fundamental courses on formal languages. The specification of the simplified Java parser to be written was delivered via a context-free grammar. Furthermore, the subjects were asked to start their task by developing a (simplified) scanner that removes special characters from a word to be tested in the parser and that groups characters as tokens. For both, the scanner and parser, we expected a simple interface: the scanner (class JScanner) required a method `scan` with one parameter that receives the string to be scanned and which returns a list of tokens. The Parser (class JParser) required a method `parse` which receives a string as a parameter and which returns a boolean value that determines whether or not the passed string represents a word of the language defined by the grammar.

The subjects were divided into two groups. Each group was trained in a programming language written for the experiment. The only difference between both groups was, that one had a language with a static type system while the other one used a dynamic type system. After training, each subject had a fixed amount of time of 27 hours to

implement the parser. The scanner implementation was part of these 27 hours. The 27 hours working time were controlled: the subjects could only work on their implementation within supervised rooms of our institute.

According to this description, the underlying experiment follows a 2x2 design where the programming language is one variable and the task is another one. Both variables (programming language and task) have two treatments. For the programming language, the two treatments are the language with and without the static type system, for the tasks we have the treatment scanner and parser.

Furthermore, the experiment was designed as a between-subject design (see for example [20]), since a subject did not perform the same tasks twice (nor changed between the language). I.e. the comparison is done between different subjects (and not between the subjects themselves). The reason for this design choice was mainly motivated by the fact that once a type system is being taught, we assume that people already ″ think in term of this type system″ , i.e. even removing the static type system from the language does not make these people ″ dynamically typed programmers″ (see discussion in section 3).

4.2 Programming Language

While other researchers (such as for example [23]) use existing programming languages and IDEs within their experiments, we decided to implement a new language with a corresponding IDE just for the purpose of the experiment (and following experiments).

Our motivation for this is, that we wanted to exclude any influence on the experiment caused either by subjects who already know the language or which already have practical experience with the IDE. In that way we also exclude the effect of different qualities of documentations between different languages: different languages have different sources of information such as handbooks and web forums, so that differences in the measurement of an experiment might not be (only) the result of the language features being studied, but also the result of different qualities of information about the language. Consequently, since a new language was used in the experiment, the language was being taught within the experiment.

We wrote a new object-oriented programming language (with some similarities to the programming languages Smalltalk, Ruby and Java) called Purity in two versions: a statically typed as well as a dynamically typed version.

Purity is a simple class-based, object-oriented language with single implementation inheritance and late binding. Corresponding to the language design of Smalltalk, Purity does not distinguish between primitives (such as Integer or Boolean with corresponding operations) and objects. Instead, all elements of the language are objects which have a class definition with corresponding methods.

The dominating language construct of Purity is blocks (which is the Smalltalk version of closures, see [9]) which are used for different purposes such as conditional statements or loops. As a consequence, Purity contains few language constructs. Similar to the language design of Smalltalk, `if` is not a language construct itself, but is provided by corresponding methods in class Boolean. The only loop provided is the while-loop which is represented by a method in block objects. Field access is only provided via methods, i.e. only an object itself is allowed to access its fields. More mature features such as multi-threading or GUI support are not implemented in the language.

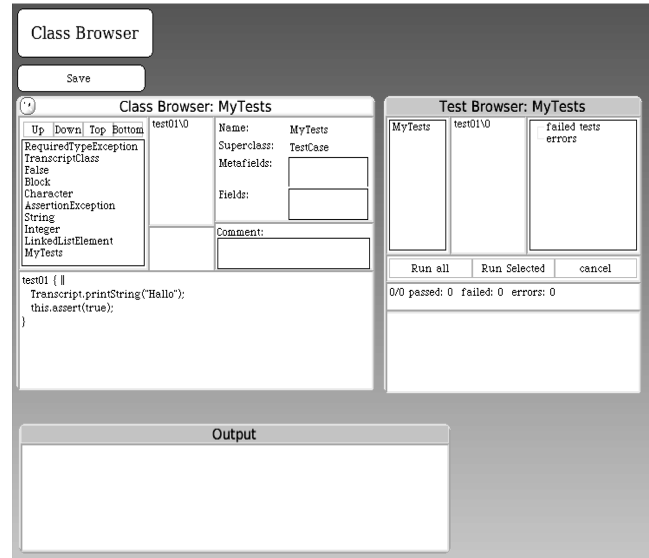

Figure 1. Snapshot of Purity IDE (Language & IDE used in Experiment).

The programming language includes a simple API with 14 basic classes such as Integer, Boolean, Block, String or LinkedList and five additional helper classes (different kinds of Exceptions). Additional to the programming language, a small IDE was provided, which includes a class browser, a test browser (for running the application and tests of applications) and a console window.

Figure 1 illustrates Purity's IDE (for the dynamically typed Purity). On the left hand side, there is a Class Browser which shows all classes, methods, fields, and superclass. The class browser also contains an editor for editing methods. On the right hand side, there is the test browser which shows all test classes and their test methods. The test browser permits to run a test (the only way to executed code in Purity) and shows failures and errors caused by a test case. The output window represents the console, which permits developers to print out strings. While developers are allowed to open as many class browsers as they like, only one output window and one test browser was allowed.

The static type system of the statically typed Purity is non-generic and nominal and can be compared to the type system of Java up to version 1.4 (without primitive types): all classes implicitly represent a type, whereby a type consists of its name and its associated methods. In contrast to e.g. Java, Purity does not provide any access modifiers (such as public, private, etc). Hence, it is not possible to exclude any methods from its static type. Since the type system is non-generic, elements of LinkedList (as the only available collection in Purity) are accessed by the static type Object. The static typing of blocks was implemented similar to the proposal that can be found in [9].

The static type checks are (for the statically typed version of Purity) performed just before the developer tries to execute a piece of code. Static type errors are shown in

an additional window which lists all static type errors in the code in addition to where these typed errors occur (of course, such a window is not available in the dynamically typed version of Purity). The static type check is performed on the whole code base. Hence, a type error in one single method (even though it is not the method currently edited by the developer) prevents the developer from executing the code.

```
//simple method for dynamically typed Purity //
myMethod { in          // method definition with parameter in //
 |locVar|    // declaration of local variable //
 locVar := in;          // sets local variable to in //
 [ || 10.largerThan(locVar);].whileFalse([  // loop from 1 to 10 //
   |blockVar|           // declaration of local block variable //
   locVar:=locVar.plus(1); // increases local variable //
   blockVar:=locVar;  // assignment to block variable //
   Transcript.printString(blockVar.asString());  // prints value //
 ]);
 ^42;                   // returns 42 //
}

//simple method for statically typed Purity //
Integer myMethod { Integer in  // method with int parameter  //
 |Integer locVar|                    // declaration of local variable //
 locVar := in;                       // sets local variable to in //
 [ || 10.largerThan(locVar);].whileFalse([// loop from 1 to 10 //
   |Integer blockVar|  // declaration of local block variable //
   locVar:=locVar.plus(1); // increases local variable //
   blockVar:=locVar; // assignment to block variable //
   Transcript.printString(blockVar.asString()); // prints value //
 ]);
 ^42; // returns 42 //
}
```

Figure 2. Example code for dynamically typed and statically typed Purity

It should be emphasized that the whole API of the statically typed Purity is the same as the API of the dynamically typed Purity (except the type annotations in the code). The only semantic difference between the statically typed and dynamically typed Purity is the existence of a type cast operator in the statically typed Purity.

Figure 2 illustrates a basic method definition for the dynamically typed as well as the statically typed Purity. In both cases the method receives an Integer object as input parameter and stores it to the local variable locVar. Then, a block (without any parameter and without any local variable) represents a loop invariant that checks whether locVar is smaller than 10. The loop's body is represented by a corresponding block passed to the method whileFalse. The block has no parameter and one local block variable (blockVar). The loop body increases the variable and prints it out.

The teaching material used for Purity is the same for the statically typed and the dynamically typed version. The only difference is, that the teaching material for statically typed Purity contains a section about the type system with a special focus on blocks.

4.3 Experiment Execution

49 subjects (undergraduate students from the University of Duisburg-Essen, selected using convenience sampling[2]) participated in the experiment as subjects. They were not aware of what was being studied within the experiment (they were told that it was an exploratory study on how students program). None of the subjects ever wrote a parser or a scanner.

First, an interview with the students was performed that checked their background. The motivation for this was, that we wanted to have balanced groups, i.e. the statically as well as the dynamically typed group should have a balanced number of experienced and unexperienced developers. Thereto, we asked for their current programming skills, their known programming languages and how long and in what environment they worked already in industry.

The whole study was not performed in a single session. Instead, the process of selecting students and dividing them into 2 groups was distributed over one year (with six sessions).

The programming language was taught to the subjects for the dynamically typed programming language in 16 hours including pure presentations, as well as hand-on sessions to the subjects. The statically typed programming language was taught in 18 hours – the difference in the time can be explained by the additional effort for teaching the static type system, where a special focus of the training was on blocks (since the subjects all had the background of Java).

After that, each subject had to implement the scanner and parser within 27 working hours divided into 4 working days. The corresponding context-free grammar was described by a Backus-Naur notation.

These 27 hours were controlled and took place in the experimental environment. Coffee-breaks etc. were not included in these 27 hours. The subjects were permitted to arrange their time freely, i.e. they were permitted to arrive at different times. The subjects were not permitted to take any material from the experimental environment. I.e. the language itself and its IDE, handbooks, etc. always stayed in the experimental environment. Furthermore, the subjects signed a paper which obliged them not to share any information with other subjects.

4.4 Recording and Measurement

While the experiment was running, all changes in the code base were logged so that it was later on possible to reconstruct all user inputs at a certain point in time: whenever the developer added or removed a class or whenever a method was added to (or removed from) the system, a corresponding log entry was generated in the background. A change in the code base was the moment, when developers edited a method and stored it in their environment.

Based on that, we measured the time the subjects required to implement a scanner and the quality of the

[2] Convenience sampling describes the construction of a sample from a population which is close at hand (see [27] for a more detailed discussion).

resulting parser. I.e. the measurement of the experiment consists of two different metrics: time and quality.

For the scanner implementation we defined 23 test cases that represented from our point of view the minimal functionality required in the system. These test cases were not delivered to the subjects but only used for evaluation purposes later on. Hence, subjects were not able to use the definition of test cases in order to increase their development speed.

Based on the log entries and the test cases we were able to determine the point in time when subjects fulfilled these test cases. We did that by reconstructing all developer activities and after any change in the code we rerun the tests. We considered the point in time when all test cases were fulfilled as the reference point where the developer finished a minimal scanner. Once, we have the development times for the statically and the dynamically typed group, we can compare them via a corresponding statistical analysis.

In order to measure the quality of the resulting parser we defined a set of 200 test cases that represented valid words of the language defined by the grammar. Additionally, we specified 200 invalid word. Hence, fulfilling 50% of the test cases has the same quality as a parser which returns the constant true or false. The test cases were not delivered to the subjects. We ran the test cases on the delivered project of each subject. Hence, we get for all subjects a number of successful test cases and can compare the number of successful test cases for the statically as well as for the dynamically typed group.

4.5 Threats to Validity

According to the known guidelines for empirical research in software engineering (see for example [16, 17]), it is necessary to make explicit all elements that potentially threaten the validity of an experiment.

First, it is necessary to emphasize the argumentation of the measurement: the paper compares development times between two groups using a different programming language. While we think that this is valid for comparing both groups, we doubt that the measured development times for the scanner or the number of successful test cases for the parser are representative for any other scenario: since the programming language being used is a new one, it is unclear how much of the development time is spent on handling a new technique instead of solving the problem. Consequently, neither the development times for the scanner nor the number of test cases can be considered representative for the same tasks using a known language. Nevertheless, the study has not the focus on the development time – its focus is on the difference of development times.

The study uses students as subject and it is often argued that students do not represent a valid sample for developers (see [24] for further discussion). It could be argued here, that one characteristic of the experiment is, that a new language must be learned. Here, it is unclear whether students or professional developers (or any of them) have a benefit (see further [3]). Concerning the students, it is necessary to state that none of them already implemented a scanner or a parser. Consequently, the measured development time does not only contain the "translation of the programming task into code" but also the "intellectual effort to design a solution".

Another threat to validity is, that the subjects were grouped into two groups based on interviews. It is possible that the estimation of the person, who did the experiment, was wrong concerning the developers' experiences.

Next, it can be argued that 16 hours training is not enough for learning a new language (and a new IDE). This is definitively the case. However, it should be emphasized that the language, its API as well as its IDE was kept very simple, so that we considered the training to be sufficient. The only problematic element we identified was that the subjects had problems to understand the semantics of blocks due to the Java background of the subjects. However, since this problem was equal for all subjects, we consider it to be less problematic for the internal validity of the experiment.

Concerning the additional two hours for the type system, it can be argued that two hours are not sufficient to learn the type system of a language. Here, it needs to be emphasized that all subjects already had a Java background. Consequently, they were already familiar with nominal type systems. The new element for the subjects were blocks, hence the training of the static type system mainly focused on the static typing of blocks.

Dyn. Typed		
Subject	sec	hours
1	8141	2.26
2	38374	10.66
3	31275	8.69
4	24666	6.85
5	7260	2.02
6	11224	3.12
7	23218	6.45
8	18317	5.09
9	7335	2.04
10	16994	4.72
11	12766	3.55
12	13975	3.88
13	39123	10.87
14	23041	6.40
15	5430	1.51
16	2851	0.79
17	8379	2.33
18	7510	2.09
19	18334	5.09
20	42501	11.81
21	30393	8.44

Stat. Typed		
Subject	sec	hours
22	40970	11.38
23	32208	8.95
24	23413	6.50
25	28463	7.91
26	51572	14.33
27	32112	8.92
28	28752	7.99
29	29509	8.20
30	56985	15.83
31	8972	2.49
32	3653	1.01
33	40864	11.35
34	37022	10.28
35	14701	4.08
36	46032	12.79
37	20949	5.82
38	26612	7.39
39	25872	7.19
40	11172	3.10
41	16464	4.57
42	6305	1.75

Figure 3. Measured experiment results for scanner

One serious argument against the validity is, that the programming task is a relative large one. Although this can be seen as a good approach to reduce the often mentioned argument, that empirical studies are typically too small in order to get any meaningful results (see [25]), it has the disadvantage that possible effects such as design choices of programmers become an influencing factor of the experiment. We do not think that it is possible to reduce this problem: either an experiment has a relative trivial task where developers do not have enough freedom to do relevant design choices, or it has not. In the first

case, the programming task can be reduced to the typing speed of the programmer. The latter one inherently contains such design choices.

Concerning the tasks, it is problematic that it was not explicitly requested from the subjects that they have to finish the scanner completely before continuing with writing the parser. As a consequence, we cannot be sure at what point in time the subjects switched to the parser task without finishing the scanner task. The experiment addresses this threat by using only a minimal set of test cases for the scanner – assuming that developers do not even want to finish before working on the parser task.

One further concern we see is our decision to use black-box testing via test cases as the criterion for the quality of the resulting applications. As a consequence, we consider a piece of software that has a high number of passed test cases to be better than a piece of software with a lower number of passed test cases. This implies, that a very good implementation of a parser, where the developer has misunderstood the start production rule of the grammar, is considered to be bad. A different alternative to address this problem could be to do code reviews by experts that judge on the quality of the resulting pieces of code. We discussed this issue intensively in section 3 where our main argument is that for this experimental setting the use of code reviews is rather not practical (see discussion of initial considerations in section 3.2).

5. Analysis of Scanner Task

We start the analysis by describing the experiment's results for the scanner implementation and then performing significance tests in order to check whether there are significant differences in the development times using the statically typed and dynamically typed programming language.

The data used here comes from 49 subjects, 25 in the dynamically typed group and 24 in the statically typed group. 4 subjects from the dynamically typed group and 3 subjects from the statically typed group failed to fulfill the scanner task (while some of them still were able to finish the parser task). Hence, we removed those seven subjects for the analysis of the scanner. Hence, the analyzed data within this experiment comes from two groups of equal size: the group with the dynamically typed language as well as the group with the statically typed language consisted of 21 subjects.

5.1 Results and Descriptive Statistics

Figure 3 shows the measured results of the experiment, i.e. the number of seconds (and hours) required by each subject to fulfill the test cases. For example, on the left hand side, the third line says that subject number 3 required 31275 seconds (respectively 8.69 hours) in order to fulfill the minimum requirements. Subject number 24 (which is a subject that used the statically typed language) required 23413 seconds (respectively 6.50 hours).

Based on the experiment results, we computed the descriptive statistics (see Figure 4). Here, we see that the sum of development times (and hence the arithmetic mean), the maximum, the minimum and the median of the group with the dynamically typed programming language is less than the corresponding values of the group with the statically typed programming language. Furthermore, we see that the differences are quite high (for example, the sum of development times of the dynamically typed solution is 391,107 seconds, while the sum of times for the statically typed solution is 582,602 seconds).

	sum (sec.)	max (sec.)	min (sec.)	arith. mean (sec.)	median (sec.)	std. dev.
Dyn. Typed	391107	42501	2851	18624	16994	11726
Stat. Typed	582602	56985	3653	27743	28463	14256

Figure 4. Descriptive statistics for scanner task

In order to have a more intuitive representation of these differences, Figure 5 shows a box plot for the dynamically typed and the statically typed group, which visualized the lower quantile, the median and the upper quantile of the underlying data set.

Because of these results it seems reasonable to assume that there is a significant difference between both times which will be checked in the next section.

5.2 Checking Significance in Means

In order to check whether there is a significant difference between both values it is necessary to formulate a hypothesis which needs to be checked by a corresponding significance test.

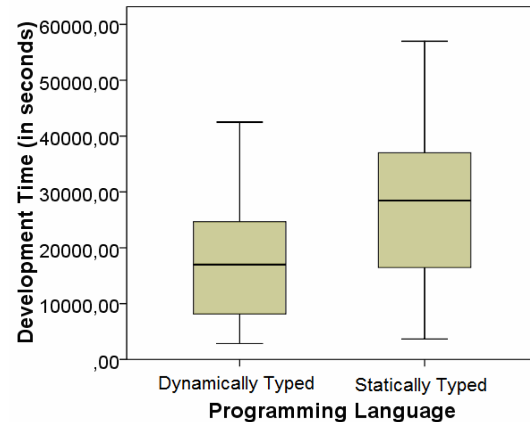

Figure 5. Box plot for scanner task

Since the subjects differ from each other with respect to the applied programming language (no subject did the implementation with the statically and the dynamically typed language), a significant test for independent samples is required.

At the current point, we cannot assume any underlying distribution for the measured data. Hence, it is necessary to apply a so-called non-parametric significance test (cf. [4]). A test that can be applied here is the (non-parametric) Mann-Whitney U-test which compares for two samples whether they come from the same population. The Mann-Whitney U-Test is a standard significance test in statistics and empirical methods (see for example [25] and [13]) for the here described purpose.

The underlying hypothesis and alternative hypothesis to be tested are

H0: The data for statically and dynamically typed development times are drawn from the same population

H1: The samples come from different populations

		N	Mean Rank	Sum Ranks
Language	Dyn. Typed	21	17.62	370
	Stat. Typed	21	25.38	533

Figure 6. Ranks (Mann-Whitney U-Test) for scanner test (with p=0.04)

Since the underlying distribution is not known, the Mann-Whitney U-Test works on ranks, i.e. the number of the development times for the statically typed as well as dynamically typed scanner become ordered and the number of positive and negative ranks for the statically typed and dynamically typed development times are computed. Based on the ranks, the U-Test computes a p-value which indicates whether the result is significant according to the underlying. The smaller the p-value the "more significant" is the rejection of the hypothesis: typically, p-values of 0.01, 0.05 or 0.1 are used to decide whether the null hypothesis can be (significantly) rejected, i.e. a result of p=0.02 means a significant rejection of the null hypothesis under the significance level of 0.05. In case the null hypothesis is rejected, the rank sum for each approach indicates whether the statically typed or dynamically typed programs dominate the other approach.

The p-value for the Mann-Whitney U-test is p=0.04. Since $p < 0.05$, the null hypothesis can be rejected, i.e. both samples come from different populations. By comparing the rank sums it turns out that the development times using the dynamically typed language are significant lower than the development times using the statically typed programming language: the rank sum for the dynamic typed languages is smaller than the rank sum of the statically typed language (370 vs. 533, see Figure 6).

5.3 Checking Significance for Quantiles

A possible explanation of the previous result could be that in the statically typed group the number of underperforming subjects is larger than in the dynamically typed group. According to [19] it is quite common to divide the subjects into different quantiles and to perform the statistical analysis in separate in order to reduce such effect.

We constructed the 2-quantiles from the experiment results, i.e. all subjects whose development time is smaller than or equal to the corresponding median are in the first quantile, all subjects whose development time is smaller in the second quantile. For the dynamically typed language, subjects 16, 15, 5, 9, 18, 1, 17, 6, 11, 12 and 10 are in the first quantile, for the statically typed language, subjects 32, 42, 31, 40, 35, 41, 37, 24, 39, 38 and 25 are in the first quantile.

Figure 7 illustrates a box plot for the resulting quantiles (we neglect here to give the corresponding descriptive statistics). It shows that the median for the first dynamically typed quantile is lower which is also true for the second quantile.

Performing the Mann-Whitney U-Test for each quantile, i.e. comparing the first dynamically typed with the first statically typed quantile as well as comparing the second dynamically typed with the second statically typed quantile reveals in both cases a significant difference (with p=0.03, respectively p=0.04).

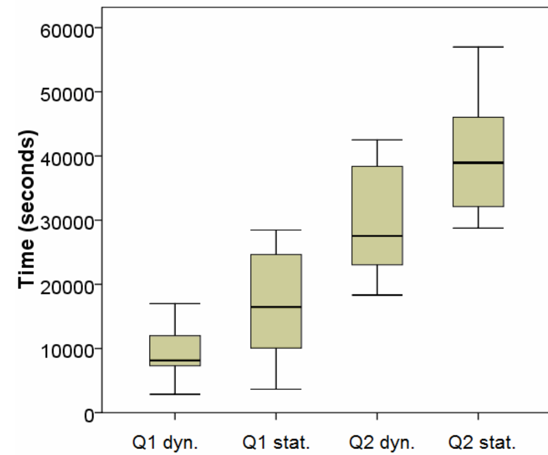

Figure 7. Boxplot for 2-quantiles (scanner task)

Since there are significant results for both quantiles, this strengthens the argument that the significance determined in section 4.1 cannot be explained by a different number of outperformers or underperformers in each group.

5.4 Comparing Difference in Means

While the previous section determined that there is a significant difference in the development times of statically and dynamically typed solutions, it did not state how large this difference is (note that the use of the arithmetic mean or the median is statistically not valid here). A typical approach here is to determine first the data's underlying distribution and to use (based on the result) a corresponding significance test.

Figure 8 illustrates a histogram for the typed language group. Furthermore, the diagram illustrates a plot of the normal distribution. According to the histogram it seems reasonable to test, whether the underlying data is normally distributed.

A check for normal distribution is done using the Shapiro-Wilk test (cf. [4]) which computes a value p. In case p is smaller than 0.05, the test rejects the assumption that the underlying data is normally distributed (under a significance level of 0.5).

Computing the p-values reveals for the dynamically typed programming language as well as for the statically typed programming language a value larger than 0.05 (although the p-value for the dynamically typed language is 0.08). Hence, in both cases the test does not reject the hypothesis of a normal distribution of the underlying sample. Therefore, we can perform a t-test for independent samples which will also reveal the size of the difference between both samples (although it must be critically

mentioned that a value p=0.08 is only not significant under a significance level of p=0.05).

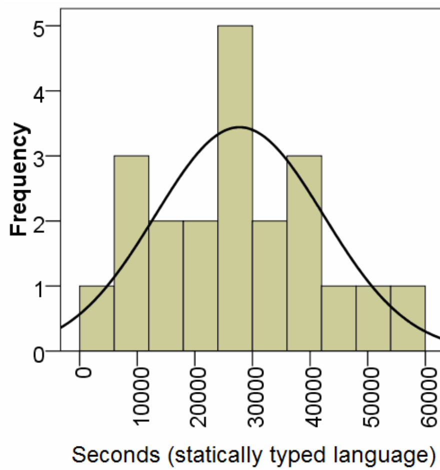

Figure 8. Histogram (statically typed language) for scanner task

Figure 9 shows the results of the t-test. Again, we can see that there is a significant difference between both samples (with p=0.03), but we have this information already from the previous section. However, the important information is the 95% confidence interval which states that with the probability of 95% the development time using the dynamically typed language is between 777 and 17461 seconds less than the development time using the statically typed language.

	t	df	sig (2-tailed)	mean diff.	95% confidence interval	
					lower	Upper
Language	-2.209	40	0.03	-13140.3	-17461	-777

Figure 9. t-test results for scanner task

According to this, the advantage of using the dynamically typed programming language in the experiment is approximately between 19 and 291 minutes per subject – but (again) it needs to be emphasized that the value for the Shapiro-Wilk-Test for the untyped language was p=0.08 (which is formally correct that the hypothesis of the normal distribution cannot be rejected, but which obviously tells, that the distribution is very close to be non normal).

5.5 Impact of Testing / Type Error Fix Time

The previous results are slightly surprising, since it seems reasonable to assume (by relying on the argumentation pro typing as given in section 1) that typing should have in the experiment a positive impact on the development time:

- for all subjects the programming language and its underlying IDE was new. It could be assumed here that the additional documentation character of type declarations eases the application of a new language
- being new in a language could mean that developers do rather trivial mistakes (such as sending wrong messages to objects). In such a case the type system saves developers from wasting run-time on testing applications that (predictably) cannot be executed due to type errors.

However, since the previous results contradict the implication of this argumentation it is reasonable to analyze how the data for the scanner implementation times is related to this argumentation. While the first argument (documentation characteristics) cannot be measured from the experiment's data, it is possible to define a corresponding measurement for the second one. The second argument (reduction of test time) implies that the correction of a type error takes less time than running a test case, receiving a NoSuchMethod exception and correcting the corresponding piece of code. In order to check this argument, we reconstructed from the logged data the test runs performed by the subjects.

For the dynamically typed subjects we measured the time from the start of a test run that stopped with exceptions that could be handled by a type systems (non-declared variables, NoSuchMethod exceptions, but not NullPointer exceptions) until the next test run successfully ran. Then, we built for each subject the sum of these test times.

For the typed subjects we measured the time from the point in time where subjects tried to run a test case whose execution was prevented by the static type checker until the next test ran that was accepted by the type checker.

Dyn. Typed		
subject	sec	minutes
1	942	15.70
2	2007	33.45
3	6857	114.28
4	2588	43.13
5	896	14.93
6	2780	46.33
7	7379	122.98
8	1873	31.22
9	405	6.75
10	1465	24.42
11	2658	44.30
12	2034	33.90
13	1600	26.67
14	2681	44.68
15	587	9.78
16	393	6.55
17	1832	30.53
18	3759	62.65
19	2538	42.30
20	4662	77.70
21	350	5.83

Stat. Typed		
subject	sec	minutes
22	6482	108.03
23	5310	88.50
24	1199	19.98
25	3425	57.08
26	5376	89.60
27	3689	61.48
28	5902	98.37
29	3138	52.30
30	6714	111.90
31	339	5.65
32	33	0.55
33	4720	78.67
34	4096	68.27
35	1589	26.48
36	3847	64.12
37	585	9.75
38	614	10.23
39	1324	22.07
40	645	10.75
41	2810	46.83
42	98	1.63

Figure 10. Measured (approximated) debugging time for scanner task.

Under the assumption that developers directly try to solve errors in the code that cause exception, the first case identifies the times people require to fix no such method exceptions in the code (which could have been handled by the type checker). The latter one measures the time for people to fix their type errors identified by the type checker. Again, we built the sum of these times for each subject.

It should be noted that this measurement represents only a rough approximation of the debugging time for each subject, because it was not explicitly required that subjects directly fix a bug that was found during testing (or by the type checker). Hence, the measurement is build upon the assumption that once an error occurred, developers directly try to fix it – the measurement is only an index for the time required to fix a bug.

Figure 10 represents the results of the measurement. A first glimpse already shows that the measured data does not reveal similar large differences between the error fix time and the measurement of the development times from the previous section. Figure 11 shows the descriptive statistics for the measurement. It shows that the arithmetic mean of statically and dynamically typed programming language are rather closely related.

	sum (sec.)	max (sec.)	min (sec.)	arith. mean (sec.)	median (sec.)	std. deviation
Dyn. Typed	50286	7379	350	2395	2007	1997
Stat. Typed	61935	6714	33	2949	3138	2113

Figure 11. Descriptive statistics for approximated error fix time

Performing again a Mann-Whitney U-test on the results supports that impression: the resulting value p=0.49 does not permit to reject the null-hypothesis stating that both samples come from the same distribution. Hence, no significant difference in both samples could be measured.

This result is slightly surprising. Having no significant difference in the debug time between dynamically and statically typed group means that the effort for fixing type-related bugs (NoSuchMethod exceptions, non-declared variable) can be assumed to be the same. Although this falsifies the statement that debugging typing errors takes less time having a static type system (and a static type check), this does not explain why the dynamically typed group requires less time for the same task.

A possible explanation for this phenomenon could be, that the dynamically typed group gets more information about the correctness of the application from their testing: instead of knowing only that no NoSuchMethod exceptions occur within the code, they already get some earlier preliminary results from the testing which helps to determine what parts of the application are already correct. However, more detailed studies are necessary to study this phenomenon in more detail.

6. Analysis of Parser Task

While section 5 analyzed the data for the scanner implementation, we now analyze the number of successful test cases for the statically and dynamically typed solutions. The performed techniques are identical to the techniques used in the previous section. Hence, we do not explain in detail each step.

6.1 Checking Differences in Means

Figure 12 shows the percentages of passed test cases for each subject. Here, in contrast to section 5, we include all subjects (the reason they were removed in the last section was, that no time could be measured for them).

We see that a large number of subjects fulfil only 50% of the test cases – these are those subjects who did not achieve a meaningful parser, since they were able to reject strings that are not words of the grammar, but they were not able to detect any of the other ones (or the other way around). In both cases, we see a slight difference in number of subjects without a meaningful result: for the dynamically typed language 14 subjects and the statically typed language 11 subjects. Hence, it seems that more people with the dynamically typed language failed to provide a meaningful parser. However,. we think that 14 and 11 is “comparable”.

Dyn. Typed	
Subject	% succ TestCases
1	50%
2	90%
3	50%
4	50%
5	50%
6	56%
7	80%
8	60%
9	50%
10	52%
11	50%
12	50%
13	50%
14	50%
15	50%
16	56%
17	69%
18	92%
19	85%
20	50%
21	50%
22	85%
23	50%
24	79%
25	50%

Stat. Typed	
Subject	% succ TestCases
26	50%
27	50%
28	75%
29	87%
30	80%
31	50%
32	50%
33	68%
34	50%
35	100%
36	50%
37	85%
38	50%
39	50%
40	55%
41	50%
42	89%
43	68%
44	86%
45	57%
46	50%
47	98%
48	50%
49	51%

Figure 12. Measured experiment results and descriptive statistics.

Again, we perform the Mann-Whitney U-Test in order to compute the p-value. However, the p-value for the table above is p=0.40. Hence, there is no significant difference between the statically typed and the dynamically typed version.

Removing those subjects that have only 50% accepted test cases (i.e. we repeat the test with 12 subjects using the dynamically typed language and 11 with the statically typed language) and repeating the Mann-Witney U-Test gives a p-value p=0.60. Hence, also no difference in both groups was determined.

Performing a test on grouped subject, i.e. we consider only one the outperforming and underperforming subjects gives the p-values p=0.60 (outperforming subjects) and p=0.31 (underperforming subjects).

However, it seems also reasonable to analyze the outperforming and underperforming subjects that have more than 50% successful test cases. However, it should be noted that this test is problematic, since the group of outperforming, successful subjects consists only of 6 subjects. However, even taking into account that a possible

significant difference would be problematic to interpret, we do not find a significant different for the outperforming subjects (p=0.33). For the underperforming subjects we find also no statistical significant difference for a significance level of 5%. The p-value is p=0.91.

Dyn. Typed		
subject	sec	minutes
1	1287	21.45
2	3075	51.25
3	6533	108.88
4	1991	33.18
5	6857	114.28
6	2588	43.13
7	896	14.93
8	4622	77.03
9	7379	122.98
10	1873	31.22
11	405	6.75
12	1470	24.50
13	3417	56.95
14	4250	70.83
15	1600	26.67
16	2861	47.68
17	1604	26.73
18	709	11.82
19	2950	49.17
20	3430	57.17
21	5012	83.53
22	2950	49.17
23	2626	43.77
24	4662	77.70
25	350	5.83

Stat. Typed		
subject	sec	minutes
26	7091	118.18
27	3901	65.02
28	4588	76.47
29	5313	88.55
30	3912	65.20
31	2313	38.55
32	5271	87.85
33	8500	141.67
34	4741	79.02
35	3416	56.93
36	4763	79.38
37	2715	45.25
38	7344	122.40
39	5266	87.77
40	3754	62.57
41	777	12.95
42	3900	65.00
43	3637	60.62
44	4006	66.77
45	2804	46.73
46	5881	98.02
47	3045	50.75
48	4029	67.15
49	2180	36.33

Figure 13. Measured (approximated) debugging time for parser task.

6.2 Debug Times

Again, we repeated the measurement of the debug times in the same way we did as in section 5 under the assumption that developers try to solve their exceptions directly: the measured times for the dynamically typed group describes the debug time of those exceptions that could have been handled by a type checker, the measured time for the typed developers describes the type required to handle typing problems identified by the type checker. Figure 13 describes the results of that measurement. Performing again the Mann-Whitney U-Test on the results reveals something remarkable: with p=0.01 the time for the dynamically typed group is less than the time for the statically typed group.

7. Conclusion

In this paper we presented results of an experiment that explores the impact of static and dynamic type systems on the development of a piece of software (a parser) with 49 subjects. We measured two different points in the development: first, the development time until a minimal scanner has been implemented, and second the quality of the resulting software measured by the number of successful test cases fulfilled by the parser. In none of these measured points the use of the static type system turned out to have a significant positive impact. In the first case, the use of the statically typed programming language had a significant negative impact, in the latter one, no significant difference could be measured.

One interesting point in the study is that we rather expected that developers benefit from the statically typed language, especially, because the language in use was new to them. Hence, we rather expected a positive impact of typing because it is commonly assumed that static typing improves the ability to learn new APIs – since static typing restricts a possible faulty use of a new API. We analyzed this by defining as (approximated) debugging time on the experiment data and performing a corresponding significance test. However, the result has been rather surprising: while in the small task no significant different could be measured, in the latter one the statically typed group required significant more time for fixing the type errors. Although it seems obvious that this should have been led to a worse result in the parser development, in fact it was not. This implies that there must be a positive impact of static typing in something else which compensates this negative effect. Possibly, it is easier for developers to extend their own code, if static type information is added and this effect is larger than a negative effect of fixing type errors. But this is rather a speculation and requires much more exploratory studies in the area of usability of static type systems.

It must not be forgotten that there are some threats to validity that threaten the experiment's results. There are two problematic issues: the first one is related to the grouping of subjects into the statically typed / dynamically typed group, the latter one is related to the testing of the results.

A problematic point here is that it is unclear whether both groups contained an equal number of good developers. However, the classification of developers is currently a problem since no objective metrics available that permit to classify subjects: the current state of the art in software engineering does not provide until now a better classification. Hence, we built quantiles for the analysis – which is a way to handle this situation.

Concerning the testing of the results, we are based on test cases. Here, it is important to note that 50% of the (black-box) test cases (for the parser) were positive tests. It is possible that if the number of test cases would been changed, that results would have changed, too. Furthermore, white box testing could be applied which potentially represent different result. However, until now we are not aware of any better way to determine the quality of the resulting software which is also practical in such a large scale study.

Probably the most interesting point of the experiment results is that they contradict studies such as [21] where a positive impact of typing could be measured. We are currently not aware of how to explain this contradiction. Possibly, the effect of typed programs is better in some situations while it is not in others. However, since the knowledge about different "kinds of programs" is currently quite restricted, it is currently unknown whether different "kinds of programs" have a different impact on the use (and the impact) of type systems. Here, it is desirable to repeat the experiment under similar conditions but with different tasks. It should also be noted that the experiment design differs widely from the one in [21]: while the experiment in [21] uses existing languages and has possibly the effect of better / worse documentation, tutorials, etc. of these languages, the experiment in this paper has implemented a language only for the purpose of

the experiment: the language and their documentation are identical (except the difference in the static type system). A subjective opinion of this paper's author is that blocks, and especially the static typing of blocks turned out to be problematic for the subjects of the experiment – it would be desirable to study this matter in isolation.

It seems that the experiment at least does not contradict with a common believe that is often articulated in the area of type systems: that dynamically typed systems have a benefit in small projects while this benefit is reduced (or even negated) in larger projects. In the experiment, the small task was quicker implemented by the dynamically typed developers while no difference could be measured for the whole task (scanner and parser). Here, it could be argued that the possible effect of dynamically typed languages for the smaller task was compensated by the benefit of the static type system later on and that (possibly) the positive effect of statically typed languages would have been even larger, if the project would have been larger. Hence, it could be argued that the positive impact of static typing could not be measured because the project was too small.

Even though the experiment seems to suggest that static typing has no positive impact on development time it must not be forgotten that the experiment has some special conditions: the experiment was a one-developer experiment. Possibly, static typing has a positive impact in larger projects where interfaces need to be shared between developers. Furthermore, it must not be forgotten that previous experiments showed a positive impact of static type systems on development time. Possibly, such a positive impact occurs in special situations, which might depend on the "kind of programing task" - however, up to now no classification of programming tasks is known that potentially influences the topic of static type systems. Here, additional research is required, too.

A further point that needs to be emphasized is, that the experiment here addresses only pure programming time. Possible influences of static type systems on design time, readability or maintenance of software cannot be concluded from the experiment's results.

A general conclusion is that an experiment is available that doubts whether static type systems really have a positive impact on development time. However, in order to get a more detailed understanding of the impact of static type systems more studies are needed. Since static type systems play such an important role in software development it is rather tragic that the discussion about the need for statically or dynamically typed programming languages is mainly driven by personal experiences and personal references of researchers instead of empirical knowledge received from an adequate research method. From that point of view, the here introduced experiment represents a first starting point and a valuable contribution.

Acknowledgments

The author would like to thank the University of Duisburg-Essen for providing a grant that made this empirical study possible. Furthermore, the author would like to thank the companies GFOS, Swarc, Senacor, clavis, and ISKV for supporting the project. Additionally, the author would like to thank Erik Ernst, Robert Hirschfeld, and Michael Haupt for discussing with them the ideas of empirical studies in programming language research.

Finally, the author would like to thank David Ungar for his encouragements and his time he spent on long discussions with the author.

References

[1] Bruce, K.: Foundations of Object-Oriented Languages: Types and Semantics, MIT PRESS, 2002

[2] Cardelli, Luca: Type Systems, In: CRC Handbook of Computer Science and Engineering, 2nd Edition, CRC Press, 1997.

[3] Carver, J; Jaccheri, L.; Morasca, S., Schull, F. Using empirical studies during software courses, Empirical Methods and Studies in Software Engineering, Experiences from ESERNET, LNCS 2765, Springer, 2003, pp. 81-103.

[4] Conover, W. J.: Practical nonparametric statistics, 3rd edition, John Wiley & Sons, 1998.

[5] Daly, M.; Sazawal, V., Foster, J.: Work in Progress: an Empirical Study of Static Typing in Ruby, Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) at ONWARD 2009.

[6] Eckel, B.: Strong Typing vs. Strong Testing, mindview, 2003, http://www.mindview.net/WebLog/log-0025, last access: August 2009

[7] Gannon, J. D.: An Experimental Evaluation of Data Type Conventions, Comm. ACM 20(8), 1977, S. 584-595.

[8] Gat, E.: Lisp as an alternative to Java. Intelligence 11(4): 21-24, 2000.

[9] Graver, J. O.; Johnson, R. E.: A Type System for Smalltalk, Seventeenth Symposium on Principles of Programming Languages, 1990, pp. 136-150

[10] Hanenberg, S.: Faith, Hope, and Love - A criticism of software science's carelessness with regard to the human factor, To appear in Proceedings of Onward 2010.

[11] Hanenberg, S.: What is the Impact of Type Systems on Programming Time? - Preliminary Empirical Results, Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) at ONWARD 2009.

[12] Hanenberg, S. Doubts about the Positive Impact of Static Type Systems on Programming Tasks in Single Developer Projects - An Empirical Study, Proceedings of ECOOP 2010, Springer, pp. 300-303.

[13] Hanenberg, S.; Stuchlik, A.: What is the impact of type casts on development time? An empirical study with Java and Groovy, Submitted, 2010.

[14] Hudak, P.; Jones, M.P.: Haskell vs. Ada vs. C++ vs. Awk vs. .. An Experiment in Software Prototyping Productivity, technical report, Dept. of Computer Science, Yale Univ., New Haven, Conn., July 1994.

[15] Juristo, N.; Moreno, A.: Basics of Software Engineering Experimentation, Springer, 2001.

[16] Kitchenham, B. et al.: Preliminary guidelines for empirical research in software engineering, IEEE Transactions on Software Engineering, Volume 28 , Issue 8, 2002, pp. 721 - 734.

[17] Kitchenham, B. et al.: Evaluating guidelines for empirical software engineering studies, International Symposium on Empirical Software Engineering, Rio de Janeiro, Brazil, 2006, pp. 38 – 47.

[18] Lamport, L.; Paulson, L. C.: Should your specification language be typed, vol. 21, ACM, New York, NY, USA. 1999.

[19] Pierce, B.: Types and Programming Languages, MIT Press, 2002.

[20] Pfleeger, S. L., Experimental Design and Analysis in Software Engineering Part 4: Choosing an Experimental Design, Software Engineering Notes, vol 20, no 3, ACM, 1995, pp. 14-16.

[21] Prechelt, L.: Kontrollierte Experimente in der Softwaretechnik: Potenzial und Methodik, Springer-Verlag, 2001.

[22] Prechelt, Lutz: An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical Report 2000-5, March 2000.

[23] Prechelt, L., Tichy W.: A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking, IEEE Transactions on Software Engineering 24(4), 1998, S. 302-312.

[24] Svahnberg, M.; Aurum, A.; Wohlin, C.: Using Students as Subject, Proceedings of the 2nd Symposium on Empirical software engineering and measurement, Kaiserslautern, Germany, pp. 288-290 .

[25] Tichy, W.: Should Computer Scientists Experiment More?, IEEE Computer 31(5): S. 32-40.

[26] Tratt, L., Wuyts, R.: Dynamically Typed Languages. IEEE Software 24(5), 2007, S. 28-30.

[27] Wohlin, C., Runeson, P., Höst, M.: Experimentation in Software Engineering: An Introduction, Springer, 1999.