

# How Do Programmers Use Optional Typing? An Empirical Study

Carlos Souza

Software Engineering Lab  
Federal University of Minas Gerais (UFMG)  
carlososouza@gmail.com

Eduardo Figueiredo

Software Engineering Lab  
Federal University of Minas Gerais (UFMG)  
figureido@dcc.ufmg.br

## Abstract

The recent popularization of dynamically typed languages, such as Ruby and JavaScript, has brought more attention to the discussion about the impacts of typing strategies over development. Types allow the compiler to find type errors earlier and potentially improve the readability and maintainability of code. On the other hand, "untyped" code may be easier to change and require less work from programmers. This paper tries to identify the programmers' points of views about these tradeoffs. An analysis of the source code of 6638 projects written in Groovy, a programming language which features optional typing, shows in which scenarios programmers prefer typing or not their declarations. Results show that types are popular in the definition of module interfaces, but are less used in scripts, test classes and frequently changed code. There is no correlation between the size and age of projects and how their constructs are typed. Finally, we also found evidence that the background of programmers has influence on how they use types.

**Categories and Subject Descriptors** D.2.3 [Software Engineering]: Coding Tools and Techniques

**General Terms** Experimentation, Language

**Keywords** Type systems, static analysis, Groovy

## 1. Introduction

Type systems are one of the most important characteristics of a programming language and also a major topic of research in software engineering [12, 14, 19, 27]. A programming language's type system determines when the type of a statement is defined [12]. Statically typed languages, such as Java and C#, require programmers to explicitly define the type of a declaration, which can then be used by the compiler to check for type errors. On the other hand, in dynamically typed languages, such as Ruby and JavaScript, the definition of the type of a statement only happens at run time.

Discussions about what is the best type system for a particular situation have become increasingly important in recent years due to the rapid popularization of dynamically typed languages. According to the TIOBE Programming Community Index [1], a

well-known ranking that measures the popularity of programming languages, 27% of the programming languages used in industry are dynamically typed. A decade ago, this number was only 17%. Among the 10 languages on top of the ranking, four are dynamically typed: JavaScript, Perl, Python and PHP. None of these languages were among the top 10 rank until 1998.

Several factors may be considered when choosing between a dynamically or statically typed language. Dynamically typed languages tend to allow programmers to code faster and to adapt their programs to frequently changing requirements more easily. Also, by removing the repetitive work of defining types, these languages allow programmers to focus on the problem to be solved rather than on the rules of the language [13].

Statically typed languages also have their advantages. They allow compilers to find type errors statically [10]. Typed declarations increase the maintainability of systems because they implicitly document the code, telling programmers about the nature of statements [5, 11]. Systems built with these languages tend to be more efficient since they do not need to perform type checking during execution [3, 6]. Finally, modern development environments, such as Eclipse and IDEA, are able to assist programmers with functionalities such as code completion based on the information provided by statically typed declarations [4].

Some languages try to combine characteristics from both static and dynamic type systems. Groovy [2] is one of these languages. Although Groovy is mostly a dynamically typed language, it gives programmers the option to use type annotations as a means to document their code. It is also possible to turn static type checking on so the compiler can find type errors before execution. This allows developers to choose the most appropriate paradigm for each situation.

Understanding the point of view of programmers about the use of types is an important matter. Programming language developers can consider this information in their design so they can develop the most appropriate features for their target audience. Tools can be created or improved to overcome weaknesses of languages. Finally, programmers can benefit from this knowledge when choosing programming languages or typing paradigms for a given context.

This paper presents a large scale empirical study about how programmers use optional typing in Groovy. Through the analysis of a massive dataset with almost seven thousand Groovy projects, we were able to identify when programmers prefer to type or not their declarations. Our results give more visibility about what is the point of view of programmers about the use of types. This analysis complements, with a different point of view, existing studies based on controlled experiments [7–9, 11, 32–34].

The remainder of this paper is organized as follows. Section 2 introduces the main concepts of the Groovy programming language and Section 3 presents the study settings. Section 4 describes the

results of the study, which are then discussed in Section 5. Threats to the validity and related work are presented in Sections 6 and 7. Finally, Section 8 concludes this study and suggests future work.

## 2. The Groovy Language

Groovy is a dynamically typed programming language designed to run on the Java Virtual Machine. Its adoption has grown remarkably over the last years. According to the TIOBE Programming Index, Groovy is the 22<sup>nd</sup> most popular language in the software industry [1], ahead of languages like Prolog, Haskell and Scala. It builds upon the strengths of Java, but has additional features inspired by dynamic languages such as Python and Ruby. Like Java, Groovy code is compiled to bytecode, allowing it to seamlessly integrate with existing Java classes and libraries. These factors have attracted a large number of Java programmers who want to use Groovy’s dynamic functionality without having to learn a completely different language or change the execution platform of their systems.

When Groovy was first launched, in 2007, it was a purely dynamically typed language. However, it allowed programmers to optionally type their declarations. Examples of typed and untyped declarations combined in the same file are shown in Listing 1. This kind of typing should not be confused with static typing since the Groovy compiler does not use these type annotations to look for errors. For example, the snippet of code shown in Listing 2 compiles without any errors. During runtime, the *string* variable references an instance of the *Integer* class. However, an exception is thrown when the method *toUpperCase* is invoked since the *Integer* class does not have such method.

**Listing 1** Groovy is a dynamic language

```

1 class DynamicTyping {
2     private String typedField
3     private untypedField
4
5     DynamicTyping(typedParam) {}
6
7     def untypedMethod(untypedParam, int
8         typedParam) {
9         def untypedVariable = 1.0
10        return untypedVariable
11    }
12    int typedMethod() {
13        String typedVariable = ""
14        return typedVariable
15    }
16 }
```

**Listing 2** A class written in Groovy

```

1 String string = new Integer(1)
2 string.toUpperCase()
```

Since version 2.0, Groovy allows programmers to explicitly activate static typing with the usage of the *@TypeChecked* annotation. This makes Groovy a gradually typed language [15–19]. In this mode, the Groovy compiler looks for type errors and fails if it finds any. Listing 3 shows an example of static typing in Groovy. Trying to compile the class *TypeCheckedGroovyClass* produces an error since the method *sum* is supposed to receive two parameters

**Listing 3** A class written in Groovy

```

1 @TypeChecked
2 class TypeCheckedGroovyClass {
3
4     static int sum(int a, int b) {
5         a + b
6     }
7
8     public static void main(String[] args) {
9         println sum("1", "2")
10    }
11 }
```

of the type *int*, but it is actually called with two parameters of the type *String*.

The *@TypeChecked* annotation is reasonably recent and most Groovy programmers still do not use it. Typing annotations on the other hand are very popular. Although they do not provide static type checking, they are capable of documenting the code and aiding in the integration with development tools. In the remainder of this text, we refer to declarations with type annotations as “typed”, while the word “untyped” is used for declarations with no type annotations.

## 3. Study Settings

The study presented in this paper consists in the static analysis of the source code of a corpus of 6638 Groovy projects. Its goal is to find in which contexts, Groovy programmers type or not their declarations. In this section, we present five research questions that guide our analysis, the data collection and analysis procedures as well as the characterization of the studied dataset.

We have shared all the artifacts of this study in our project website<sup>1</sup>. This includes the source code of the programs used in the data collection and analysis procedures, the analyzed data (except the source code of the analyzed projects) and detailed results. Our goal is to allow this study to be easily replicated or extended.

### 3.1 Research Questions

We aim to answer the following research questions about the usage of types by Groovy programmers:

- **Question Q1: Do programmers use types more often in the interface of their modules?** We believe that the benefits of the use of types are more clear in declarations that define the interface of modules. In such cases, programmers are specifying how the rest of the program should interact with a module, potentially improving the readability and maintainability. On the other hand, in declarations that are hidden from external modules, we expect programmers to opt for the simplicity and flexibility offered by untyped declarations more often.
- **Question Q2: Do programmers use types less often in test classes and scripts?** Many studies analyze typing paradigms in the main classes of a program. However, little is known about this question in scripts or test classes. We want to understand if programmers consider different typing strategies in these scenarios.
- **Question Q3: Does the experience of programmers with other languages influence their choice for typing their code?** We believe that programmers familiar with a dynamically typed

<sup>1</sup> <http://github.com/carlosgsouza/groovonomics>

language are more comfortable with the lack of types and end up using types less often in Groovy.

- **Question Q4: Does the size, age or level of activity have any influence on the usage of types?** We hypothesize that as these metrics grow, there is an increased concern about keeping code more maintainable. This can lead programmers to use types more often as a means to improve code readability.
- **Question Q5: In frequently changed code, do developers prefer typed or untyped declarations?** It makes sense to assume that developers try to increase the maintainability of frequently changed code. One way to achieve that is improving the readability of such code with the use of types. On the other hand, the flexibility of untyped declarations is capable of increasing the changeability of those files. We want to understand which one of these strategies is actually preferred by Groovy developers in the end.

### 3.2 Data Collection Procedure

The projects used in this study were obtained from GitHub, a popular source control service based on Git. For each project, it was necessary to retrieve its source code, metadata, commit history, and the metadata of all of its developers. GitHub does not offer a listing of all hosted projects, but it offers two search mechanisms, a REST API and a Web based search page. Unfortunately, the GitHub API is too limited for our requirements. It imposes a limit of one thousand results and does not allow filtering projects by their programming language.

In order to retrieve an extensive dataset, it was necessary to write a bot to simulate human interactions with the GitHub webpage and search for projects. Some special care was necessary to make this work. For instance, because the number of results is limited to 1 thousand projects, we had to segment the queries. Multiple requests were made, each of them asking for the name of all projects created on a given month. Results were then combined into a single list. Another problem faced was that GitHub denies excessive requests from the same client. By adding artificial delays between requests, it was possible to overcome this limitation.

With the name of all projects in hands, it was then possible to use the GitHub REST API to query their metadata. That metadata also contains the identifiers, of the developers and of the commits of that project. Using those identifiers we once again used GitHub REST API and obtained the background of all developers and the file changes of all projects.

### 3.3 Dataset

Our dataset consists of 6638 projects with almost 9.8 million lines of code. Table 1 shows descriptive statistics for the size, age and number of commits of these projects. There are more than 1.5 million declarations of all types and visibilities in our dataset. More details about these declarations are shown in Tables 2 and 3. Note that most fields are declared with private visibility while most methods are public. This is explained by the fact that, in Groovy, the default visibilities for fields and methods are private and public respectively.

	Mean	Median	Sd	Max	Total
Size (LoC)	1471	529	4545	149933	9770783
Commits	31	5	175	6545	203375
Age (Days)	361	280	333	1717	2395441

**Table 1.** Characterization of Projects

The projects of our dataset were developed by 4481 people. While 96% of the projects were developed by small groups of 3

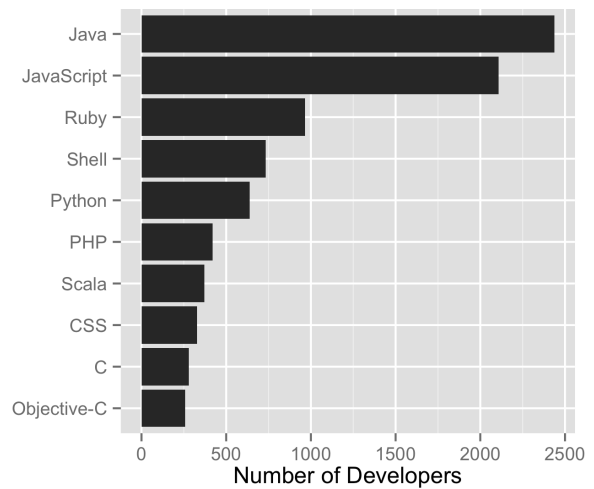
	Mean	Median	Sd	Max	Total
Field	54	19	163	5268	366148
Constructor Parameter	3	0	16	933	18956
Method Parameter	30	6	110	3554	202617
Method Return	53	15	165	4893	357997
Local Variable	88	21	361	16427	602645
Public	74	20	239	7942	507296
Protected	6	0	32	1394	42646
Private	58	21	178	5268	395776
All Declarations	227	71	744	29862	1548363

**Table 2.** Number of Declarations per Project

people or less, there were projects with up to 58 people involved. These developers have different backgrounds. Figure 1 shows what are the most popular languages used by the Groovy developers in other GitHub projects. Java is the most popular among them. Almost 2500 out of the 4481 developers of projects in our dataset also have Java projects hosted on GitHub.

	Private	Protected	Public
Field	346462	2996	16690
Constructor Parameter	680	246	18030
Method Parameter	27174	10897	164546
Method Return	21460	28507	308030

**Table 3.** Number of Declarations by Visibility



**Figure 1.** Most popular languages among Groovy developers

### 3.4 Analysis

In order to understand where programmers use types, we developed a static code analyzer based on the Groovy metaprogramming library. This analyzer is capable of retrieving the declaration information of parameters and returns of methods, parameters of constructors, fields and local variables. In addition, the analyzer can

tell if a declaration is part of a test class or a script and what is its visibility.

A relevant decision we made was not to compile projects, which would require all dependencies to be resolved. This is not feasible given the size of our dataset. Instead, we generated the AST for each file using the *CONVERSION* phase of the Groovy compiler. At this phase, the compiler has not tried to resolve any dependencies yet, but it is capable of generating an AST with enough information to determine whether a declaration is typed or not. This makes it possible to analyze each Groovy file separately without having to compile the whole project. The downside of this approach is that we can not analyze Groovy code in conjunction with its dependencies. For example, it is impossible to determine whether programmers tend to type code that interacts with other typed modules since we have not resolved any dependencies to these modules. However, our choice was fundamental in order to execute a study with such an extensive dataset.

## 4. Results

This section presents the data obtained from our analysis and its statistical treatment. We show how the usage of types varies according to the kind and visibility of declarations, nature of code, programmers' background, project maturity and frequency of changes. The interpretation of these results is left to Section 5, which answers the proposed research questions (Section 3.1) based on this data and raises new research questions.

### 4.1 Overall Result

Figure 2 shows a histogram and the descriptive statistics for the relative usage of types in declarations of projects. This value can vary from 0 (a project does not declare any types) to 1 (all declarations of a project are typed). All declarations are considered. Note that there is a significant number of projects for which the relative usage of types is either approximately 0 or 1. These are mostly small projects. About 95% of them have less than 1000 lines of code and 22% of them have less than 100 lines of code. In such projects, it is easier to be consistent on the typing strategy since there are just a few declarations. We initially considered not to include these projects in the rest of our analysis since they could not represent well the entire population of Groovy projects. However, doing so did not alter the results significantly and we decided to include all projects in our analysis regardless of their size. In the rest of this section, this data will be presented in more details so we can understand which factors lead programmers to use types or not.

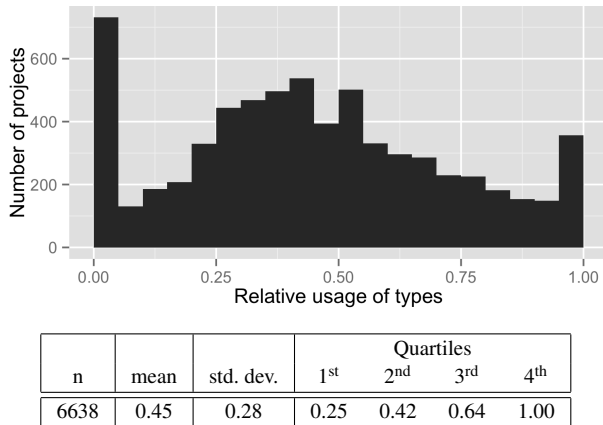
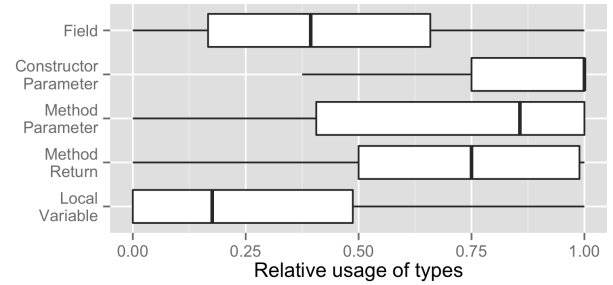


Figure 2. Usage of types in all declarations of all projects

### 4.2 Declaration Kind

This section investigates whether programmers use types differently depending on the kind of the declaration. For each project, we measured the relative usage of types by declaration type, which can be a field, a constructor parameter, the return or a parameter of a method or a local variable. These results are displayed in box plots in Figure 3 along with the corresponding descriptive statistics. Note that the size of each sample,  $n$ , is different since not all projects have all types of declarations. For instance, there are only 1670 out of 6638 projects with declarations of constructor parameters. Meanwhile, 6000 projects have declarations of fields.



Declaration Type	n	mean	median	std. dev.
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 3. Usage of types in all declarations by type of declaration

The results presented in Figure 3 suggest that programmers use types differently depending on the type of a declaration. Local variables, for example, are typed less often. Half of the projects have only 18% or less of their local variables typed. Conversely, methods and constructors are typed in most of the cases. Note that the median for constructor parameters is equal to 1.00, which means that, at least, half of the projects with constructor parameters type all declarations of this kind. Since local variables are never part of a module interface, these results suggest a positive answer for Question Q1, i.e., declarations that compose module interfaces are typed more often than other declarations.

The box plot graph and the descriptive statistics are not enough to determine whether the difference in the usage of types in any two kinds of declaration is significant. In order to do that, a significance test should be applied. We start by defining a hypothesis below, which can then be rejected or accepted by this test.

**H0** There is no difference in how programmers type different kinds of declarations

**H1** Programmers type their declarations differently depending on the kind of the declaration

The appropriate significance test should be chosen carefully. It needs to compare multiple treatments, which represent the 5 distinct kinds of declaration. We first considered applying repeated *t*-tests or the *Mann-Whitney U*-tests to compare every two kinds of declaration, i.e., fields vs local variables, fields vs method returns, etc. However, applying repeated tests over the same sample increases the probability of getting Type-I errors (rejecting the null hypothesis when it actually should be accepted).

A valid alternative for our scenario is to use One-Way Between Groups ANOVA, which compares all means simultaneously

and maintains the Type-I error probability at the designated level. ANOVA computes a  $p$  value which indicates whether at least two treatments are significantly different from each other. The smaller the value of  $p$ , the "more significant" is the difference and, consequently, the stronger the rejection of the null hypothesis. Given the level of significance,  $\alpha$ , we can reject the null hypothesis if  $p < \alpha$ . Typically,  $\alpha = 0.05$  or  $\alpha = 0.01$  are used, but in this study we decided to use a very small value for this purpose,  $\alpha = 0.001$ . This value might seem too small at first, which would require the difference between two treatments to be unnecessarily high in order to be considered significant. However, since we are analyzing such a large dataset, this value of  $\alpha$  seems reasonable [21]. For the treatments described in Figure 3 the  $p$  value reported by ANOVA is 0. This allows us to strongly reject the null hypothesis, even though we are using such an extreme value for  $\alpha$ , and state that at least two treatments are different from each other.

The results above show a very clear influence of the kind of variable over the usage of types. However, it is also desirable to know which kinds of variables are different from each other and how different they are. For this purpose, we apply the Tukey Honest Significant Differences (Tukey HSD) test in conjunction with ANOVA. This method calculates, for every two treatments, a  $p$  value indicating whether they are significantly different. It also reports a confidence interval for the difference between the means of these two treatments. The results of the Tukey Honest Significant Differences are displayed in Table 4. Confidence intervals were calculated with a confidence of  $0.999 (1 - \alpha)$ .

		p	Difference
Local Variable	Constructor Parameter	0	(-0.55, -0.47)
Field	Constructor Parameter	0	(-0.41, -0.34)
Method Parameter	Constructor Parameter	0	(-0.17, -0.10)
Method Return	Constructor Parameter	0	(-0.16, -0.09)
Local Variable	Field	0	(-0.16, -0.11)
Method Return	Field	0	( 0.22, 0.26)
Method Parameter	Field	0	( 0.22, 0.27)
Method Parameter	Local Variable	0	( 0.35, 0.40)
Method Return	Local Variable	0	( 0.36, 0.41)
Method Return	Method Parameter	0.95	(-0.02, 0.03)

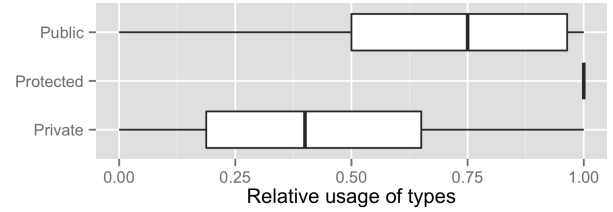
**Table 4.** Tukey Honest Significant Differences Test results for the comparison between the usage of types by kind of declaration

The table above shows that there are only two kinds of declaration for which there is no significant difference, parameters and returns of methods. This result is reasonable. Since returns and parameters of methods are declared together as part of a method signature, programmers probably use the same typing strategy in both declarations. All other declaration types can be considered significantly different from each other. In particular, note that these results clearly show that local variables and constructor parameters are the least and most typed declarations respectively. Another interesting insight provided by these results is that parameters of methods and parameters of constructors are typed differently. Although these are essentially the same type of declaration in Groovy, they seem to be perceived differently by programmers when it comes to typing.

### 4.3 Declaration Visibility

This section presents an analysis about how programmers use types according to the visibility of a declaration. We follow the same approach from the previous section. Figure 4 shows the box plots for the usage of types per declaration visibility along with the descriptive statistics. The ANOVA test reported  $p$  value equal to 0 for these treatments, allowing us to strongly reject the null

hypothesis. Finally, results of the Tukey HSD Test are reported in Table 5. These results show that all treatments are different from each other since all  $p$  values are equal to 0.



Declaration Visibility	n	mean	median	std. dev.
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 4.** Usage of types in all declarations by type of declaration

Protected declarations are those typed most often. Note how skewed is the distribution for these elements in Figure 4. Almost all 2387 projects which use protected visibility in their declarations have all of their protected fields, methods and constructors typed. The confidence intervals reported by the Tukey HSD test show very large differences between these declarations and those with either private or public visibility. Although public declarations are not typed as much, they are also typed very often. At least half of the projects type 75% or more of their declarations. Conversely, private declarations are those with the smallest relative use of types. These results suggest a positive answer for Question Q1, which hypothesizes that declarations that are part of a module definition are typed more frequently.

		p	Difference
Protected	Private	0	( 0.47, 0.52)
Public	Private	0	( 0.24, 0.28)
Public	Protected	0	(-0.27, -0.22)

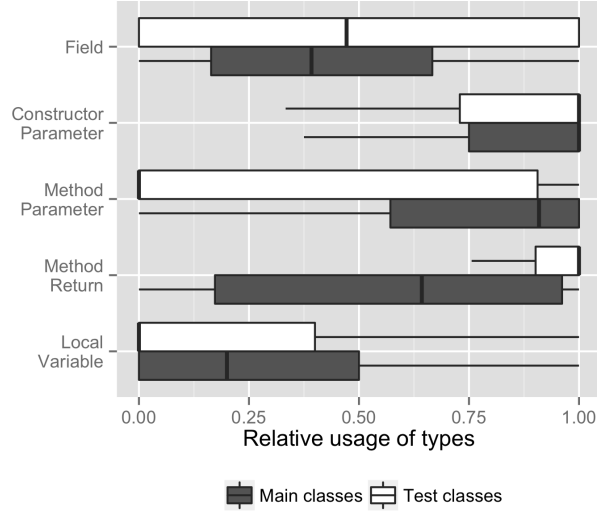
**Table 5.** Tukey Honest Significant Differences Test results for the comparison between the usage of types by visibility of declaration

### 4.4 Test Classes and Main Classes

We now analyze the use of types in test classes in comparison to main classes. A simple heuristic to determine if a class was written as a test class or a main class is used. In Groovy, like in Java, it is common to organize test classes and main classes in different source folders. The convention adopted by popular build tools among Groovy programmers, such as Gradle and Maven, assumes that test classes and main classes are in the *src/test/groovy* and *src/main/groovy* directories respectively. Based on these conventions, we can assume that all classes inside a *test* directory, but not in a *main* directory, are test classes.

For every project, we measured the usage of types in test classes and main classes. Script files are not considered in this analysis. We found test classes in 4350 of the 6638 projects in our dataset. Results are displayed in Figure 5 and show the relative usage of types by declaration type. White and gray box plots correspond to test classes and main classes respectively.

In order to compare the usage of types in test and functional classes, we use a slightly different approach. In this analysis, there are two independent variables, the kind of declaration and the kind of class. Thus, we are required to use Factorial ANOVA, which is the generalization of the One Way ANOVA for multiple factors.



Declaration Type	Class Type	n	mean	median	std. dev.
Field	Test	1769	0.48	0.47	0.43
	Main	5857	0.43	0.39	0.33
Constructor Parameter	Test	124	0.77	1.00	0.41
	Main	1623	0.80	1.00	0.34
Method Parameter	Test	1524	0.34	0.00	0.43
	Main	4593	0.71	0.91	0.35
Method Return	Test	4334	0.85	1.00	0.31
	Main	5299	0.54	0.60	0.39
Local Variable	Test	2842	0.23	0.00	0.35
	Main	5548	0.30	0.19	0.32

**Figure 5.** Usage of types by declaration type in test classes and main classes

Multiple values of  $p$  are calculated by this test, one corresponding to the comparison of treatments according to one of the factors. We report the  $p$  value corresponding to the factor representing the kind of class. We also apply the Tukey HSD test, which results are displayed in Table 6. This time we want to show what is the difference of the relative usage of types between the same kinds of declaration, but in different kinds of classes. For example, the third row of Table 6 shows that there is a significant difference in how programmers type local variables in functional and test classes. The difference between the relative usage of types in functional classes and test classes falls in the  $(0.05, 0.11)$  interval.

The ANOVA test reported once again a  $p$  value equal to 0, implying that the usage of type is different in test and functional classes. Figure 5 and Table 6 show that this difference is significant for all kinds of declarations except constructor parameters. While local variables in main classes are not typed very often, they are typed even less in test classes. At least half of the projects type none of the declarations of this kind in test classes. The difference in declarations of parameters of methods is even more evident since they are often typed in main classes, but almost never typed in test classes. The confidence interval reported by the Tukey HSD Test in this case is  $(-0.36, -0.44)$ . The large width of the box plots for fields and method parameters is noteworthy. This indicates that many projects type either almost all or none of these declarations.

Curiously, method returns are significantly more typed in test classes. The difference reported by the confidence interval in Table 6 for this case is  $(-0.31, -0.26)$ . At least half of the projects type all

of their method returns in test classes. Although counter intuitive this result can be easily explained. Automated testing frameworks usually enforce a certain method signature for test methods. JUnit for example, which is used in 2525 of the 4350 projects with test classes, requires test methods to be typed as *void*. Other popular test frameworks, such as TestNG, have similar requirements. This implies that, in this case, developers type their methods not because they want to, but because they need to.

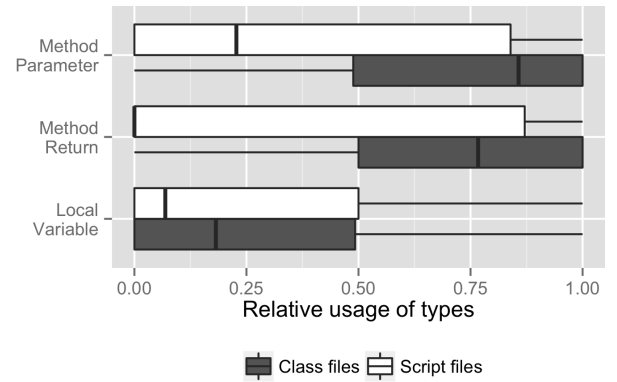
Declaration Type	p	Difference
Constructor Parameter	0.98	$(-0.08, 0.16)$
Field	0	$(-0.08, -0.02)$
Local Variable	0	$(0.047, 0.11)$
Method Parameter	0	$(0.36, 0.44)$
Method Return	0	$(-0.31, -0.26)$

**Table 6.** Tukey Honest Significant Differences Test results for the comparison between the usage of types by functional and test classes

#### 4.5 Script Files and Class Files

This section investigates how programmers type their code in script files. Similar to what was done in the previous section, we measured the usage of types in script files and class files in all projects and compared the obtained data. We do not consider test classes in this analysis. Determining whether a file corresponds to a script or a class is fairly simple. In Groovy, a script file is compiled into a class extending *groovy.lang.Script*. We consider all classes extending this class to be scripts.

Figure 6 shows the distribution of the relative usage of types in class files and script files. Note that constructors and fields are not considered since there is no way to declare those elements in scripts. Also, we do not present an analysis of declarations grouped by visibility since, although allowed, defining the visibility of a declaration inside a script does not make much sense.



Declaration Type	File Type	n	mean	median	std. dev.
Method Parameter	Script	504	0.40	0.23	0.42
	Class	4647	0.69	0.86	0.35
Method Return	Script	583	0.34	0.00	0.43
	Class	5662	0.70	0.77	0.30
Local Variable	Script	1775	0.28	0.07	0.37
	Class	5246	0.30	0.18	0.32

**Figure 6.** Usage of types by declaration type in script files and class files

The execution of the ANOVA test reported a  $p$  value equal to 0, revealing that the declarations are typed significantly different in script files. Table 7 displays the results for the Tukey HSD test, which provide detailed results by the kind of declaration. There is no significant difference on the usage of types in local variables. On the other hand, declarations of parameters or returns of methods are typed much less frequently in scripts. Note however that the value for the last quartile of these declarations is very high, superior to 0.8. This indicates that, although most projects prefer not to use types in method returns, there are a few projects that consistently type most of them. Along with the results of the analysis of test classes, these results show that the answer for Question Q2 is positive. There are large differences in how programmers type scripts and classes. Although it is not clear from our analysis what is the reason for such phenomena, we will raise some hypothesis in our discussion, presented in Section 5.

Declaration Type	p	Difference
Local Variable	0.39	(-0.01, 0.04)
Method Parameter	0	( 0.24, 0.35)
Method Return	0	( 0.30, 0.40)

**Table 7.** Tukey Honest Significant Differences Test results for the comparison between the usage of types in script files and class files

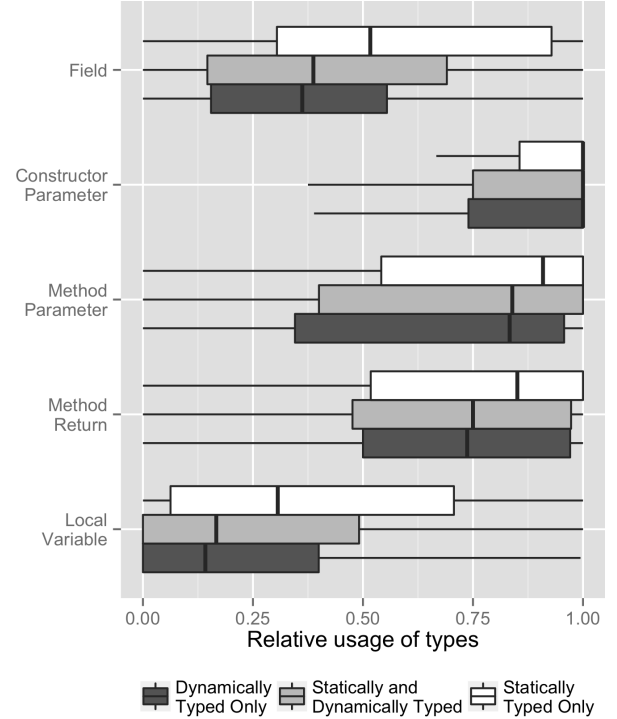
#### 4.6 Programmers' Background

In this section, we analyze how programmers use types in their declarations according to their backgrounds. Projects are distributed in three groups based on the type system of the languages their developers have used on GitHub. The first group comprises those projects of programmers who developed only in statically typed languages, such as Java or C#. The projects of those who developed only in dynamically typed languages, such as Ruby or JavaScript, comprise the second group. Finally, the third group is formed by the projects of those programmers with both dynamically and statically typed languages in their portfolio. We refer to these three groups by the names *Static Only*, *Dynamic Only* and *Static and Dynamic* respectively.

Figures 7 and 8 show results by declaration type and visibility. The  $p$  value reported by the ANOVA test is equal to 0, showing that there is a significant difference in how programmers with different backgrounds type their declarations. The results of the Tukey HSD test are reported in Tables 8 and 9. These tables are divided in three parts, each one corresponding to the comparison between the data of two of the three groups.

There are significant differences in the usage of types between all groups. These differences, however, are not as clear as those found in the previous analyses. Let's start with the comparison between projects in the *Static and Dynamic* and the *Dynamic Only* groups. There are significant differences only in private declarations and declarations of fields. Still, these differences are not very large, (0.01, 0.08) for fields and (0.02, 0.09) for private declarations. All in all, these two groups present very similar behavior when typing their declarations, apart from those two exceptions.

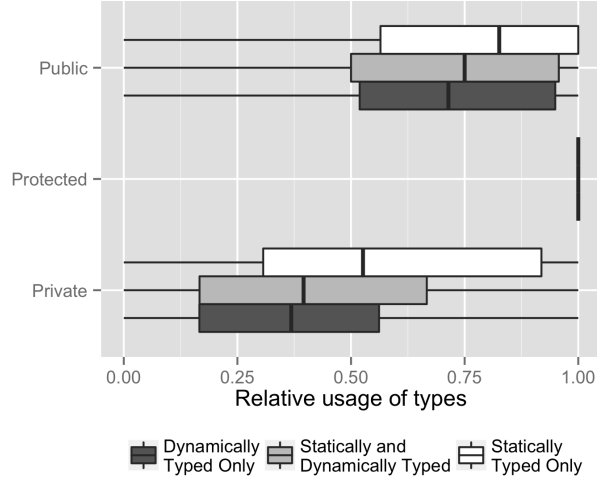
The comparison between the *Static* and the other groups shows more clear differences. Most of the  $p$  values reported by the Tukey HSD test are equal to 0. However, constructor parameters and protected declarations never present significant differences. This indicates a strong influence of these types of declarations over the programmers' behavior. There are other two cases which do not present significant differences, method returns and public declarations, both in the comparisons between programmers of the *Dynamic* and *Static* groups.



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

**Figure 7.** Usage of types by declaration type and programmer background

The results reported in this section suggest a positive answer for Question Q3, i.e., programmers use types differently depending on their background. This difference is larger when comparing those projects of the *Static* group with projects of the other groups. These results are statistically strong, but should be generalized with care. The detailed results reported by the Tukey HSD test reveal many exceptions, specially in the comparison between the groups comprising the projects of those programmers who have at least one dynamically typed language in their portfolio, *Static and Dynamic* and *Dynamic Only*. This comparison reveals that, for all kinds and visibilities of declarations, the difference is either not significant or small.



Declaration Type	Background	n	mean	median	std. dev.
Public	Static	757	0.73	0.83	0.29
	Both	3191	0.68	0.75	0.30
	Dynamic	1904	0.69	0.71	0.27
Protected	Static	287	0.92	1.00	0.20
	Both	1275	0.94	1.00	0.18
	Dynamic	825	0.93	1.00	0.20
Private	Static	787	0.56	0.53	0.34
	Both	3196	0.43	0.40	0.33
	Dynamic	2040	0.38	0.37	0.28

**Figure 8.** Usage of types by declaration visibility and programmer background

	Declaration Type	p	Difference
Static and Dynamic vs. Static	Field	0	(-0.17, -0.07)
	Constructor Parameter	1.0	(-0.12, 0.06)
	Method Parameter	0	(-0.11, 0.00)
	Method Return	0	(-0.11, -0.01)
	Local Variable	0	(-0.15, -0.05)
Dynamic vs. Static	Field	0	(-0.23, -0.12)
	Constructor Parameter	0.99	(-0.12, 0.07)
	Method Parameter	0	(-0.14, -0.02)
	Method Return	0.01	(-0.10, 0.00)
	Local Variable	0	(-0.18, -0.07)
Static and Dynamic vs. Dynamic	Field	0	( 0.01, 0.08)
	Constructor Parameter	1.00	(-0.07, 0.06)
	Method Parameter	0.84	(-0.02, 0.06)
	Method Return	0.94	(-0.05, 0.02)
	Local Variable	0.12	(-0.00, 0.06)

**Table 8.** Tukey Honest Significant Differences Test results for the comparison between the usage of types by declaration type and programmers background

#### 4.7 Project Size, Age and Number of Commits

This section investigates whether programmers use types differently in their code depending on the project characteristics. We analyze three project metrics: age, number of lines of code and number of commits. We start by analyzing the correlation between these

	Declaration Visibility	p	Difference
Static and Dynamic vs. Static	Public	0.00	(-0.10, 0.00)
	Protected	1.00	(-0.07, 0.09)
	Private	0.00	(-0.18, -0.08)
Dynamic vs. Static	Public	0.01	(-0.10, 0.01)
	Protected	1.00	(-0.08, 0.09)
	Private	0.00	(-0.23, -0.13)
Static and Dynamic vs. Dynamic	Public	0.98	(-0.04, 0.03)
	Protected	1.00	(-0.05, 0.06)
	Private	0.00	( 0.02, 0.09)

**Table 9.** Tukey Honest Significant Differences Test results for the comparison between the usage of types by declaration visibility and programmers background

metrics and the relative use of types in declarations by type and visibility. The Spearman rank correlation coefficient is used for this purpose. This coefficient, which ranges from -1 to 1, is a measure of the dependence between two variables. A positive value means that two variables are correlated, i.e., as the value of one grows, so does the value of the other. A negative value means an inverse correlation. Values close to 1 or -1 indicate very strong relationships and values above 0.5 or below -0.5 can be considered strong correlations.

Declaration Type/Visibility	Size	Age	Commits
Field	0.221	-0.063	0.153
Constructor Parameter	-0.072	-0.132	-0.053
Method Parameter	-0.123	-0.079	-0.004
Method Return	-0.071	0.168	-0.027
Local Variable	0.057	-0.049	0.112
Public	-0.063	0.119	-0.024
Protected	-0.286	-0.020	-0.165
Private	0.213	-0.068	0.160

**Table 10.** Spearman Correlation between the usage of types and the size, age and number of commits of projects

Table 10 shows the Spearman correlation coefficient between the usage of types and the age, size and number of commits of a project. Most of values in this table are close to 0. There are only a few which could indicate a very weak relationship, such as *Size* vs. *protected* or *commits* vs. *private*. All in all, these values don not seem to suggest any direct relationship between these metrics and the usage of types.

The lack of correlation between the relative usage of types and these metrics does not necessarily imply that they have no influence on the usage of types. A possibility is that this influence appears only in the most mature projects, where the values of all of these three metrics are high enough. In order to determine whether this is true, we conduct now a comparison between mature projects and the rest of the dataset. We define a mature project as a project that is 100 days old or more and has, at least, 2KLoC and 100 commits. These numbers were defined by manually inspecting our dataset and finding that there are popular and mature projects that barely exceed these three metrics. According to our criteria, there are 223 mature projects in our dataset, which are characterized in Table 11.

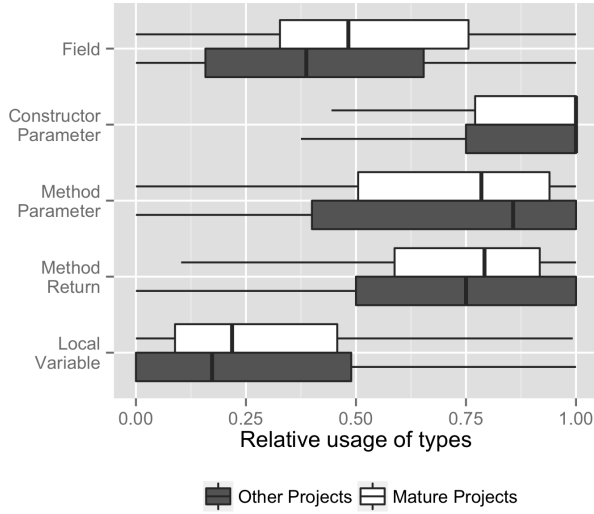
Figures 9 and 10 show the box plots for the usage of types in mature projects and others by declaration type and visibility respectively. The ANOVA test reported a *p* value equal to 0.0518 for this analysis, which implies that there are no treatments that are significantly different from each other. Because of this, we do not



	mean	median	std. dev.	max	total
Size (LoC)	9947	5627	14594	149933	2218189
Commits	487	213	800	6545	108583
Age (Days)	600	574	350	1469	133697

**Table 11.** Descriptive statistics for mature projects

report the results for the Tukey HSD test. This result, along with the very low correlation coefficients reported in Table 10, imply that the answer for Question Q4 is negative. There is no significant difference in how programmers type declarations in mature projects compared to declarations in other projects.



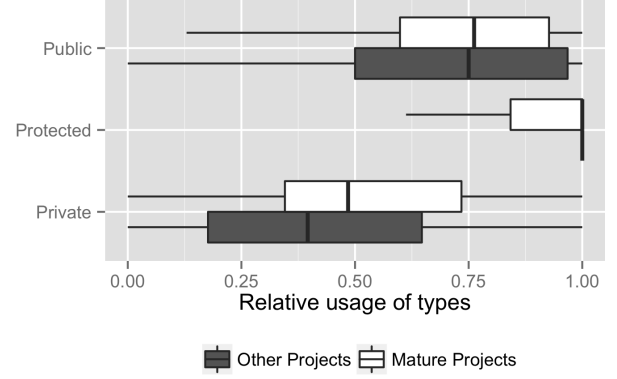
Declaration Type	Project Type	n	mean	median	std. dev.
Field	Mature	221	0.53	0.48	0.27
	Other	5779	0.43	0.39	0.33
Constructor Parameter	Mature	172	0.83	1.00	0.30
	Other	1498	0.80	1.00	0.35
Method Parameter	Mature	222	0.69	0.78	0.29
	Other	4645	0.67	0.86	0.37
Method Return	Mature	222	0.72	0.79	0.24
	Other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

**Figure 9.** Usage of types in projects by declaration type and project maturity

#### 4.8 Frequency of changes

This section investigates whether programmers prefer to type their declarations in frequently changed code or not. Only the mature projects defined in the previous section are considered since we would not be able to obtain meaningful results from small and young projects.

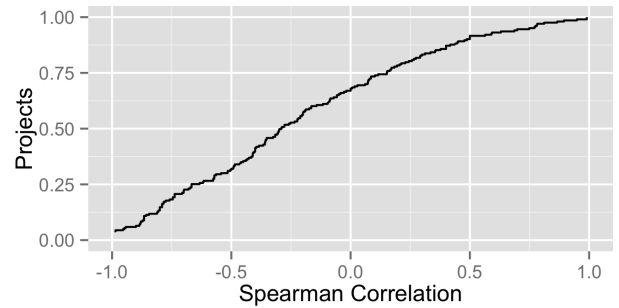
We found that in most of the mature projects programmers prefer untyped declarations in frequently changed code. We calculated the Spearman correlation coefficient between the frequency of changes of a file and the usage of types in that file for all mature projects. In projects where programmers prefer the usage of types in frequently changed files, this coefficient is positive, and negative



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

**Figure 10.** Usage of types in projects by declaration visibility and project maturity

when untyped declarations are preferred. Figure 11 displays the cumulative distribution of this coefficient across the mature projects dataset. It shows that 65% of them present a negative Spearman correlation coefficient and that almost half of these present strong negative correlations, i.e., inferior to -0.5. On the other hand, it can be said that only 10% of the mature projects present strong positive correlations.



**Figure 11.** Spearman ranking for the correlation between frequency of changes of files and the usage of types in mature projects

## 5. Discussion

In this section, we discuss the results of our study in the light of the research questions proposed in Section 3.1. Although we were able to get a good understanding of the usage of types in different contexts, the cause for such results is still unclear in many of them. We provide several hypothesis with the goal of identifying future research topics that can provide more detailed insights about such causes.

### **Q1: Do programmers use types more often in the interface of their modules?**

The analysis of the usage of types by kind and visibility of declarations, presented in Sections 4.2 and 4.3, shows evidence that the answer for Q1 is affirmative. Private declarations are typed less often than protected and public declarations. Also, fields, methods and constructors are typed more often than local variables. Although fields are significantly less typed than methods and constructors, this can be explained by the fact that most fields are declared privately as shown in Section 3.3. In Groovy, similar to what happens with Java, interactions with fields of other modules usually happen through accessor methods.

While it is clear that module definitions are typed more often, the cause for such phenomena is still open to discussion. We believe that the main motivation for this is the implicit documentation provided by types. In these scenarios, types provide useful hints about the behavior of modules [36] and define pre and post condition of contracts [22–27]. Users of an well defined module learn how to use it faster and do not need to read its implementation to understand how to use it. Programmers may consider that delicate contracts, such as those defined by protected methods, should be well documented and thus are typed more often.

Constructor parameters and protected declarations are always the most typed elements. In addition, the analysis of different contexts, i.e., functional and test classes, mature and non-mature projects and programmers' backgrounds, shows no significant difference in the usage of types. This reveals that the presence of these particular declarations are considered by programmers as more important factors in their choice whether to type a declaration than these contexts. We hypothesize that programmers consider documentation more important in these elements. Constructors usually define the dependencies of an object, at least for its creation. In addition, they might be the first element of a class which programmers interact with when dealing with a new module. Protected declarations are often used as a means to delegate the implementation of a method to subclasses, which requires an well defined contract so the superclass can work properly. Moreover, they give subclasses and other classes in the same package access to internal elements of a class, forming a tightly coupled relationship [28].

We can also speculate that declarations that are not part of a module definition, local variables or those with private visibility, require less documentation and thus are typed less often. Programmers can easily find all the references to these elements. Local variables are only used inside the block of code where they were declared, while all the references to elements declared privately are in the same file. This makes it easier for them to infer the types of such declarations even when it is not explicitly defined.

Documentation may not be the only reason why programmers type declarations in modules interfaces. We can think of at least other two reasons. First, a programmer might type a declaration so he or she can get code assistance from the development environment. For example, typing the declaration of a method parameter allows the development environment to provide code completion for that parameter inside the method. Another possibility is that, even though Groovy is actually a dynamically typed language, programmers might type their declarations thinking that the compiler will check for type errors, which would lead to safer interactions between modules

### **Q2: Do programmers use types less often in test classes and scripts?**

There are notable differences between the usage of types in either test classes or scripts and the main classes of a program. Sections 4.4 and 4.5 show that, in these scenarios, programmers use types less often. If we are right about our hypothesis that programmers

type their modules as a means to document their code, this could explain this less frequent use of types. Scripts and test classes are usually not designed as reusable modules. Test classes have the sole goal of verifying a program's functionality and not interfering with it while scripts can not be instantiated or referenced by other modules. In these scenarios, programmers might perceive documentation as less important. It is curious however that test code itself is usually perceived as a form of documentation [37, 38]. Because of this, we were expecting programmers to actually use more types in test classes as a means to improve the documentation provided by them. Perhaps, although programmers use test classes as documentation, they might not write them with this goal in mind.

An alternative explanation for the fact that scripts and tests are typed less frequently is that most of them are probably simpler than functional classes. As found in the recent work of Hanenberg et al [34], untyped code potentially has positive impacts on the development time of easier tasks. In such case, programmers might not type their declarations in scripts or test classes since this would allow them to finish their tasks faster.

### **Q3: Does the experience of programmers with other languages influence their choice for typing their code?**

The analysis presented in Section 4.6 indicates that the answer for Question Q3 is affirmative. The choice for using types on a language with optional typing, such a Groovy, is in fact influenced by the programmers' experience with other languages. In general, those programmers who have only statically typed languages in their portfolio type more often than all others. However, the two groups with projects written by programmers who have either statically and dynamically typed languages or only dynamically typed languages are very similar in most cases. Apparently programmers who develop in an "untyped" language get used to the lack of types, leading them to declare types less often. This hypothesis supports the work of Daly et al which suggests that programmers have ways of reasoning about types that compensate for the lack of static type information [7].

### **Q4: Does the size, age or level of activity have any influence on the usage of types?**

We initially believed that, as these metrics grow, the maintenance of projects becomes more difficult, leading programmers to use more types as a means to make code more readable. However, the analysis presented in Section 4.7 shows no evidence of such behavior. We consider two hypothesis in order to explain these results. First, the considered metrics might not actually correlate to the need for maintenance of projects. Second, even if these metrics are a good indicative of the necessity of better maintainability, once projects start growing and aging, programmers might not have the opportunity or desire to make their code more maintainable.

### **Q5: In frequently changed code, do developers prefer typed or untyped declarations?**

In frequently changed code, there are arguments in favor and against using types. Since types act as documentation, programmers might use them to make code more maintainable and easier to change [10]. On the other hand, untyped code is simpler and can potentially be changed faster [14]. The results presented in Section 4.8 however show that the latter is considered more often than the first.

In most projects, the usage of untyped declarations grows as the frequency of changes in a file increases. Apparently, programmers understand that untyped code makes maintenance tasks easier. One can argue that the causal relationship is the opposite, i.e., these files have to be changed more often due to the use of untyped declarations. However, it is very unlikely that programmers would not no-

tice that untyped declarations would be causing such problems and would not add types to those declarations in order to fix them.

## 6. Threats to Validity

In this section, we discuss potential threats to validity of our study. As usual, we have arranged possible threats in two categories, internal and external validity [31].

### Internal Validity

Perhaps the most relevant internal threat to our study is that in a large scale empirical study such as ours, there might be many confounds which are difficult to identify. In Section 4.6 we consider that the GitHub portfolio of a programmer represents well his experience with other languages and type systems, but this might not be true for all programmers. They may have projects in their portfolio that they have not worked on or projects hosted elsewhere written in other languages. There is still the possibility of a programmer having multiple GitHub accounts with different languages in each one, causing this programmer to be measured twice with different inferred backgrounds. Due to the large number of programmers considered in our study, we expect these special cases not to have a large influence over the results.

There are other factors which might have influenced programmers besides the ones considered in this study. Some frameworks require programmers to use typed or untyped declarations in some cases. For example, we found that the data collected in test classes is biased by the fact that popular testing frameworks, such as JUnit, require test methods to have their returns declared as *void*. There might be other similar cases that we are not aware of.

### External Validity

Although we have analyzed a very extensive number of Groovy projects, it can not be said that we have all possible scenarios covered. By manually inspecting our dataset, we could find only a few projects with characteristics of software developed inside an organization. Most of them were developed by small groups of people or open source communities. Such enterprise projects are probably hosted privately on GitHub or in private servers, hence unavailable to us.

The behavior observed for Groovy projects can be very different in other languages. Most languages are not like Groovy and feature either static or dynamic typing, forcing programmers to choose a single typing strategy for all scenarios in a single project. Even a language with a hybrid typing paradigm might implement different strategies which will be perceived differently by the programmers of that language. Finally, the tools used to code in a given language might influence programmers to chose different type strategies.

## 7. Related Work

There are multiple studies in the literature which compare different typing strategies. Although we are not aware of any studies that analyze this question using a large scale case study as ours, we know of multiple controlled experiments with significant results. One of the goals of this paper is to complement theses results by providing a different point of view to their analyses.

In a recent experiment, Hanenberg et al. [34] studied the impact of the use of types on the development time of programmers while performing tasks on an undocumented API. The experiment divided 27 people in two groups. They developed in two languages, one statically typed, Java, and the other dynamically typed, Groovy. Results revealed a positive impact of the use of types when these were used to document design decisions or when a high number of classes had to be identified by programmers. On the other hand, for easier tasks, programmers developed faster in the dynamically

typed language. Our analysis suggest that programmers are aware of these tradeoffs and consider them when choosing whether to type or not their declarations. In potentially simpler scenarios, such as scripts and tests, we show a lower usage of types. Conversely, programmers type the interface of their modules very often, probably as a means to document their code. This implicit documentation potentially improves the development time in more complex scenarios, such as those where programmers benefited from types in [34].

In experiments conducted by Daly et al. [7], the authors compare the performance of two groups working on small development tasks. One group used Ruby, a dynamically typed language, while the other used DRuby, a statically typed version of Ruby. Results showed that the DRuby compiler rarely managed to capture any errors that were not already evident for programmers. Most subjects involved in the study had previous experience with Ruby, which suggests that programmers get used to the lack of typing in their declarations. Our analysis of the programmers' backgrounds supports this argument. It shows that those programmers which have worked with dynamically typed languages in fact use types less often.

## 8. Conclusions and Future Work

In this paper we conducted a large scale case study in order to investigate how types are used in Groovy. Our goal is to find what are the points of views of programmers about what typing strategies are more suitable to which contexts. The main findings of this study are:

- Groovy programmers type declarations that define the interface of modules more often than other declarations
- Types are less popular in test classes and script files
- Those programmers who have developed in at least one dynamically typed language use types less frequently than those who have only worked with statically typed languages
- Apparently, there is no influence of the size, age or level of activity of a project on how programmers use types
- In most projects, the files that change more frequently are also those files with a lower usage of types

We believe that these results are valuable to developers of programming languages and development tools who can base their designs on real user data. Also, programmers can understand the tradeoffs between using or not types in their projects. Our results provide a different point of view and complement previous studies which analyzed typing strategies through the use of controlled experiments. We hope that the questions raised in our discussion can inspire other researchers to analyze our findings in more details.

In future work we want to conduct controlled experiments and qualitative studies in order to evaluate some of our results in more details. In particular, we have started a new study with the goal of finding what are the impacts of the use of the documentation provided by types and the documentation provided by unit tests on the development time of maintenance tasks. Another work in progress is a framework for the static source code analysis of large numbers of repositories called Elastic Repository Analysis - ERA<sup>2</sup>. This framework is based on the artifacts we built to analyze the Groovy projects in this study. ERA will help researchers to retrieve large numbers of projects from GitHub and process these projects quickly on the cloud using Amazon Web Services.

<sup>2</sup> <http://github.com/carlossouza/ERA>

## Acknowledgments

Many thank to Stefan Hanenberg for the invaluable contributions.

## References

- [1] Tiobe programming community index. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Accessed in 23/09/2013.
- [2] Groovy Programming Language. <http://groovy.codehaus.org/>. Accessed in 10/10/2013.
- [3] Bruce, K. (2002). Foundations of object-oriented languages: types and semantics. MIT press.
- [4] Bruch, M., Monperrus, M., and Mezini, M. (2009). Learning from examples to improve code completion systems. In Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundati- ons of software engineering, pages 213222. ACM.
- [5] Cardelli, L. (1996). Type systems. ACM Comput. Surv., 28(1):263264.
- [6] Chang, M., Mathiske, B., Smith, E., Chaudhuri, A., Gal, A., Bebenita, M., Wimmer, C., and Franz, M. (2011). The impact of optional type information on jit compilation of dynamically typed languages. SIGPLAN Not., 47(2):1324.
- [7] Daly, M. T., Sazawal, V., and Foster, J. S. (2009). Work In Progress: an Empirical Study of Static Typing in Ruby. In Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU), Orlando, Florida.
- [8] Hanenberg, S. (2010). An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. SIGPLAN Not., 45(10):2235.
- [9] Kleinschmager, S., Hanenberg, S., Robbes, R., and Stefik, A. (2012). Do static type systems improve the maintainability of software systems? An empirical study. 2012 20th IEEE International Conference on Program Comprehension (ICPC), pages 153 162.
- [10] Lamport, L. and Paulson, L. C. (1999). Should your specification language be typed. ACM Trans. Program. Lang. Syst., 21(3):502526.
- [11] Mayer, C., Hanenberg, S., Robbes, R., Tanter, E., and Stefik, A. (2012). Static type systems (sometimes) have a positive impact on the usability of undocumented software: An empirical evaluation, 18:5.
- [12] Pierce, B. (2002). Types and programming languages. MIT press.
- [13] Tratt, L. (2009). Chapter 5 dynamically typed languages. volume 77 of Advances in Computers, pages 149 184. Elsevier.
- [14] Siek, Jeremy, and Walid Taha. Gradual typing for objects. ECOOP 2007 Object-Oriented Programming. Springer Berlin Heidelberg, 2007. 2-27.
- [15] Gray, K. E. Safe cross-language inheritance. ECOOP 2008 - European Conference on Object-Oriented Programming, pp. 5275.
- [16] Gray, K. E. Interoperability in a scripted world: Putting inheritance & prototypes together. In Foundations of Object-Oriented Languages (2011).
- [17] Gray, K. E., Findler, R. B., Andflatt, M. Fine-grained interoperability through contracts and mirrors. In Object-Oriented Programming, Systems, Languages, and Applications (2005), pp. 231245.
- [18] Siek, J., Andtaha, W. Gradual typing for objects. In European Conference on Object-Oriented Programming (2007), pp. 227.
- [19] Takikawa, Asumu, et al. "Gradual typing for first-class classes." ACM SIGPLAN Notices. Vol. 47. No. 10. ACM, 2012.
- [20] Fowler, M.. Domain-specific languages. Pearson Education, 2010.
- [21] Labovitz, S. "Criteria for selecting a significance level: A note on the sacredness of. 05." The American Sociologist 3.3 (1968): 220-222.
- [22] Meijer, E., and Peter D. "Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages." OOPSLA, 2004.
- [23] Wadler, Philip, and Robert Bruce Findler. "Well-typed programs cant be blamed." Programming Languages and Systems. Springer Berlin Heidelberg, 2009. 1-16.
- [24] Plosch, Reinhold. "Design by contract for Python." Software Engineering Conference, 1997. Asia Pacific and International Computer Science Conference 1997. APSEC'97 and ICSC'97. Proceedings. IEEE, 1997.
- [25] Flanagan, C. "Hybrid type checking." ACM Sigplan Notices. Vol. 41. No. 1. ACM, 2006.
- [26] Meyer, B. Object-oriented software construction. Vol. 2. New York: Prentice hall, 1988.
- [27] Furr, Michael, et al. "Static type inference for Ruby." Proceedings of the 2009 ACM symposium on Applied Computing. ACM, 2009.
- [28] Chidamber, Shyam R., and Chris F. Kemerer. "A metrics suite for object oriented design." Software Engineering, IEEE Transactions on 20.6 (1994): 476-493.
- [29] ISO, ISO, and IEC FCD. "25000, Software Engineering-Software Product Quality Requirements and Evaluation (SQuaRE)-Guide to SQuaRE." Geneva, International Organization for Standardization (2004).
- [30] Fenton, Norman E., and Shari Lawrence Pfleeger. Software metrics: a rigorous and practical approach. PWS Publishing Co., 1998.
- [31] Wohlin, Claes, et al. Experimentation in software engineering. Springer Publishing Company, Incorporated, 2012.
- [32] Gannon, J. D. An experimental evaluation of data type conventions. Commun. ACM 20, 8 (1977), 584595.
- [33] Prechelt, L., and Tichy, W. F. A controlled experiment to assess the benets of procedure argument type checking. IEEE Trans. Softw. Eng. 24, 4 (1998), 302312
- [34] Hanenberg, Stefan, et al. "An empirical study on the impact of static typing on software maintainability." Empirical Software Engineering (2013): 1-48.
- [35] Bruce, Kim B. Foundations of object-oriented languages: types and semantics. The MIT Press, 2002.
- [36] Curtis, B. Five paradigms in the psychology of programming. MMC, 1987.
- [37] Beck, Kent. Test-driven development: by example. Addison-Wesley Professional, 2003.
- [38] Meyerovich, Leo A., and Ariel Rabkin. "Empirical Analysis of Programming Language Adoption." (2013).
- [39] Cohen, J. Statistical power analysis for the behavioral sciences (2nd ed.). Hillsdale, NJ: Lawrence Erlbaum.