How Do Programmers Use Optional Typing? An Empirical Study

Carlos Souza
Software Engineering Lab
Federal University of Minas Gerais (UFMG)
carlosgsouza@gmail.com

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

Abstract

The type paradigm is one of the most important things to be taken in consideration when choosing a programming language. This question has become increasingly important due to the recent popularization of dynamically typed languages such as Ruby and JavaScript. While, declaring types potentially improves the readability and maintainability of a program, it may require more work for the programmer. This paper presents a large scale empirical study with the goal of identifying programmers' points of views about these tradeoffs An analysis of the source code of almost seven thousand projects written in Groovy, a programming language which features optional typing, shows in which situations programmers prefer typing or not their declarations. Results suggest that the need for maintainability, frequency of change and programmers background are important factors in this decision.

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

General Terms Experimentation, Language

Keywords Type systems, static analysis, Groovy

1. Introduction

When choosing a programming language for a project, a developer considers several characteristics of that language. One of the most important of these characteristics is its type system, which can be static or dynamic. The type system determines when the type of a statement is defined [12]. Statically typed languages, such as Java and C#, require programmers to 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 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. Since the type system of dynamically typed languages is simpler, they tend to allow programmers to code faster [12] and adapt to frequently changing requirements more easily [14]. 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. For instance, 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 tradeoffs between type paradigms is an important matter. Programmers can make an informed decision by knowing which languages provide the right benefits for their particular context. Programming language developers can consider this information in their design so they can develop the most appropriate features for their target audience. Finally, tools can be developed or improved to overcome any weaknesses of a given language.

This paper presents a large scale empirical study about how programmers use optional typing in Groovy in order to understand which factors actually influence the decision of a developer for using types or not. This question was studied based on the analysis of a massive dataset with almost seven thousand Groovy projects. Through a static analysis of these projects, it was possible to understand when developers use types and then extract what are the factors that influence this decision.

Results show that programmers consider types as a means to document their code. This is even more evident on the definition of

the interface of modules. Conversely, when neither readability nor stability is a concern, programmers tend to type their declarations less often. In addition to that, programmers seem to prefer the flexibility of untyped declarations in frequently changed code. Finally, the experience of a programmer with other languages has a relevant influence on his or her choice for typing a declaration or not.

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 respectively. 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 22nd 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, however, 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 we try to invoke the method *toUpperCase* since the *Integer* class does not have this 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
         typedParam) {
8      def untypedVariable = 1.0
9      return untypedVariable
10   }
11
12   int typedMethod() {
13     String typedVariable = ""
14     return typedVariable
15   }
16 }
```

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.

Listing 2 A class written in Groovy

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

Trying to compile the class *TypeCheckedGroovyClass* produces an error since the method *sum* is supposed to receive two parameters of the type *int*, but it is actually called with two parameters of the type *String*.

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 }
```

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. Setudy Settings

The study presented in this paper consists in the static analysis of the source code of a corpus of 6638 Groovy projects with the goal of finding the factors that have an actual influence over the decision of a developer to type their declarations or not. In this section we present the research questions we want to answer, the data collection and analysis procedures as well as the characterization of the studied dataset.

3.1 Research Questoins

Static and dynamic type system have known advantages and disadvantages. We propose some research questions below in order to find which ones are actually considered by programmers in their code.

- **Question Q1: Do programmers use types to implicitly document their code?** By typing their declarations, developers are telling their colleagues (or themselves) about the nature of those declarations. This increases the readability and, hence, the maintainability of their code. If we are able to observe a higher usage of types in declarations that usually require more documentation, then we can assume that these developers indeed consider types as a means to document their code.
- **Question Q2: Are untyped declarations preferred when neither readability nor stability is a concern?** Besides readability, typing also contributes to stability. It reduces the risk of unexpected effects of modifications increasing the maintainability of the software system [30]. Our hypothesis is that, where these are not important factors, typing becomes less necessary and

developers prefer the simplicity and flexibility offered by dynamic typing.

- **Question Q3: 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 files. 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 developers in the end.
- **Question Q4: Does the previous experience of programmers with other languages influence their choice for typing their code?** We believe that programmers familiar with a statically typed language keep using types since they are used to it.

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, and each one 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 a 10 seconds delay 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 Table 2.

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

Table 1. Dataset characterization

More than 4 thousand developers were involved in the projects in our dataset. While 96% of the projects were developed by small groups of 3 people or less, there were projects with up to 58 people. These developers have different backgrounds. Figure 1 shows what are the most popular languages used by these developers in other GitHub projects. Java is the most popular among them. About 40% of the developers of projects in our dataset also have Java projects hosted on GitHub.

	Mean	Median	Std. Dev.	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	227	71	744	29862	1548363

Table 2. Number of Declarations

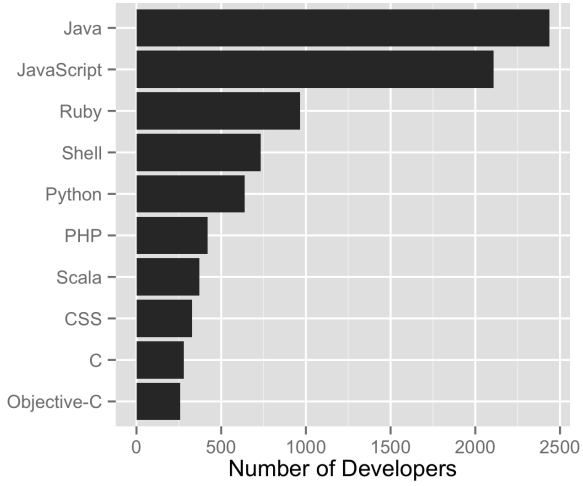


Figure 1. Most popular languages among Groovy developers

3.4 Analysis

In order to understand where programmers use types, we developed a static code analyzer based on the Groovy metaprogramming library. It is capable of retrieving the declaration information of parameters and returns of methods, parameters of constructors, fields and local variables. In addition, it 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 if a a declaration is typed or not. This makes it possible to analyze each Groovy file separately without having to compile the whole project.

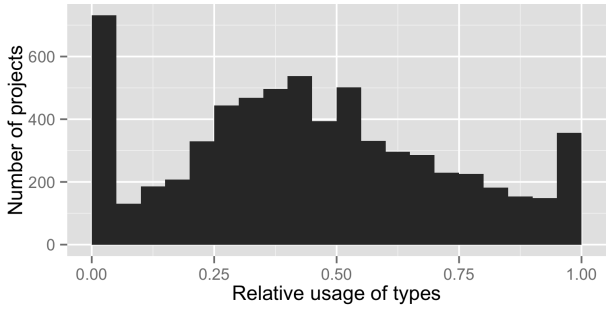
The downside of the approach described above is that we can not analyze Groovy code in conjunction with its dependencies.

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. Nevertheless, as shown in the next section, we were still able to obtain detailed and relevant results.

4. Results

4.1 Overall Result

Figure 2 gives an overview of the usage of types in all projects in our dataset. It shows the mean, standard deviation and the quartiles of the relative use of types in a project. This value can vary from 0, in case a project does not declare any types, to 1, in case all declarations of a project are typed. This result is shown in more details on a histogram in , which shows the distribution of the values of relative use of types across all 6638 projects.



			Quartiles			
n	mean	std. dev.	1st	2nd	3rd	4th
6638	0.45	0.28	0.25	0.42	0.64	1.00

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

4.2 Declaration Type

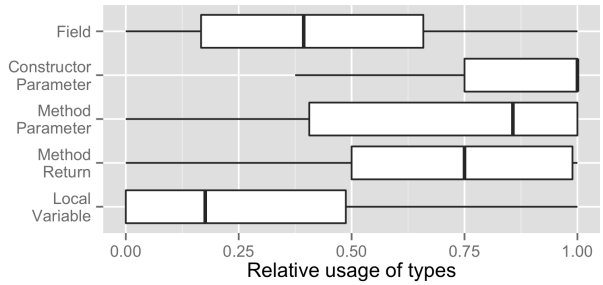
This section investigates whether programmers use types differently depending on the type of the declaration. For each project, we measured the relative usage of types by declaration type, which can be a field, a constructor parameter, the return or a parameter of a method or a local variable. These results are displayed in box plots in Figure 4.2 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.

The results presented in Figure 4.2 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 18% or less of their local variables typed. Conversely, methods and constructors are typed 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.

The box plot graph and the descriptive statistics are not enough to determine whether the difference in the usage of types in any two types 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.

H0 The data for two declaration types are drawn from the same population, i.e., there is no difference in how programmers type different types of declarations

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



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 appropriate significance test should be chosen carefully. We can not assume that the data presented in Figure 4.2 follows any particular distribution. In fact, an analysis of the box plots for declarations of local variables, methods and constructors shows that these samples are heavily skewed and thus do not follow a Normal distribution. Because of this, parametric tests such as the *t-test* can't be applied.

A valid alternative for our scenario is the paired Mann-Whitney U test, which works on ranks and doesn't require an underlying Normal distribution. It computes a *p-value* indicating whether two samples 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. Typically values of 0.05 and 0.01 are used to determine if the null hypothesis can be rejected or not. In this study, we use 0.001 for this purpose. This value might seem too small at first, which would require the difference between two samples to be too high in order to consider them different from each other. However, due to the large size of our dataset, this *p-value* seems reasonable [21]. In addition to *p*, the Mann-Whitney U test also reports a confidence interval describing the difference of the medians of the two populations. This interval can be used to measure how different are two samples. All of the confidence intervals in this study are calculated with a confidence of 99%.

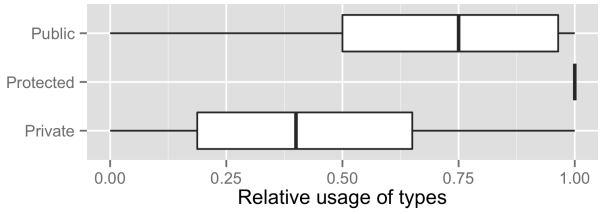
The significance test results are displayed in Table 4.2. It shows the *p* value and confidence interval reported by the Mann-Whitney U test for every two declaration types. There are only two types of declaration for which *p* values different from zero are reported, parameters and returns of methods. This allows us to reject the null hypothesis and consider the usage of types in these two declarations not significantly different. This is reasonable since returns and parameters of methods are declared together as part of a method signature. All other declaration types, however, can be considered significantly different from each other. This includes parameters of methods and constructors. Although these are essentially the same type of declaration in Groovy, they seem to be perceived differently by programmers when it comes to typing. Nevertheless, the difference between these two declaration types reported by the confidence interval is relatively small, $(-0.09, -0.06)$. Other interesting insights provided by these results are that local variables are the least typed declarations while constructor parameters are the most typed.

		p	conf. interval
Field	Constructor Parameter	0.0000	(-0.50,-0.44)
	Method Parameter	0.0000	(-0.32,-0.27)
	Method Return	0.0000	(-0.30,-0.26)
	Local Variable	0.0000	(0.13, 0.17)
Constructor Parameter	Field	0.0000	(0.44, 0.50)
	Method Parameter	0.0000	(0.06, 0.09)
	Method Return	0.0000	(0.08, 0.13)
	Local Variable	0.0000	(0.62, 0.67)
Method Parameter	Field	0.0000	(0.27, 0.32)
	Constructor Parameter	0.0000	(-0.09, -0.06)
	Method Return	0.0054	(0.00, 0.00)
	Local Variable	0.0000	(0.44, 0.50)
Method Return	Field	0.0000	(0.26, 0.30)
	Constructor Parameter	0.0000	(-0.13, -0.08)
	Method Parameter	0.0054	(0.00, 0.00)
	Local Variable	0.0000	(0.44, 0.47)
Local Variable	Field	0.0000	(-0.17, -0.44)
	Constructor Parameter	0.0000	(-0.67, -0.27)
	Method Parame	0.0000	(-0.50, -0.26)
	Method Return	0.0000	(-0.47, 0.17)

Table 3. Mann-Whitney U Test results for the comparison between the usage of types by type of declaration

4.3 Declaration Visibility

This sections presents an analysis about how programmers type their declarations according to the their visibilities. We follow the same approach from the previous section. Figure 4.3 shows box plots for the usage of types per declaration type along with the descriptive statistics. Declarations of fields, parameters and returns of methods and parameters of constructors of all projects are considered. Mann-Whitney U test results are presented in Table 4.3.



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

The null hypothesis can be rejected for all cases since the reported p values are always equal to 0. Protected declarations are those typed most often. Notice how skewed is the distribution for these elements in Figure 4.3. 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 U tests show a large difference 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.

		p	conf. interval
Public	Protected	0.0000	(-0.22,-0.18)
	Private	0.0000	(0.28, 0.31)
Protected	Public	0.0000	(0.18,0.22)
	Private	0.0000	(0.54,0.57)
Private	Public	0.0000	(-0.31,-0.28)
	Protected	0.0000	(-0.57,-0.54)

Table 4. Mann-Whitney U Test results for the comparison between the usage of types by visibility of declaration

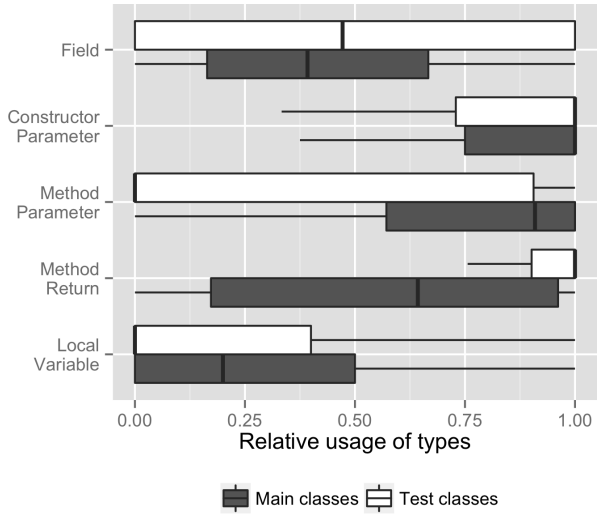
4.4 Test Classes and Main Classes

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's 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 test classes and main classes are in the *src/test/groovy* and *src/main/groovy* directories respectively. Projects written in Grails, an extremely popular web framework in the Groovy community, use a slightly different convention and organize the test classes in the *test* directory. Based on these conventions, we can assume that any 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 4.4 and show the relative usage of types by declaration type. White and gray box plots correspond to test classes and main classes respectively. We also applied the Mann Whitney U test, but this time however we are not comparing declaration types to each other. Instead, we are comparing declarations in test classes to declarations in main classes. Results for the U test are shown in Table 4.4. We don't show results grouped by visibility since this is usually not a concern when writing test classes.

According to Figure 4.4 and Table 4.4, the usage of types in test classes is very different for declarations of local variables and methods. 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 type 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 U test in this case is $(-0.533, -0.420)$. 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 4.4 for this case is (0.200, 0.250). 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	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

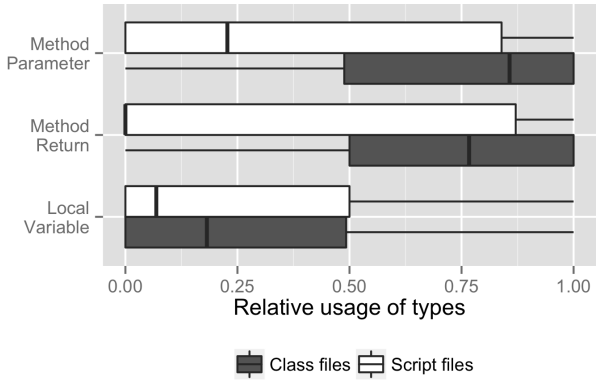
Declaration Type	p	conf. interval
Field	0.0020	(0.000, 0.000)
Constructor Parameter	0.4389	(0.000, 0.000)
Method Parameter	0.0000	(-0.533, -0.420)
Method Return	0.0000	(0.200, 0.250)
Local Variable	0.0000	(-0.070, -0.032)

Table 5. Mann-Whitney U Test results for the comparison between the usage of types by test classes and main 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. 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 and Table 4.5 show the results of our analysis. Note that constructors and fields are not considered since there is no way to declare those elements in scripts. Also, we don't present an analysis of declarations grouped by visibility since, although allowed, defining the visibility of a declaration inside a script doesn't make 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

Programmers type all kinds of declarations differently in scripts. According to the *p* values shown in Table 4.5, the null hypothesis can be rejected for all cases. The confidence interval reported for local variable declarations though is relatively small. This indicates a small difference for this type of declaration between scritps and classes. On the other hand, declarations of methods present a very different behavior. Unlike in classes, most of these declarations are not typed in scripts. This is even more clear in method returns, where most of projects do not type any of these declarations. 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 these declarations, there are a few projects that consistently type most of them.

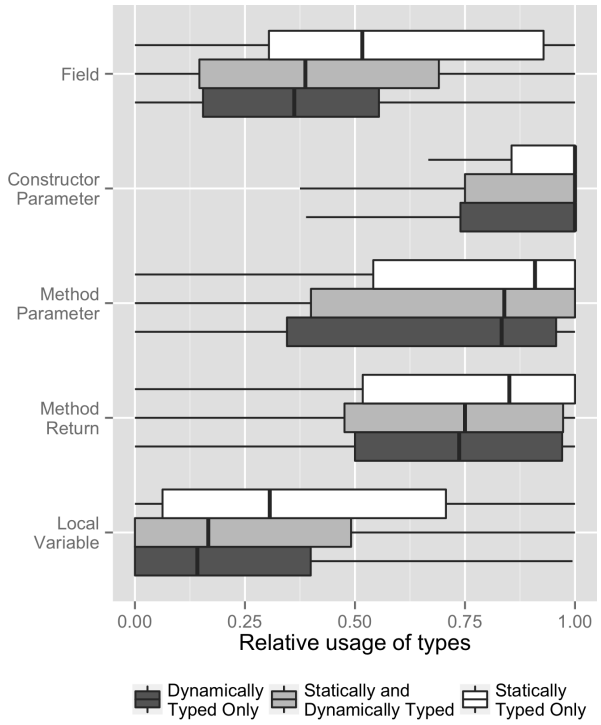
Declaration Type	p	conf. interval
Method Parameter	0.0000	(-0.385, -0.234)
Method Return	0.0000	(-0.500, -0.400)
Local Variable	0.0000	(-0.021, 0.000)

Table 6. Mann-Whitney U 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 project of those programmers with both dynamically and statically typed languages in their portfolio. Figures 4.6 and 4.6 show results by declaration type and visibility respectively. Results for the Mann-Whitney U test are reported in Tables 4.6 and 4.6. Notice

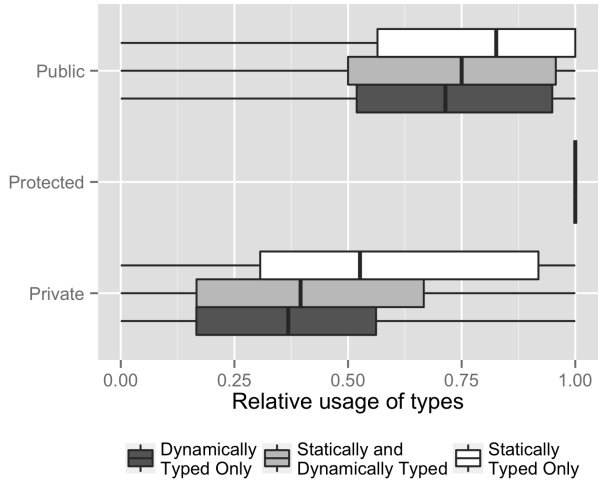
that these tables are divided in three groups, each one corresponding to the comparison between the data of two of the three different groups.



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 usage of types by programmers with both statically and dynamically typed languages in their portfolio is similar to that of programmers with dynamically languages only. According to Tables 4.6 and 4.6, the only significant differences in the usage of types are in declarations of fields or in those with public visibility. Still, the difference indicated by the confidence interval in these cases is arguably small, between 0 and 0.05 in both cases. On the other hand, there is evidence that programmers with only statically typed languages in their portfolio tend to use types differently from all others. There are only two situations where the p value reported by the Mann-Whitney U test is big enough so the null hypothesis



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	conf. interval
Static vs. Static and Dynamic	Field	0.0000	(0.073, 0.163)
	Constructor Parameter	0.7281	(0.000, 0.000)
	Method Parameter	0.0000	(0.000, 0.036)
	Method Return	0.0000	(0.000, 0.067)
	Local Variable	0.0000	(0.038, 0.100)
Static vs. Dynamic	Field	0.0000	(0.139, 0.228)
	Constructor Parameter	0.9347	(0.000, 0.000)
	Method Parameter	0.0000	(0.005, 0.076)
	Method Return	0.0000	(0.000, 0.062)
	Local Variable	0.0000	(0.057, 0.126)
Static and Dynamic vs. Dynamic	Field	0.0000	(0.000, 0.050)
	Constructor Parameter	0.7531	(0.000, 0.000)
	Method Parameter	0.0032	(0.000, 0.013)
	Method Return	0.5269	(0.000, 0.000)
	Local Variable	0.0071	(0.000, 0.005)

Table 7. Mann-Whitney U Test results for the comparison between the usage of types by declaration type and programmers background

can not be rejected, protected declarations and declarations of constructor parameters. In all other cases, these programmers use type more often than the others.

	Declaration Visibility	p	conf. interval
Static vs. Static and Dynamic	Public	0.0000	(0.000, 0.058)
	Protected	0.1287	(0.000, 0.000)
	Private	0.0000	(0.082, 0.167)
Static vs. Dynamic	Public	0.0000	(0.000, 0.065)
	Protected	0.1130	(0.000, 0.000)
	Private	0.0000	(0.143, 0.229)
Static and Dynamic vs. Dynamic	Public	0.0000	(0.000, 0.051)
	Protected	0.8260	(0.000, 0.000)
	Private	0.8328	(0.000, 0.000)

Table 8. Mann-Whitney U Test results for the comparison between the usage of types by declaration visibility and programmers background

4.7 Project Maturity

This section investigates whether programmers use types differently in their code depending on the project maturity. In this study, we consider three metrics to define the maturity of a project: age, number of lines of code and number of commits, which measures the level of activity of the project. We start by analyzing the correlation between these 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 can go 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 while 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 9. Spearman Correlation between the usage of types and the size, age and number of commits of projects

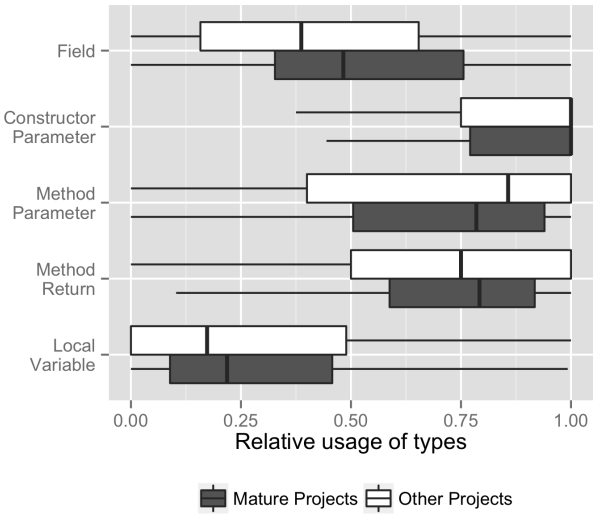
Table 4.7 shows the Spearman correlation coefficient between the usage of types and the three metrics of maturity. Most of values in this table are close to 0. There are a few which could indicate a weak relationship, such as *Size* vs. *protected* or *commits* vs. *private*, but these values are too small and could have happened by chance. All in all, we can not say that there is a correlation between the metrics of maturity and the usage of types in Groovy projects.

	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 10. Descriptive statistics for mature projects

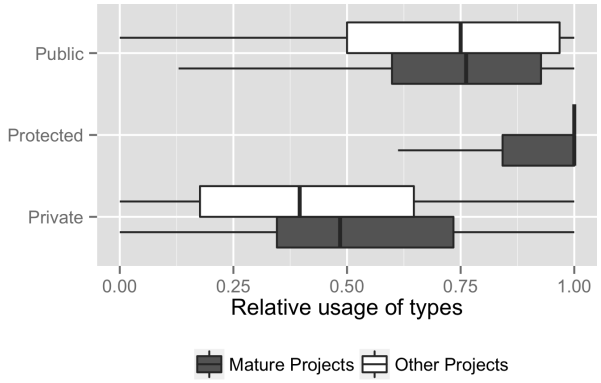
The lack of correlation does not necessarily imply that the maturity of a project has no influence on the usage of types. A possibility is that this influence appears only in the most mature projects, where the values of all of these three metrics are high enough. In order to determine whether this is true, we conduct now a comparison between the most 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 detailed in Table 10.

Figures 4.7 and 4.7 show the box plots for the usage of types in mature projects and others by declaration type and visibility respectively. Mann-Whitney U test results for the samples presented in these graphs are displayed in Table 4.7. There are significant differences in the usage of types only in two cases, declarations of fields and protected declarations. For private declarations, although the value of p is 0, the confidence interval for the difference of medians is negligible. These differences are too small, specially when compared to the differences found when analyzing programmers' background, script files and test classes. Hence, we can not conclude that programmers type their declarations differently depending on the maturity of a project, at least, when considering our definition of maturity, based on size, age and number of commits.



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

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



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

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

Declaration Type/Visibility	p	conf. interval
Field	0.0000	(0.053,0.173)
Constructor Parameter	0.3203	(0.000,0.000)
Method Parameter	0.0794	(-0.051,0.002)
Method Return	0.9373	(-0.031,0.051)
Local Variable	0.0002	(0.012,0.078)
Public	0.8540	(-0.031,0.045)
Private	0.0000	(0.000,0.000)
Protected	0.0000	(0.055,0.169)

Table 11. Mann-Whitney U Test results for the comparison between the usage of types by mature projects and others

4.8 Frequency of changes

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. On the other hand, untyped code is simpler and potentially faster to change. This section investigates what points of view are considered more often by Groovy programmers. 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 most of the mature projects 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 in case programmers prefer untyped declarations. Figure 4.8 displays the cumulative distribution of the 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 correlations, inferior to -0.5. On the other hand, it can be said that only 10% of the mature projects present strong positive correlations.

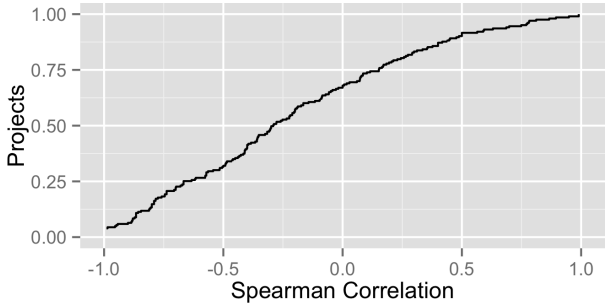


Figure 11. Spearman ranking for the correlation between frequency of changes of files and the usage of types in mature projects

5. Discussion

In this section, we discuss the questions proposed in Section 3.1 based on the results above. Section 5.1 shows that documentation is the main reason why Groovy developers use types, specially on the definition of their modules. An analysis of local variables, private declarations and test classes in Section 5.2 supports the hypothesis that untyped declarations are used more frequently when readability or stability are not a concern Section 5.3 discusses the higher usage of untyped declarations in most Groovy projects. Finally, Section 5.4 shows how programmers with experience using statically typed languages use types more frequently than other programmers.

5.1 Types as implicit documentation

Well documented code plays an important role in making code more readable, hence improving the code overall maintainability [30]. By analyzing the results presented in Section 4, we were able to find evidence that this is an important factor considered by Groovy programmers.

It is well known that the main advantages of using statically typed languages are implicit code documentation, static type verification and execution performance [5, 11]. Groovy however is not a statically typed language, but a dynamically typed language with optional typing (we are not considering the $@TypeChecked$ annotation in this study). This means that Groovy compiler can not verify types statically and that types still have to be checked during execution, which degrades runtime performance. Hence from the usual advantages of using a statically typed language, only code documentation holds true for Groovy. Considering the result shown on previous section, which shows that 60% of all declarations are typed, this means that code documentation is an important matter for Groovy developers.

It is possible to observe an even higher usage of typing in the definition of modules. As stated in related literature [22–27], types greatly aid in the definition of the contract of a module. They restrict the pre and post conditions of methods and define the nature of the properties of that module. Therefore, clients of a well typed module learn how to use the module faster and make less mistakes since the contract of that module is clear to them.

Declarations that define the interface of a module, i.e, fields, returns and parameters of methods and parameters of constructors, are more typed than the average. Figure **??** shows that, while most of these elements are typed, only 30% of the local variables, which do not contribute to the definition of a module, are typed. On the other hand, parameters of constructors, which are some of the most

important elements of a contract definition, are almost always typed [22].

The results presented above above are further refined by visibility in Figures **??**, **??**, **??** and **??**. They show that types are indeed considered more often on the definition of modules. First, notice that declarations of private fields or methods are always less typed than their public and protected counterparts. Private declarations are similar to local variables since they do not contribute to the interface of a module and are only accessible to the module where they were defined. In this case there is a smaller need for documentation and programmers do not feel as compelled to use types.

Public and protected elements define the interface of a method and thus tend to be typed often. In the case of parameters and returns of methods, the usage of types in declarations with protected visibility surpasses that of methods and fields with public visibility. In Groovy, protected declarations restrict access to internal elements of a class to its subclasses or other classes in the same package. It can be argued that the relationship between these classes is tightly coupled [29]. Apparently programmers understand that this type of contract is more delicate than others and that it should be well document.

The usage of types in public declarations is heavily influenced by the project size. Figure **??** shows this clearly. In projects with more than 65KLoC, these untyped declarations are seen with less than half the frequency observed in the smallest projects. However, no particular pattern can be observed for private or protected declarations, presented in Figures **??** and **??**.

Intuitively, the larger the project, the greater the difficulty of integration and the need for maintenance. This is more critical in public methods and fields. Unlike private or protected declarations, the number of modules to which a public methods or field is visible grows with the project size. This suggests that as projects grow, programmers realize that the documentation of public elements is more important and consider typing these declarations more often.

The project size has an opposite effect on the declarations of local variables. While public declarations get typed more often, the frequency of types in local variable declarations decreases. It is hard to tell the reason for this behavior. A possible explanation is that, as project grows, developers get more used to flexibility and objectivity of dynamic typing. They stop typing their declarations unless they understand they are really necessary.

5.2 Less typing when readability and stability are not a concern

When readability or stability are not a concern, the advantages of typing become less apparent and programmers seem to prefer the flexibility of untyped declarations. This is what we found when we analyzed how typing is used in local variables, private declarations and test classes. These are scenarios where we consider readability and stability to be less relevant.

Most local variables are untyped. Figure **??** shows that less than 30% of these variables are declared with a type, a value significantly smaller than the overall result for all declaration types, 60%. It can be assumed that typing local variables does not aid in the readability of the code as much as in other types of declarations. The value of a local variable is defined very close to its declaration, usually in the same statement. This makes it easier for a reader to understand the nature of a given variable and even infer its type. Programmers can be more objective in this scenario and eliminate the repetitive work of typing their variables.

Local variables have a smaller scope. They are only visible inside the block where they were instantiated and are destroyed once the execution of that block is over. Stability in this scenario is not as critical making the flexibility of untyped declarations more clear. For example, one can easily change the type of a local variable without having to worry about its use anywhere else.

A similar conclusion can be derived for declarations of private methods, fields and constructors. As shown in Figures **??**, **??**, **??** and **??**, untyped declarations are much more common in in these declarations that in their public or private counterparts. Private elements are usually more stable than others. Since they are only visible inside the class they were defined in, they can often be changed without having any unexpected effects outside that class. Also, since those elements are only referenced inside the class where they were defined, a reader can easily find those references and use them as a means to understand the code.

Another scenario with smaller need for readability and stability are test classes. These classes usually have simple relationships with the rest of the project, not having any clients and depending only on the classes under test. Test code is frequently considered a peripheral artifact of a software project and programmers usually do not worry about maintaining the quality of such code [28].

Test classes are also typed less often than the average. Figure **??** shows that declarations made inside the main classes of a project are typed in 68% of the cases. On the other hand, only 56% of the test classes are typed.

A counter example to this analysis is the usage of types in scripts. It can also be said that readability and stability are not big concerns in scripts. These are pieces of code usually written to solve punctual problems. They do not establish complex relationships with other modules and are not reused very often. Nevertheless, as shown in Table **??**, there is no significant difference between the overall usage of types in scripts and classes.

5.3 Frequently changed files have more untyped declarations

As stated in Question Q3, there are reasons to use typed and reasons to use untyped declarations in files that change often. Types improve the readability of code by implicitly documenting statements [10]. On the other hand, it is easier to change untyped code [14]. The results presented in Section 4.8 however show that the latter is considered more often than the first.

Figure 4.8 shows clearly that for 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 inverted, i.e., these files have to be changed more often because of problems generated by the use of untyped declarations. However, it is very unlikely that programmers would not notice that untyped declarations would be causing such problems and would not add types to those declarations in order to fix them.

An important conclusion that can be extracted from this analysis is that types are not absolutely necessary in order to build a maintainable system. At least, in most software projects written in Groovy, where there is a higher need for maintenance, programmers seem to prefer the flexibility of untyped declarations over the readability of types.

5.4 Programmers used to static typing use types more often

Results indicate that the answer for Question Q4 is affirmative. The choice for using types on a language with optional typing, such a Groovy, is in fact influenced by the experience with other languages. Those developers who have only static languages on their GitHub portfolio leave their declarations untyped less often than other developers. Figure **??** shows that these developers have only 28.7% of their declarations untyped compared to 40% of those developers with dynamic languages on their portfolio.

6. Threats to Validity

In this section, we discuss potential threats to validity. As usual, we have arranged possible threats in two categories, internal and external validity [32].

Internal Validity

The main threats to the internal validity involve the fact that, in such a large scale empirical study, we can not analyze data in much detail. 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 somewhere else 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 counted 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. I

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, one of the ways a programmer can create a stub object in Spock, a popular Groovy testing framework, requires programmers to type the mock declaration. In addition to that, the previous experience of programmers, which we have shown to have influence over programmers, might interfere with the analysis of other influencing factors.

In order to overcome the threats shown above, we are planning a controlled study as a future work. This will provide us with more detailed information, which we can use to complement the results of this empirical study.

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. It is probable that such enterprise projects are 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, a programmer must consider other features in the choice for a programming language like tool support, available libraries and the preferences of his or her organization. If these are the features considered in such choice, a programmer might end with the typing paradigm that is not ideal for his or her specific context.

7. Related Work

This work is based on a series of studies that 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.

In [8], the author compares the performance of two groups of students asked to develop two small systems. Both groups used a language developed by the author, Purity. The only difference in the language used by the two groups was the type system. One group used a statically typed version of Purity while the other used a dynamically typed version of the same language. Results showed that the group using the dynamic version was significantly more productive than the other. Similarly to this work, the author was able to compare two typing strategies directly while isolating any external factors. However, it can be argued that these results may not represent well real life situations in the software industry. This was a short duration study where students were used as examples of developers with no interaction with other programmers. In our study, we try to get more relevant results when analyzing source code developed by programmers during their normal activities.

In a follow up study [9], the authors obtained opposite conclusions. They compared the performance of two groups of developers in maintenance tasks. The first group used Java, a statically typed language, and the other used Groovy, but restricting developers to use only untyped declarations in order to simulate a dynamically typed version of Java. In this case, the group using the statically typed language, Java, was much more productive. This contradiction with a previous work reinforces the argument that the results of controlled studies are not reliable enough for this type of study.

In experiments conducted in [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. We tried to investigate this phenomenon, by analyzing how programmers use types depending on their experience with other languages.

8. Conclusions and Future Work

In this study, we investigated, from a practical point of view, what are the most influential factors on the choice of a programmer for using or not types. Understanding these factors can help programmers choosing the most appropriate programming language for a given context. This information can also be valuable for programming language designers who can base their design on real user data. To answer this question, we conducted a large scale case study with almost 7000 software projects written in Groovy, a dynamically typed language with optional type annotations.

We found evidence that programmers use types as a means to implicitly document their code. This tends to grow in the presence of some factors, such as the visibility, the complexity of the contract defined by such declaration and the size of the project where it was defined. Conversely, when readability or stability are not a concern, the simplicity and flexibility of untyped declarations seems to be preferred, as seen on local variables, private declarations and test classes. Another important factor is the previous experience of programmers with a given type system.

In future work we wish to analyze the influence of static and dynamic type systems over the robustness of software systems. In particular, we want to understand whether the use of dynamic typing, which limits the compiler's ability to find type problems, has any correlation with the occurrence of defects in the system and if the use of automated testing is able to reduce this correlation.

Acknowledgments

This work was partially supported by FAPEMIG, grants APQ-02376-11 and APQ-02532-12.

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 .,andStefik,A.(2012). Static type systems (sometimes) have a positive impact on the usability of undocumented software: An empirical evaluation. self, 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 2007Object-Oriented Programming. Springer Berlin Heidelberg, 2007. 2-27.

[15] Gray, K. E. Safe cross-language inheritance. In European Conference on Object-Oriented Programming (2008), 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. InObject-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, Martin. Domain-specific languages. Pearson Education, 2010.

[21] Labovitz, Sanford. "Criteria for selecting a significance level: A note on the sacredness of. 05." The American Sociologist 3.3 (1968): 220-222.

[22] Meijer, Erik, and Peter Drayton. "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, Cormac. "Hybrid type checking." ACM Sigplan Notices. Vol. 41. No. 1. ACM, 2006.

[26] Meyer, Bertrand. 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] Meszaros, Gerard. xUnit test patterns: Refactoring test code. Pearson Education, 2007.

[29] Chidamber, Shyam R., and Chris F. Kemerer. "A metrics suite for object oriented design." Software Engineering, IEEE Transactions on 20.6 (1994): 476-493.

[30] ISO, ISO, and IEC FCD. "25000, Software Engineering-Software Product Quality Requirements and Evaluation (SQuaRE)-Guide to SQuaRE." Geneva, International Organization for Standardization (2004).

[31] Fenton, Norman E., and Shari Lawrence Pfleeger. Software metrics: a rigorous and practical approach. PWS Publishing Co., 1998.

[32] Wohlin, Claes, et al. Experimentation in software engineering. Springer Publishing Company, Incorporated, 2012.