# Impact of Using a Static-type System in Computer Programming*

Ismail Rizky Harlin[1*], Hironori Washizaki[1], Yoshiaki Fukazawa[1]
[1] Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo 169-8555, Japan

* Corresponding author. Tel.: +6221 8219041; email: mayharlin@fuji.waseda.jp
Manuscript submitted on January 10, 2014; accepted on March 8, 2014.
doi: ???

**Abstract:** Static-type systems are a major topic in programming language research and the software industry because they should reduce the development time and increase the code quality. Additionally, they are predicted to decrease the number of defects in a code due to early error detection. However, only a few empirical experiments exist on the potential benefits of static-type systems in programming activities. This paper describes an experiment that tests whether static-type systems help developers create solutions for certain programming tasks. The results indicate that although the existence of a static-type system has no positive impact when subjects code a program from scratch, it does allow more errors in program debugging to be fixed.

**Keywords:** static-type systems, programming language, empirical study, program debugging.

## 1. Introduction

Type systems are generally formulated as collections of rules to check the consistency of programs. This kind of checking exposes not only trivial mental slips, but also deeper conceptual errors, which are frequently manifested as type errors. A programming language can be divided into several categories as shown in Table I [1].

Table 1. Programming Language Classification

| | **Statically checked** | **Dynamically checked** |
|---|---|---|
| Strongly typed | ML, Haskell, Pascal (almost), Java (almost) | Lisp, Scheme |
| Weakly typed | C, C++ | Perl |

The traditional, simplified, definition of dynamic-type languages is that they do not enforce or check type safety at the compile-time (as opposed to a static-type language), but defer such checks until the run-time. While factually true, this definition leaves out what makes dynamic-type languages interesting—for example, they lower development costs and provide the flexibility required by specific domains such as data processing [14].

*This paper is an extended version of a paper presented at the 18th IEEE International Symposium on High Assurance Systems Engineering [7].

It should be noted that since there is no central authority defining dynamic-type languages, such languages vary greatly. Nevertheless all such languages share a great deal in common. In contrast to dynamic-type programming languages, static-type programming languages do type checking (the process of verifying and enforcing type constraints) at the compile-time as opposed to the run-time. In addition, prototype-based scripting languages (e.g., JavaScript [8]) also exist where everything belongs to a single type, but a variable declaration is still necessary.

There is a long, ongoing debate about the potential strengths and weaknesses of static- and dynamic-type systems in software development. Although many authors claim that static-type systems reduce the amount of time required to develop a program and consequently, improve software quality, others hold the opposite view.

Static-type checking allows early detection of some programming errors. Errors that are detected early can be fixed immediately, rather than lurking in the code to be discovered at a later time when the programmer may be busy with something else—or even after the program has been deployed [1]. Many experts and practitioners in Software Engineering have argued that fixing program errors, including bugs and defects, in the later stage of Software Development may require a much larger intellectual effort (and time) compared to removing them at the early stage. Moreover, when a type system is well designed, type checking can capture a large fraction of routine programming errors, eliminating lengthy debugging sessions [15]. Fareer [3] even mentioned that unit testing cannot replace static-type checking.

However, the expressive power of current static-type systems in mainstream object-oriented languages is limited. For example, although they prevent users from adding a string to a bool, they do not prevent them from accessing the first element of an empty list, creating off-by-one errors, or using null pointers. In fact, static-type systems cannot detect most common programming errors [14]. For such systems to work, developers must manually input the types during development.

In static-type systems, reasoning about a specification will easily find any errors caught by type checking. However, large specifications are seldom verified, and type checking can catch errors that would otherwise go undetected [13]. Moreover, based on the data provided by Stefan Hanenberg in his empirical study [10], at least in his experiment settings, the total debugging time of the exceptions, which could have been handled by a type checker (in static-type systems), is less than the time that must be invested to handle the type checker itself.

This paper contributes to the discussion with a controlled experiment that empirically investigates possible conditions when developer should use static-type systems and potential advantages of using such systems. The experiment in this paper is built to test the hypothesis that static-type programming languages decrease development time and consequently enable developers to create better solutions for certain programming tasks as well as debugging certain program codes. Specifically this paper examines the following two research questions:

RQ1) How do static-type systems affect the development of specific programming tasks when developers code a program from scratch?

RQ2) How do static-type systems affect program debugging?

The experiment reveals that subjects who used a static-type system had a significant positive impact for debugging tasks, especially for encryption programs with many data types. On the other hand, when developers coded a program from scratch, a significant difference was not observed between static- and dynamic-type systems. The measurements are based on the number of requirement points that are successfully achieved and the number of fixed errors.

Section 2 briefly discusses related works in the area of empirical studies on type systems. Section 3 describes the initial considerations of the experiment and programming tasks used in the experiment as well as threats to validity. Then section 4 shows the results by describing the measured data. Finally, section 5 concludes the paper.

## 2. Literature Review and Related Works

### 2.1. Definitions

#### 2.1.1. Type Systems

According to Luca Cardelli, the most obvious symptom of an execution error is an unexpected software fault such as an illegal instruction fault. The fundamental purpose of a type system is to prevent execution errors from occurring while running a program. [15]

#### 2.1.2. Typed Programming Languages

A program variable can assume a range of values when a program is executed. The upper bound of such a range is called the type of the variable. For example, variable x of type boolean is supposed to assume that only boolean values can be assigned to x. Languages where variables can be given (nontrivial) types are called typed languages. [9]

A type system is a component of a typed language that keeps track of the types of variables of all expressions in a program. Type systems are used to determine whether programs behave well. In a typed language, only program sources that comply with a type system can run. Typed languages are explicitly typed if the types are part of the syntax, and implicitly typed otherwise. No mainstream language is purely implicitly typed. [15]

#### 2.1.3. Untyped Programming Languages

Untyped languages do not restrict the range of the variables; they do not have types or, equivalently, have a single universal type that contains all values. In these languages, operations may be applied to inappropriate arguments. The result may be a fixed arbitrary value, a fault, an exception, or an unspecified effect. The pure λ-calculus is an extreme case of an untyped language where no fault ever occurs. [15]

#### 2.1.4. Static Typing

For example, in C programming language (static-typing):

```
/* C code */
char foo[] = “50”; //explicit
int j = 10;
int k = 5;
foo = foo + j; // error
j = j + k //success
```

Fig. 1. C program code

The example above uses two built-in C types: **int** (representing an integer) and Unicode character or **char arrays** (represent a string). Although this program might be expected to run, when *foo* is set to 60, the C compiler refuses to compile this code and says that the + operation is not defined between the values of the type integer and the String. In some dynamic typing languages, the above error will not prevent the program from running (or being compiled), but when the program runs, the function will raise a run-time type exception. In some other dynamic typing languages, such an error will be ignored, or automatic type conversion (implicit type-casting) will enable the program to successfully run.

Statically typed languages define and enforce types at the compile-time. Dynamic typing, at its simplest level, is when type checks are left until run-time. It is important to note that is different from being typeless. Both statically and dynamically typed languages are typed, but the chief technical difference is when the types are enforced. [14]

### 2.2. Related Works

To the best of our knowledge, only a few works are published in the area of empirical evaluations of type systems. The first one is by Prechelt and Tichy [17], which concentrates on the impact of static-type checking in procedure arguments. Their experiment suggested that for many realistic programming tasks, type checking of interfaces improves both productivity and program quality. However, in another paper [16], which compares seven programming languages, Prechelt showed that programmers who used a scripting language (dynamic-type) needed less than half the time to finish the experimental task compared to those using a static-type language.

In a different experiment, Hanenberg [10] showed a negative impact for a static-type system in one task and no significant difference in the other. The author measured two different points: the development time required to create a minimal scanner program and the quality of the resulting software measured by the number of successful test cases. Another experiment performed by Hanenberg, which focused on the relationship between type casting and development time [11], revealed a positive impact for a dynamic-type language. However, a positive impact could not be measured for non-trivial programming tasks.

A study on the Rosetta code, which is a code repository of solutions for common programming tasks in various languages, concluded that strongly typed languages are significantly less prone to runtime failures than interpreted or weakly typed language because more errors are caught at the compile-time. Nevertheless, these works referred to run-time failures or errors that make a program terminate (including inputs that cause a program to malfunction) or unable to run rather than using a set of test cases or testing based on specific requirements (black/white box testing) [9].

A qualitative study on the Ruby programming language carried out by Daly et al. [6] suggested that, at least in the specific setting of the experiment, the benefit of the type of system could not be shown.

In a paper entitled Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects[12], the authors investigated 100,000 open source projects available on GitHub and found no correlation between programming language employed and the number of issues listed on the bug tracker.

## 3. Experiment Description

### 3.1. Initial Considerations

Whether static-type systems reduce the development time and produce a better output remains controversial. For example, static-type systems may increase the development time due to type casting. The intent of the experiment is to check whether static-type systems help programmers code a solution from scratch and debug programs as well as identify under what conditions static-type systems are beneficial with regards to the number of fulfilled requirements and the number of fixed errors measured by manually prepared test cases.

We divided the experiment into two sessions. In the first session, subjects were asked to code a program from scratch, while in the second session they were asked to fix several errors in a given program code.

The day before the experiment, the subjects were given the program requirements, which included a demo video showing how the finished program should look. Therefore, during the experiment, the subject knew what to do and what functions or procedures were necessary to complete the tasks.

### 3.2. Environment and Measurements

The programming languages used in the experiment were C# for static-type systems and PHP for dynamic-type systems. Each language has its own built-in functions and procedures. Therefore, we informed the subjects about all equivalent functions in C# and PHP. We also imitated some functions by defining those available in one language but not the other.

The subjects were allowed to select their own code editor because we assumed that using a familiar development environment would produce a better code. Although the development environment and the employed code editor may affect the productivity, the experimental setup was designed to minimize the impact. First, the experiment requires a relatively small number of classes and procedures. Second, the auto-complete feature is not very useful in the experimental tasks.

We measured the number of achieved requirement points and fixed errors by running several test cases on the programs created by the subjects.

## 3.3. Programming Tasks

In the first session of this experiment, 14 subjects were asked to write 2 kinds of programs: a simple validation program and an encryption program. The main difference between these two is that one involves considerably more data types and requires more type casting. Each program had 7 requirement points (features) that must be implemented. Details of the programing tasks include:

### 3.3.1. Simple Validation Program

A Simple Validation Program requires the subject to create a form with several textboxes and apply a validation to each textbox. Example validations are username, password, phone number, and email address.

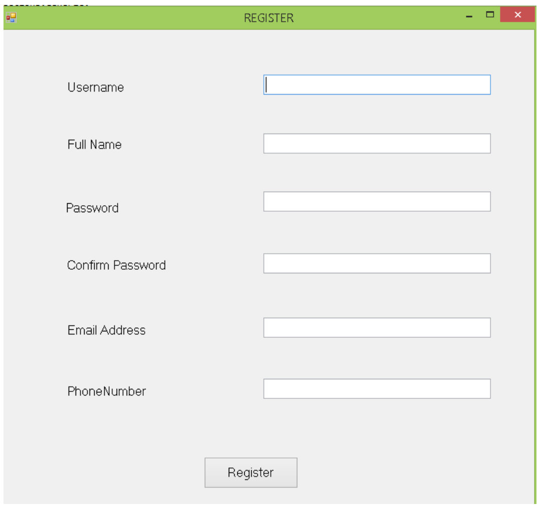
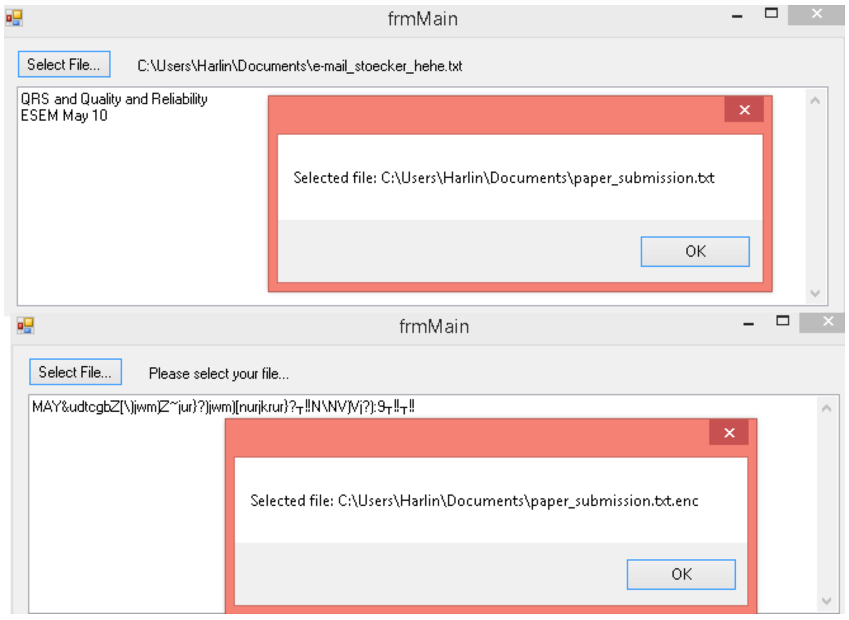


Fig. 2. Example Interface of a Simple Validation Program

### 3.3.2. Encryption Program

An Encryption Program requires the subject to create a simple algorithm to encrypt and decrypt a text file and validate whether the target file is created using the same program by placing a specific signature.


```
frmMain
Select File...   C:\Users\Harlin\Documents\e-mail_stoecker_hehe.txt
QRS and Quality and Reliability
ESEM May 10

Selected file: C:\Users\Harlin\Documents\paper_submission.txt
OK

frmMain
Select File...   Please select your file...
MAY&udtcgbZ[\)jwm)Z~jur}?)jwm)[nurjkrur}?┬‼N\NV)Vj?):9┬‼┬‼

Selected file: C:\Users\Harlin\Documents\paper_submission.txt.enc
OK
```

Fig. 3. Encryption Program

### 3.4. Experiment Execution

Fourteen subjects, including IT-professionals and graduate school students, participated in the experiment. They were divided equally into the static-type group and the dynamic-type group. All subjects were recent Computer Science graduates (within the last three years) and had one to two years of experience in the programming language they chose.

We divided the experiment into two sessions with a long break between the sessions. Each session was 90 minutes. The first session (coding) included two tasks: a simple validation program followed by an encryption program. Each subject had 45 minutes to complete a task and did not have a break between tasks. In the second session (debugging), the target program was similar to the program the subjects created in the first session. However, this time subjects were asked to fix errors in the given program code. Errors included a semantic error, a logical error, and a defect error related to the software requirements.

After the experiment, we asked several subjects to provide comments regarding the experiment and what was needed for a future experiment with an increased number of subjects and more diverse programming tasks.

### 3.5. Threats to Validity

As with any empirical study, this study has a number of potential threats to validity, including a small number of subjects, small programming tasks, and an artificial development environment. However, it should be emphasized that while a small programming task might not represent a real-world programming task, a large programming task has other factors that must be taken into account.

Another possible threat to validity is developer knowledge. Although we used only Computer Science graduates, we did not interview the subjects prior to experiment. Hence, there might be a gap in the subject's coding capabilities. Nonetheless, we also realize that there is not a well-accepted standard to classify whether someone is a good or bad software developer or to indicate if one subject is equal to another.

## 4. Results and Discussion

The experiment results and analysis are presented by giving descriptive statistics followed by significance tests to verify whether there is a significant difference between static- and dynamic-type solutions.

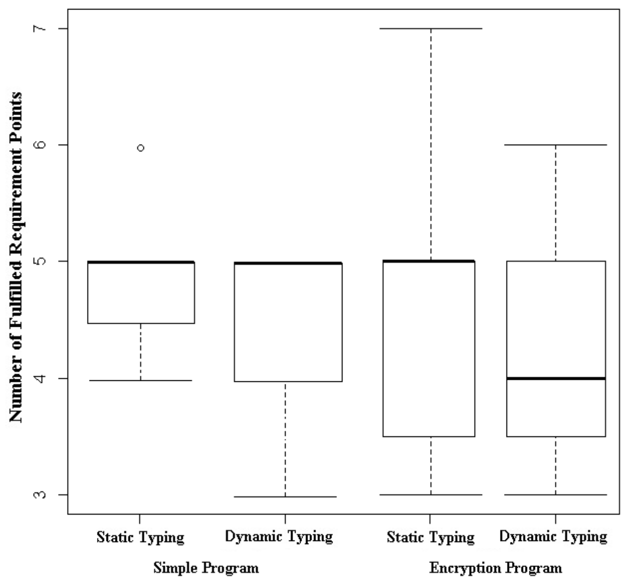


Fig. 4. Number of Fulfilled Requirement Points in the Coding Session.

## 4.1. Research Question 1 - *How do static-type systems affect the development of specific programming tasks when developers code a program from scratch?*

Figure 1 shows that a gap does not exist between solutions written in a static- and a dynamic-type system with regards to the number of achieved requirement points. This also applies to the result of the encryption program.

Table 2. Wilcoxon Rank Sum Value of Program Solutions

| | | | **Simple Program** | | **Encryption Program** | |
|---|---|---|---|---|---|---|
| | **N** | **Type Systems** | **Mean Rank** | **Sum Rank** | **Mean Rank** | **Sum Rank** |
| Session 1 Code from scratch | 7 | Static | 8.43 | 59.0 | 7.86 | 55.0 |
| | | Dynamic | 6.57 | 46.0 | 7.17 | 50.0 |
| Session 2 Debugging | 7 | Static | 9.93 | 69.5 | 10.36 | 72.5 |
| | | Dynamic | 5.07 | 35.5 | 4.64 | 32.5 |

Furthermore, we used Wilcoxon Rank Sum Test for independent samples to determine whether there is a significant difference between the number of fulfilled requirement points by type. Since the number of samples of static- and dynamic-type solutions is equal (n1=n2=7) for both the simple validation program and the encryption program, we chose 36 [2] as the critical value (Wcrit using α = 0.05 two tail). Because the Wilcoxon Rank Sum values (W) are 46 and 50 for the simple validation and the encryption program, respectively (Table II), we cannot reject the null hypothesis. Thus, for code written in the scratch tasks, both type systems produce similar results.

## 4.2. Research Question 2 - *How do static-type systems affect program debugging?*

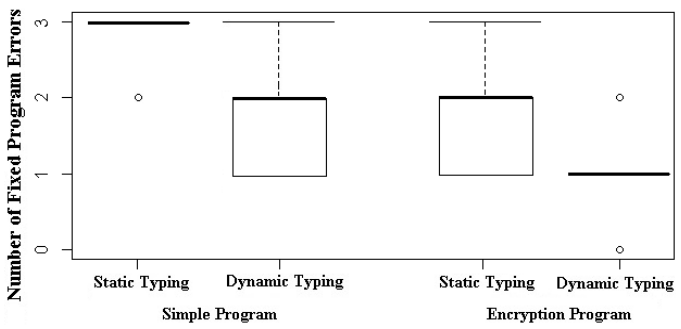


Fig. 5. Number of Fixed Program Errors in the Debugging Session

Figure 2 shows the number of successfully fixed errors in the simple validation program. It shows that there is a visible gap between the static- and dynamic-type solutions. This gap is larger for the encryption program solutions.

Again, we used the Wilcoxon Rank Sum Test for independent samples to test whether there is a significant difference between the number of successfully fixed errors in the debugging session. Since the static- and dynamic-type solutions have the same number of samples (n1=n2=7) for both the simple validation program and the encryption program, we chose 36 [2] as the critical value (Wcrit using α = 0.05 two tail). The Wilcoxon Rank Sum value for the simple validation program solutions (W) is 35.5, which is slightly lower than the specified critical value. For the encryption program solutions, the W value is 32.5 (see Table II). Therefore, there is a meaningful difference in the number of fixed errors between static- and dynamic-type solutions.

In conclusion, we reveal that a static-type system enhances the effectiveness of developers in program fixing or program debugging. Nonetheless, our current data still unable to confirm whether the type of program directly affects the debugging process.

### 4.3. Discussion

Although there is no difference when the subjects develop solutions of given tasks from scratch, the experiment reveals a benefit of using a static-type system when the subjects debug two given program codes. This might be because when debugging a program code, the developer must understand how the program actually works and what the initial programmer was thinking while writing the code. Dynamic-type systems provide little to none of this information, which may lead to confusion [18] [19]. From this perspective, the experiment provides evidence that static-type systems benefit developers in situations where program documentation is limited or unavailable (which are common scenarios). [5]

This benefit is more pronounced in debugging an encryption program containing more lines of code and data types. Although there is no direct proof that static-type systems are more suitable for a program with many lines of code and high data types [4], we conjecture that the encryption program has a more complex application flow. Hence, the time required to understand the program will be greater. Static-type systems make it easier for developers to understand the code, allowing the code that is “possibly” related to certain errors or defects to be easily identified.

## 5. Conclusion and Future Work

In this paper, we present an experiment that explores the impact of using a static-type system in the development of certain programming tasks with respect to the number of fulfilled requirements. We also investigate whether a static-type system helps developer fix program errors. Although the tasks are considered trivial, we hope our experiment contributes to the discussion of when to use a static-type system because empirical data that can be used to identify such situations is scarce.

One interesting point is that this study weakens the argument of authors who argue that dynamic-type systems reduce the development time due to the absence of type casting. In fact, in the debugging session of our experiment, the result shows a positive impact of static-type solutions for encryption program, which involves type casting.

The result of the experiment can be summarized as follows:

- When subjects coded from scratch, there is not a significant difference in terms of the number of successfully achieved requirement points between static- and dynamic-type solutions. This applies to both programs.
- In errors-fixing tasks, a static-type system may be beneficial. Subjects who used a static-type system tended to fix more errors. This benefit is more pronounced in encryption programs, which are relatively more difficult to complete and contain more data types.

Based on these conclusions, although direct proof is inconclusive, we suggest that the use of static-type systems is more beneficial and preferable in large-scale software developments due to the large number of software developers involved in a complex system with a large codebase. A large-scale software development creates a situation where one programmer should understand other programmers' code and be able to debug any kind of defects. In addition, the number of software bugs in such a system may be considerable.

As a future work, we will conduct additional experiments with more subjects and more diverse programming tasks to elucidate the characteristics of software programming activities where static- and dynamic-type systems are more beneficial. This will allow software developers to select the most suitable language. We are currently investigating the possible benefits of a static-type system in large-scale software projects. However, in large-scale software development there are too many factors to take into account because there are several software development phases. As the result, the use of a static- or a dynamic-type system in programming activities may become negligible.

## References

**(Book)**

[1] Benjamin C. P. (2002). *Types and Programming Languages*. Massachusetts: MIT Press.

[2] Kanji, Gopal K. (1993). *100 Statistical Tests*. London: SAGE Publication Ltd.

**(Report/working paper etc.)**

[3] Fareer, Evan R. (2011). *A Quantitative Analysis of Whether Unit Testing Obviates Static-type Checking for Error Detection*. California: California State University.

**(Conference paper in published proceedings)**

[4] Baishakhi R., Daryl P., Vladimir F., Premkumar D. (2014). A Large Scale Study of Programming Languages and Code Quality in Github. *Proceedings of the International Symposium on the Foundations of Software Engineering*. Hong Kong.

[5] Clemens M., Stefan H., Romain R., Eric T., Andreas S. (2012). An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software. *Proceedings of the Object-oriented Programming, Systems, Languages, and Applications* (pp. 683 – 702). Tucson, AZ.

[6] Mark T. D., Vibha S., Jeffey S. F. (2009). Work in progress: An Empirical Study of Static Typing in Ruby. *Workshop on Evaluation and Usability of Programming Languages and Tools*. Orlando, FL.

[7] Harlin I. R., Hironori W., Yoshiaki F. (2017). Impact of Using a Static-type System in Computer Programming. *Proceedings of the 18th International Symposium on High Assurance Systems Engineering* (unpublished). Singapore, Singapore.

[8] Humaira R., Sakamoto K., Ohashi A., Hironori W., Yoshizaki F. (2012). Towards a Unified Source Code Measurement Framework Supporting Multiple Programming Languages. *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering* (pp. 480 – 485). Redwood City, CA.

[9] Sebastian N., Carlo A. F. (2015). A Comparative Study of Programming Languages in Rosetta Code. *Proceedings of the 37th International Conference on Software Engineering* (pp. 778 – 788). Firenze, Italy.

[10] Stefan H. (2010). An Experiment about Static and Dynamic Type Systems: Doubts about the Positive Impact of Static Type Systems on Development Time. *Proceedings of the Object-oriented Programming, Systems, Languages, and Applications* (pp. 22 – 35). New York, NY.

[11] Stefan H., Andreas S., (2011). Static vs. Dynamic Type Systems: An Empirical Study about the Relationship between Type Casts and Development Time. *Proceedings of the Symposium on Dynamic Languages* (pp. 97 – 106). Portland, OR.

[12] Tegawende F. B., Ferdian T., David L., Lingxiao J., Lauret R. (2013). Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects. *Proceedings of the 37th Annual International Computer Software and Application Conference* (pp. 1 – 10). Kyoto, Japan.

**(Journal article)**

[13] Lamport L., Paulson L. C. (1999). Should Your Specification Language Be Typed? *ACM Trans. Programming Languages and Systems 16(3)*, 872-923.

[14] Laurence T. (2009). Dynamically Typed Languages. *Journal of Advances in Computers, vol. 77,* 149 – 184.

[15] Luca C. (1997) Type Systems. *In Allen B. Tucker, The Computer Science and Engineering Handbook*, chapter 103, 2208-2236.

[16] Lutz P. (2000). An Empirical Comparison of Seven Programming Languages. *IEEE Computer, vol. 33(10)*, 23-29.

[17] Lutz P., Walter F. T. (1998). A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking. *IEEE Trans. Software Engineering, vol. 24(4)*, 302-312.

**(Online Article)**

[18] University of Washington. (2004). Dynamic Typing vs Static Typing. *CSE 341: Lecturer Notes*. Retrieved

January 26, 2017, from: http://courses.cs.washington.edu/courses/cse341/04wi/lectures/index.html

[19] Zack G. (2013, April). An Introduction to Programming Type Systems. *Smashing Magazine*. Retrieved January 26, 2017, from https://www.smashingmagazine.com/2013/04/introduction-to-programming-type-systems/

(**All authors should include biographies with photo at the end of regular papers.**)

**Ismail Rizky Harlin** received a Bachelor of Computer Science from Binus University, Jakarta, Indonesia in 2010. He is now a Masters course student of Department of Information and Computer Science, Waseda University, Tokyo, Japan. His research interests include software engineering, especially programming languages.

**Hironori Washizaki** is a professor at Waseda University, Tokyo, Japan. He is also a visiting professor at the National Institute of Informatics, Tokyo, Japan. He obtained his Doctorate in Information and Computer Science from Waseda University in 2003. His research interests include software reuse, patterns and quality assurance. He has served as a member of program committees for many international conferences, including ASE, SEKE, PROFES, APSEC and PLoP. Additionally, he has served as a member of the editorial board for several journals, including the Journal of Information Processing.

**Yoshiaki Fukazawa** received the B.E., M.E. and D.E. degrees in Electrical Engineering from Waseda University, Tokyo, Japan in 1976, 1978, and 1986, respectively. He is now a professor of Department of Information and Computer Science, Waseda University as well as the Director of Institute of Open Source Software, Waseda University. His research interests include software engineering, especially reuse of object oriented software and agent-based software.