Static and Dynamic Types in Software Development

Emma G. Callery
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
calle052@morris.umn.edu

ABSTRACT

Recent years have seen an increase in the popularity of dynamically type languages, such as Groovy, bringing greater attention to the discussion on the benefits of the different type systems. Static type systems arguable have the benefit that because types are checked by the compiler before run time users may find type errors earlier. This could potentially improve the readability and maintainability of code as well as improve the software's documentation. In contrast, dynamic type systems argue of the benefit the provided flexibility allows to users. Unfortunately most arguments made for and against both type systems tend to come from personal opinions or personal experience. Several studies were conducted recently in an attempt to empirically ascertain possible benefits of type systems on software development. One such study tried to determine the programmers' point of view about the trade-offs by analyzing 6,638 open source projects written in Groovy. Two more studies performed experiments to find a relationship between type systems and software development time; to support arguments that because static type systems improve readability and documentation programmers can complete programming tasks faster. All three of these studies, which are discussed in this paper, provide evidence for the type system discussion and suggested reasons for their results.

Keywords

Static Types, Dynamic Types, Programming Languages, Java, Groovy.

1. INTRODUCTION

Type systems are a major part of programming languages and play important roles in software education, research, and industry. While a great number of programming languages used in industry are statically typed (e.g.,Java), a fair number of programs prefer to use dynamically typed systems (e.g.,Groovy); especially for areas such as web development. This leads to the proposed question of "whether one system or another, either static or dynamic, has a larger benefit for the humans that use them"[3] than the drawbacks that are argued by the other side.

This question has lead to a large discussion of the benefits and drawbacks of each of the different type systems, with people arguing strongly for and against both. For example, there are some programmers that believe strongly that the structural benefits provided by static types outweighs the 'tediousness' or 'rigidity' that programmers that prefer dynamic types claim static types systems inherently have. Several recent studies have been conducted to try to determine if any of the arguments given by both sides have any real empirical support.

This paper starts by defining static and dynamic typed languages along with commonly argued benefits for both in section 2. Section 3 looks at a study that analyzed past projects to find patterns in how programmers use types. Two more studies have tried to look for empirical evidence to support the claim the static type systems have better trade-offs, their studies and results are in sections 4 and 5. We conclude by attempting to answer the proposed questions in section 6.

2. BACKGROUND INFORMATION

This section defines static and dynamic types, as used in this paper, along with the common arguments for both type systems. Within this paper the only languages used are Java and Groovy for static and dynamic languages, respectively.

2.1 Static Types

A language like Java is said to be statically typed if the type of every variable is known by the compiler at compile-time. For the type to be known at compile-time the programmer has to have included the information required for the compiler to determine the type of the variable, which is done through several options, usually via a declaration. Static language compilers enforce these types by only allowing values of the variable's type to be assigned to the variable (see Figure 1). This means that when the compiler goes through the code it not only checks for syntactic errors, but also checks for type errors.

2.1.1 Arguments For Static Types

Type checking is one benefit of static types. Since this checking is done at compile-time, the errors are more likely to be caught quickly and without having to run the whole program as frequently[5]. A second common argument is that types improve the structure of a program [3]. Better

UMM CSci Senior Seminar Conference, December 2014 Morris, MN.

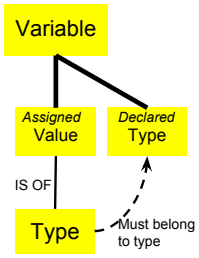

Figure 1: Diagram of Static Types and Type Checking [2]

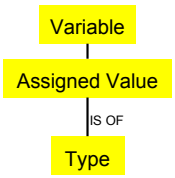

Figure 2: Diagram of Dynamic Types [2]

structure is believed to help the programmer reason about the program. The last significant argument for static types is that the types provide a form of documentation that can communicate to programmer [3]. When the documentation from the types is added to the documentation written by the programmer it will, in theory, result in a documentation of greater quality.

2.2 Dynamic Types

A language, like Groovy, is dynamically typed if the type of a variable is interpreted at run-time. This is because types in dynamic languages are only associated with values [2], as shown in Figure 2. This does not mean that a type can not be associated with a variable, only that the language would not require it [4]. This *option* leads to what is called *optional typing* where the programmer is primarily responsible for the types incorporated in the program, explained further in section 3.

2.2.1 Arguments for Dynamic Types

Not being required to declare a type for every variable is argued to make dynamically typed programs shorter and therefore faster to write and read. Dynamic types are also argued to allow greater flexibility to the programmer [3]. There are several reason for this including easier passing of variables between different parts of a program. No specific type requirements allow for easier reuseability of part of programs[5]. Declaring variables is considered 'administrative' and would allow for the programmer to focus on the more conceptual aspects of the program[3].

2.3 Groovy

Groovy[1] is a dynamic language designed for the Java Virtual Machine, or JVM. Groovy builds on the static strengths of Java while having additional features of dynamically typed languages; features such as closure and support for scripts. Groovy claims to "seamlessly integrates with all existing Java classes and libraries" [1].

This statement means that programmers using Groovy can also access the static types within Java while also having the additional features found in other dynamic programming languages. Programmers using Groovy have the option of adding type declarations without being required to do so. This keeps Groovy from being a static language because this type information is not used by the compiler to check for errors. The result is Groovy being called an optional-type. This is important because it means that by removing the extra dynamic features Groovy becomes a basic dynamic replacement for Java, with type declarations being replaced with a default. This restricted Groovy is the form of Groovy used in the latter two studies discuses in this paper

3. PROGRAMMER PREFERENCES

This section will look at a study which attempted to determine where programmers use types and what types they used in the optional-type language *Groovy*.

3.1 Study on Optional Typing

Carlos Souza and Eduardo Figueiredo studied programmer preferences in their paper *How do Programmers Use Optional Typing? An Empirical Study*. In this study Souza and Figueiredo looked at 6,638 open source projects written in Groovy, all gathered from Github. For each project Souza and Figueiredo gathered the source code, the commit history for the project, and the metadata for both the source code and all programmers involved in the project. The source code metadata included the kinds of type declaration, such as whether the declaration was public, private or protected. The programmer metadata primarily included their past language experience, as indicated by their other Github projects.

Looking over all of the projects Souza and Figueiredo tried to answer several questions about the use of types in the different projects. The questions were:

1. "Do programmers use types more often in the interface of their modules?"
2. "Do programmers use types less often in test classes and scripts?"
3. "Does the experience of programmers with other languages influence their choice of types in their code?"
4. "Does the size, age or level of activity of a project have any influence on the usage of types?"
5. "In frequently changed code, do developers prefer typed or untyped declaration?"

After collecting data from all the projects the researchers sorted the data by different measures. The measures were the kind of type declaration, the visibility of the declaration, the usage of the different types and where. Also used to sort the collected data was the size of the project in lines of code, the age of the program along with the number and frequency of commits. It was from this data and different comparisons that Souza and Figueiredo attempted to answer their research questions.

3.2 Results for Optional Typing Study

Question 1 was found to be true; variables and fields that are public, such as constructor and method parameters and returns, are frequently typed, see Figure 4. In addition the visibility of the declaration is also important. Souza and

Declaration Visibility	number	mean	median	standard deviation
Public	5852	0.69	0.75	0.29
Protected	2387	0.93	1.00	0.19
Private	6023	0.43	0.40	0.32

Figure 3: Visibility of type declarations

Declaration Type	number	mean	median	standard deviation
Field	6000	0.43	0.39	0.33
Constructor Parameter	1670	0.80	1.00	0.35
Method Parameter	4867	0.67	0.86	0.36
Method Return	5881	0.68	0.75	0.31
Local Variable	5845	0.29	0.18	0.32

Figure 4: Type declaration usage

Fiqueiredo found that protected declarations are almost always typed and while public declarations are not as frequently typed as protected, they are typed more often than private declarations, see Figure 3. Though the reason for why module interface definitions are most often typed is still unresolved; Souza and Fiqueiredo suggest that the implicit documentation the types provide are the main incentive as programmers may consider documentation in these area important.

On the other hand, question 2 was found to be false. Documentation is rarely needed for test classes and scripts as most are small and easy to understand already. Test classes tend to have one sole purpose, and are very rarely reused. Scripts can not be accessed by other modules so the type being passed is most likely already known.

Unsurprisingly, question 3 was true. How a programmer uses types in an optional language greatly reflects how that programmer used types in other languages. Programmers coming from a statically typed language are more likely to add types than programmers coming from a dynamically typed language, see Figure 5. It has been found that programmer become comfortable in whatever language they use most frequently.

Declaration Type	Background	Number	Mean	Median	Standard Deviation
Field	Static Both Dynamic	782 3183 2035	0.56 0.43 0.38	0.52 0.39 0.36	0.35 0.34 0.29
Constructor Parameters	Static Both Dynamic	224 991 455	0.83 0.80 0.80	1.00 1.00 1.00	0.33 0.35 0.34
Method Parameters	Static Both Dynamic	662 2694 1511	0.73 0.67 0.65	0.91 0.84 0.83	0.34 0.36 0.37
Method Returns	Static Both Dynamic	764 3205 1912	0.73 0.66 0.68	0.85 0.75 0.74	0.30 0.32 0.29
Local Variables	Static Both Dynamic	798 3230 1817	0.39 0.28 0.25	0.31 0.17 0.14	0.36 0.32 0.30

Figure 5: Type declaration usage by programmer background

Declaration Type	Project Type	number	mean	median	standard deviation
Field	Mature Other	221 5779	0.53 0.43	0.48 0.39	0.27 0.33
Constructor Parameter	Mature Other	172 1498	0.83 0.80	1.00 1.00	0.30 0.35
Method Parameter	Mature Other	222 4645	0.69 0.67	0.78 0.86	0.29 0.37
Method Return	Mature Other	222 5659	0.72 0.68	0.79 0.75	0.24 0.32
Local Variable	Mature Other	223 5622	0.32 0.29	0.22 0.17	0.28 0.32

Figure 6: Type declaration usage by project maturity

Declaration Type	Project Type	number	mean	median	standard deviation
Public	Mature Other	223 5629	0.72 0.69	0.76 0.75	0.24 0.29
Protected	Mature Other	183 2204	0.88 0.94	1.00 1.00	0.21 0.19
Private	Mature Other	221 5802	0.53 0.43	0.48 0.40	0.26 0.32

Figure 7: Type visibility usage by project maturity

Souza and Figueiredo initially believed that as time goes on and the projects grow and *mature*, the maintenance of projects becomes more difficult. This leads programmers to use more types as a means to make code more readable. A mature project was defined as "a project that is 100 days old or more and has, at least, 2K LoC [Lines of Code] and 100 commits."[5] The data gathered to try to answer question, shown in Figures 6 and 7, however, showed that this was not the case. Souza and Figueiredo thought that maybe the data they gathered could not "actually correlate to the need for maintenance" or if the data could work "programmers might not have the opportunity or desire to make their code more maintainable" [5].

Question 5 probably has the biggest debate attached to it. That being the argument that types, acting as a form of documentation, could make frequently changed code easier to change. Others argue that untyped code, being simpler, can be changed faster. It is this latter argument that Souza and Figueiredo found evidence for. They found that as changes in a file increase in frequency, 65% of the mature projects showed a preference for the use of untyped declarations.

Overall this means that programmers are influenced by the type system that they are use to. Generally, programmers mostly type the definitions of a modules interface more frequently than anything else. Types are more likely omitted when programmers have to make changes to pieces of code. This means that the debate of static vs dynamic types needs to be looked at more closely to determine an actual benefit to one type system over the other.

4. STATIC VS DYNAMIC TYPE SYSTEMS

This section discusses the 2011 study and results of Andreas Stuchlik and Stefan Hanenberg's study *Static vs. Dy-*

namic Type Systems: an Empirical Study About the Relationship between Type Casts and Development Time [6]. The study was designed for the argument that, in simple programming tasks, the existence of type declarations leads to a reduction of development time. The reasoning behind this hypothesis is that including type declarations should lead to better development times, through types improving the program structure and the programmers' understanding.

4.1 Experiment

The study had 21 subjects who were asked to complete two equivalent sets of programming tasks, with five programming tasks in each set. One set of tasks would be completed using Java and the other set using the restricted form of Groovy from section 2.3. The subjects were split into two groups, as evenly as possible; one group started by completing one set of tasks in Groovy then completed the other tasks in Java, the other group did the reverse. It should be noted that, all subjects were familiar with Java, but none had used Groovy. Each of the five tasks within a set had varying numbers of type declarations required to successfully pass the associated tests. Also associated with each task was the expected lines of code(LoC). However the number of declarations match between sets.

All subjects were given both sets of tasks, though only one set of tasks were explained in the study's paper, and here, for simplicity.

Task 1(declarations:1,LoC:5)
This task requires the subject to write a method that receives a 'player' object and 'goal' object as its parameters. The method should cause values within each object to increase if an aspect of the 'player' object matches a condition.

Task 2(declarations:2,LoC:9)
Similar to task 1, the subject is required to write a method that takes a 'player' object and a 'goal' object as parameters, along with an additional 'kick' object. If the 'player' object matches a specific condition then a value in both the 'player' and 'goal' objects is increased.

Task 3(declarations:2,LoC:5)
For task 3 subjects needed to write a method that takes two separate 'player' objects and increases the corresponding instance variable for both objects.

Task 4(declarations:4,LoC:13)
Task 4 requires subjects to write a method for storing a value between two 'player' objects that match on a specific condition.

Task 5(declarations:8,LoC:25)
Task 5 requires subjects to write a method to replace one 'player' object with another 'player' object, such that the second 'player' object now has the values of the first 'player' object and the first has the values of the second.

The assumption in the design of this study was that "the higher the number of type [declarations] is, the larger is the difference in development times for the statically of dynamically typed code." [6]. It was important to note that all the tasks are trivial, and did not require any other software, such as an API.

Lowest Time Results	Sum	Task 1	Task 2	Task 3	Task 4	Task 5
Group A	—	Groovy	—	—	—	—
Group B	Groovy	Groovy	—	Groovy	—	—

Figure 8: Statistically significant positive impact results for shortest time between Group A (Groovy first) and Group B (Java first)

4.2 Results

For the subjects that started by using Groovy first, only task 1 showed a statistically significant positive impact for the dynamically type language, Groovy. The subjects that started using Java first showed significant positive impact of Groovy, for both tasks 1 and 3 as well as the sum of all tasks. None of the other tasks showed any significant positive impact, for both groups. See Figure 8 for results.

For both groups, no significant positive impact was found for the static type system for any task. Additionally, the amount of time saved ranged from 8 minutes to 37 minutes, for all tasks, though the median of the summed development times for the statically typed task was less than 100 minutes. This can not be explained just by the time taken to write the type casts, and suggests that type systems have some kind of complexity beyond this.

It should be noted that these results only concern trivial programming tasks, and would not apply to non-trivial task. The authors argue that "as soon as we speak about non-trivial programming tasks, no positive impact [of dynamic types] can be measured." Though they do not point to a study that could actually answer this statement, the authors do suggest that further studies need to be conducted in order to verify their hypothesis.

5. INFLUENCE OF STATIC TYPES

This section looks at the 2012 study conducted by Stefan Hanenberg, Clemens Mayer, Romain Robbes, Andreas Stefik, and Eric Taner in their paper *An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software* [3]. This study looked into the argument that types act as a form of documentation, as a benefit of static type systems. The study argues that if there is no other form of documentation present and variables are not typed, it should take a programmer longer to complete a task than if the variables are typed.

The researcher came up with two null hypotheses, the first stating that,

> The development time for completing a programming task in an undocumented API is equivalent when using either a static type system of a dynamic type system.[3]

Because this hypothesis only accounts for the design of the API as a whole, a second null hypothesis was included:

> There is no difference in respect to development time between static and dynamic type systems, despite the number and complexity of type declarations in an undocumented API. [3]

Both hypotheses were used so that if either hypothesis is rejected by the data, insight can be gained on the relative benefits and consequences of both type systems.

In the case of this study an undocumented API means that the interfaces used by the subjects had no documentation available to tell what exactly the interfaces did and which parameters needed to be passed to the methods within the API. It should be noted that the dynamic API used in this study was made by taking the static API and removing the type annotation; replacing them with the Groovy default, *def*. This posses a potential problem with the study methodology in that this does not use any of the possible features of dynamically typed languages.

5.1 Experiment

The experiment designed for this study had 27 subjects that had experience with the statically typed language, Java, but had no experience with the dynamic language Groovy (restricted as described in section 2.3). For this the authors simply introduced the subjects to Groovy as "a Java version where all declarations of variables, parameters, etc. only required the keyword *def*." [3]

While a learning effect was a potential issue to the validity of the experiment; this experiment design has already been applied to other studies successfully[3]. In addition there are valid approaches for statistically analyzing the data fairly. So long as the learning effect is less than the effect caused by the language then the measures can still be used to determine the effect of the language[3].

As with the study in the sections above, subjects were separated into two groups, as evenly as possible, and each group was asked to complete the same five programming tasks. The five tasks were completed in one language using one undocumented API before completing the five tasks in the second language with another API. Group one (called GroovyFirst) would complete the first set of assigned tasks using Groovy and then complete the tasks using Java. The other group (called JavaFirst) did the same tasks but did the tasks in Java first.

The tasks were designed to vary in difficulty and the number of type declarations required for classes within each language.

Task 1
This task was classified as being of easy difficulty and only asked to return an instance of a specific class. This required only one type to be identified and taking only one line of code to write.

Task 2
This task was also considered easy. It required the initialization of an object found in the class discovered in task 1. The initialization required an 'initializer', which required a secondary object. This means that the subjects had to identify three classes.

Task 3
This task was considered medium difficulty and required subjects to create a transformation of an object, from task 2, with a corresponding class. To do this a graph first had to be created from the task 2 object. An initialization of a different object had to be created, along with a node. The node and both objects where then passed to the transformation class. This means that the subject had to identify three classes.

Aspect	Task 1	Task 2	Task 3	Task 4	Task 5
Less Development Time	Java	Groovy	Groovy	Java	Java
Fewer Builds/Runs	———	Groovy	Java	Java	———
Fewer Files Looked At	Java	Groovy	Java	Java	Java
Fewer File Switches	Java	Groovy	Groovy	Java	Java

Figure 9: Collective results of all tasks showing leading language(either statically typed Java or dynamically typed Groovy)[3]

Task 4
This task was the only task to be considered difficult. The subjects were required to add a node to a graph through parameterizing a non-trivial initializer correctly. The parameters required were an instance of a sequence object, each object contained an identifier and a sequence of pairs. These pairs contain an identifier and another sequence object. Despite the subject only needing to identify three classes, the recursive definition of the objects lead to a suspicion that the code may be hard to understand, especially in the dynamic type system, with no defined types.

Task 5
This task was considered easy, but required the highest number of classes that needed to be identified. The subjects were required to take the object from task 2 and create method that takes a command object(also created) and make a 'menu' from the task 2 object. This 'menu' method also takes three additional classes to fully complete the command. In total six classes needed to be identified.

It is interesting to note is that one subject was removed from the experiment because it was found that the subject had "spent a very large amount of time in reading the complete source code while working on task 2 and then solved tasks 3 and 4 quickly" [3]. After this had been confirmed to be the case the subjects data was removed. This indicates that different programmers have different ways of processing and writing code, suggesting that no one way of programming will work equally well for everyone.

5.2 Results

It should be noted that, for several reasons, the programming tasks were designed with the assumption that "for all tasks, except task 1, the static type system would show a measurable positive impact"[3].

5.2.1 Null Hypotheses results

After analysis of the collected time data it was found that, concerning the first hypothesis, tasks 1, 4, and 5 showed a positive impact for the static type system, Java. Tasks 2 and 3, however, showed a positive impact for the dynamic type system, Groovy, as seen in the first line of Figure 9. This caused hypothesis one to be rejected.

Hypothesis 2, however, cannot be rejected because just the number of types and difficulty of the tasks "cannot be a main effect of the difference between static and dynamic" [3]. This is because for the 'easy' tasks (1, 2, and 5) the dynamic type system was found to have a positive impact on

only task 2, while tasks 1 and 5 showed a negative impact. This contradicted the assumption that the number of types needing to be identified is a main factor. Task 2 required more types to be identified than task 1 but fewer types than task 5. For tasks 3 and 4, which had the same number of type identifications required but different difficulties, contradictory results were also found.

5.2.2 Exploratory Study and Results

The contrary nature of the hypothesis 2 results lead to an investigation to find if other influences could have contributed to the results. One suggested influence was the measured number of builds and times tests were run during each task. These numbers were taken from the times the compiler ran and the times the 'start-button' was clicked. Tasks 1 and 5 registered no significant difference between Java and Groovy in number of builds and tests run. However tasks 3 and 4 showed the static type system, Java, had the fewer test runs. Task 2 was the only task where the dynamic type system, Groovy, had fewer test runs. The results are seen in second line of Figure 9.

A second suggested factor was the number of files the subject was looking at, which may indicate the amount of the source code that a subject needed to read in order to complete the task. The authors assumed that this could have an effect because subjects using the dynamic system 'should' be more likely to look at unrelated source code. Evidence gathered seems to support this theory for all but one task, task 2. For tasks 1, 3, 4, and 5 fewer files were viewed when the subject was using Java. Task 2 showed that fewer files were viewed when the subjects used Groovy. See line three of Figure 9.

The final suggested factor was that the number of times the subject switched could indicate the amount of exploration the subject did while solving the task. Similarly to the previous suggestion, the effect was assumed to come from the 'required' need for user of a dynamically typed language to frequently change files to find or formulate answers. This was measured separately from the previous analysis. Oddly enough the results from this analysis directly correspond to the development time results, for each task. See line four of Figure 9. Tasks 1, 4, and 5 showed fewer file switches when the subjects used the static type system Java, while tasks 2 and 3 showed fewer files switched when subjects used the dynamic type system, Groovy. This result suggests that the number of switched files could be an indicator for the resulting development time.

6. CONCLUSIONS

This paper looked at three studies that attempted to help address the discussion on the benefits of the different type systems. The first was a case study that looked at how programmers use types in Groovy. The second study was an experiment to compare static and dynamic types when programmers where completing tasks with an undocumented piece of software. The third study experimented to determine the influence of types on the development time for trivial programming tasks.

The goal of these studies were to answer some questions about benefits type systems have for programmers. The first study looked at how do programmers use types? for this the first study analyzed data gathered from 6,638 projects written in Groovy. The researchers found that past experience plays a part in how programmers use types; if the programmer has used at least one dynamically typed language in the past, they are likely to use types less frequently. While programmers tend to use types in their interfaces more then any other form of declaration very few include types in test classes and script files. Also the size, age, or level of activity in the overall project has no influence on how programmers include types.

The other two studies that we considered gave 20-30 subjects a set of programming tasks in both Java and Groovy to determine the effect type systems have on development time. Both studies determined that while there is no simple answer for the question of static versus dynamic, the type system does have an effect. Study two found that static type systems improve development time if the types help explain document design decisions, or the number of classes the programmer needs to identify is high. For simpler task dynamic type systems can potentially reduce development times. Study three found that the dynamic type system had a positive effect in three of their five tasks as well as over all, the remaining tasks showed no significant difference between the two type systems.

7. ACKNOWLEDGMENTS

Thank you to Elena Machkasova, Stephen Adams and Peter Dolan

References

[1] Groovy programming language. http://groovy.codehaus.org/.

[2] S. Ferg. Static vs. dynamic typing of programming languages, 2012. From Python Conquers the Universe on wordpress.com.

[3] C. Mayer, S. Hanenberg, R. Robbes, E. Tanter, and A. Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 683–702, New York, NY, USA, 2012. ACM.

[4] N. Nome. What is the difference between statically typed and dynamically typed languages?, October 2009. From stackoverflow.

[5] C. Souza and E. Figueiredo. How do programmers use optional typing?: An empirical study. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 109–120, New York, NY, USA, 2014. ACM.

[6] A. Stuchlik and S. Hanenberg. Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 97–106, New York, NY, USA, 2011. ACM.