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 typing system is one of the most important things to be taken in consideration when choosing a programming language. This question has become increasingly more important due to the recent popularization of dynamically typed languages such as Ruby and JavaScript. This paper presents a large scale empirical study with the goal of finding what are the most influential factors for a programmer when choosing between different typing paradigms. An analysis of the source code of over 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 typing system, which can be static or dynamic. The typing system determines when the type of a statement is defined [12]. Statically typed languages, such as Java and C#, require the type of a statement to be defined by the programmer, which can then be used by the compiler to check for any typing errors. On the other side, in dynamically typed languages, like Ruby and JavaScript, the definition of the type of a statement only happens at run time.

Discussions about what is the best typing system for a particular situation have become increasingly more 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 the 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 dynamically typed languages are simpler, they 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 their 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 annotate their declarations with types. It is also possible to turn static type checking so the compiler can find type errors before execution. This allows developers to choose the most appropriate paradigm for each situation.

Understanding what are the most relevant factors on the choice for a typing system 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 unsing types or not. This question was studied based on the analysis of a massive dataset with more than 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 readability or stability are not 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. Some threats to the validity of this study are presented in Section 6 and related work are presented in Section 7. Finally, Section 8 concludes this study and suggests future works.

2. The Groovy Language

Groovy is a dynamically typed programming language which allows programmers to optionally type their declarations. It was designed to run on the Java Virtual Machine and its adoption has grown remarkably over the last years, specially among Java developers who seek more dynamism without having to learn a completely new language. It builds upon the strengths of Java, but has additional features inspired by dynamic languages such as Python and Ruby.

Despite having been launched less than ten years ago, Groovy is already a very popular language. 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.

The syntax of the Groovy language is similar to Java's and most of Java code is also valid in Groovy. 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.

Groovy was designed to be more expressive and concise than Java. Two implementations of a simple algorithm are shown below. Given a list of numbers, return a list containing only the even numbers of that list. Listing 1 shows the Java implementation while Listing 2 shows the Groovy counterpart.

Because of its high level of expressiveness, Groovy is able to reduce much of the boilerplate required in Java. Listing 2 shows that Groovy offers a native syntax for lists(lines 3, 6 and 14) and operator overloading (line 6). Semicolons are optional, except when there are multiple statements in the same line. When the keyword $return$ is omitted (line 10), the last expression evaluated with a method is returned. Also, parenthesis in method calls can often be omitted (line 16). In addition to that, Groovy implicitly imports frequently used classes, like those of the $java.util$ package, and methods, like $System.out.println$ (line 16).

The design of Groovy was influenced by dynamic features of programming languages such as Ruby and Python. Listing 3 shows how these features can be used to rewrite the same algorithm presented in Listing 1 in a single line of code. First, notice that the code shown in Listing 3 is a script, rather than a class file. It makes use of a closure to allow a programmer to define a filter logic. This closure is passed down to the $findAll$ method, which apply this closure to every element of the list to in order to decide if that element should be returned or not. Closures are one of the most important features of Groovy compared to Java. They allow a functional programming style of code, which is both expressive and powerful.

Metaprogramming is another dynamic feature present in Groovy. Listing 4 shows how to add a method to an existing class dynamically. By adding the method $evenNumbers()$ to the $List$ class, it is possible to achieve higher expressiveness. This is specially useful when implementing Domain Specific Languages [20].

Listing 1 A simple algorithm written in Java

```java
import java.util.ArrayList;
import java.util.List;

public class JavaFilter {
  List<Integer> evenNumbers(List<Integer>
      list) {
    List<Integer> result = new ArrayList<
        Integer>();
    for(int item : list) {
      if(item % 2 == 0) {
        result.add(item);
      }
    }

    return result;
  }

  public static void main(String[] args) {
    List<Integer> list = new ArrayList<
        Integer>();
    list.add(1);
    list.add(2);
    list.add(3);
    list.add(4);

    List<Integer> result = new JavaFilter().
        evenNumbers(list);
    System.out.println(result);
  }
}
```

When Groovy 1.0 was first launched, in 2007, it was a purely dynamically typed language. However, it allowed programmers to optionally type their statements. Examples of typed and untyped declarations combined flexibly in the same file are shown on Listing 5.

This kind of typing should not be confused with static typing since Groovy compiler does not use these type annotations to look for errors. For example, the snippet of code shown in Listing 6 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.

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 fail if it finds any.

Listing 7 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 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 side 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.

Listing 2 A simple algorithm written in Groovy

```
1  class GroovyFilter {
2    List<Integer> evenNumbers(List<Integer>
         list) {
3      List<Integer> result = []
4      for(int item : list) {
5        if(item % 2 == 0) {
6          result << item
7        }
8      }
9
10     result
11   }
12
13   public static void main(String[] args) {
14     List<Integer> list = [1, 2, 3, 4]
15     List<Integer> result = new GroovyFilter
           ().evenNumbers(list)
16     println result
17   }
18 }
```

Listing 3 A class written in Groovy

```
1  println([1, 2, 3, 4].findAll {it % 2 == 0})
```

Listing 4 An example of metaprogramming in Groovy

```
1  List.metaClass.evenNumbers = {
2    delegate.findAll {it % 2 == 0}
3  }
4  println([1, 2, 3, 4].evenNumbers())
```

Listing 5 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 }
```

Listing 6 A class written in Groovy

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

Listing 7 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 }
```

3. Setudy Settings

In this paper, we conduct a empirical study with the goal of understanding what are the factors that have an actual influence over the decision of a developer to type their declarations or not. We analyze the source code of more than 7 thousand Groovy projects and find where types are used. In the remainder of this section, we present the study settings starting by the research questions, listed in Subsection 3.1. The Data Collection procedure is described in Subsection 3.2 while the dataset itself is detailed in Subsection 3.3. Finally, we discuss the static code analyzer and the analysis procedure in Subsection 3.4.

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 statements, developers are telling their colleagues about the nature of those statements. This increases the readability and, hence, the maintainability of their code. If we are able to observe a higher usage of types in statements 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 readability or stability are not 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 [29]. Our hypothesis is that, where these aren't important factors, typing becomes less necessary and developers prefer the flexibility and objectivity 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 side, 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 considered 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 get 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, its metadata, its commit history, and the metadata of all developers involved in that project. 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 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 a expressive 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 identificators of the developers and of the commits of that project. Using those identificators 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 7268 Groovy projects obtained from GitHub. These projects include a total of 9.8 million lines of code and approximately 169 thousand Groovy files. There are about 412 thousand declarations considering variables, methods and fields. Figures 1 shows the cumulative distribution of the size of all these projects. Notice that a logarithm scale is used. Although most of the projects are relatively small, there are projects with up to 150 KLoC.
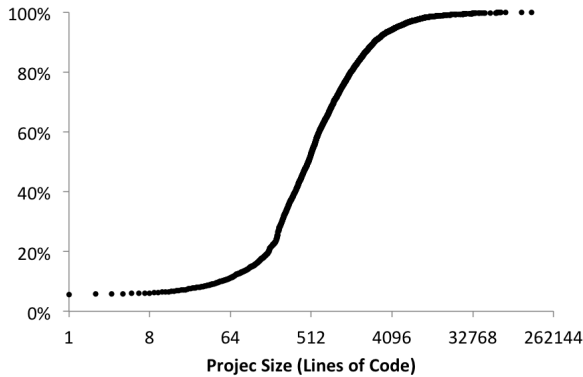
Figure 1. Cumulative Distribution of the Size of the Projects

The projects in our dataset were developed by 4481 different people. Table 1 shows that most projects were developed by a single person. A total of 96% of the projects were developed by small groups of 3 people or less. However, there were projects with up to 58 people.

These developes of the 7268 projects have different backgrounds. Figure 2 shows what are the languages used by these developers in other GitHub projects. Java is the most popular among

them. More than 50% of the developers of projects in our dataset also have Java projects hosted on GitHub.

Figure 3 shows what is the type system of the languages the developers in our dataset have experience with. Most of them have experience with both statically and dynamically typed languages. There are two small groups however that have experience with only one type system outside Groovy.

Table 1. Distribution of the Number of Developers in a Project

Number of Developers	Fraction of Projects
1	84%
2	9%
3	3%
4 or more	4%
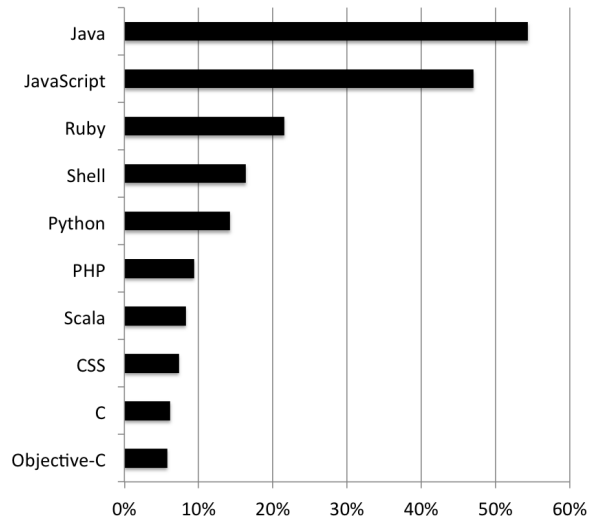
Figure 2. Usage of other languages by Groovy developers on GitHub

	Static	Dynamic	Static and Dynamic	No other languages
Developers	13.55%	14.73%	46.84%	24.88%
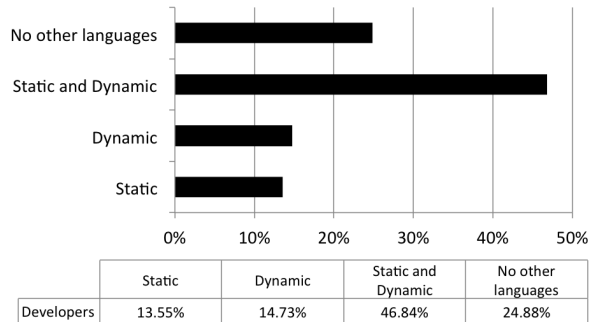
Figure 3. Type System of other languages used by programmers

3.4 Analysis

In order to understand where programmers use types, a static code analyzer was built. This analyzer is based on the Groovy metaprogramming library, which is capable of compiling Groovy files and generate an abstract syntax tree (AST) for them. The following types of declarations can be extracted

- Method Returns
- Method Parameters
- Constructor Parameters
- Fields
- Local Variables

In addition to the declarations extracted, we rely on the static code analyzer to answer the following questions:

- What is the typing system of the statement?
- Is the statement part of a script or a class?
- Is the statement part of a test class?
- What is the visibility of the method or field?

A relevant decision we made was no to compile projects. We made this decision because compiling the projects would require all dependencies to be resolved, which would be unfeasible. This would be even harder when we consider the dimension of our dataset. Instead, we generated the AST for each file using the $CONVERSION$ phase of the Groovy compiler. At this phase, the compiler hasn't tried to resolve any dependencies yet, but it is capable of generating an AST with sufficient information to determine if a statement had its type defined by the programmer 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 answer if programmers tend to type code that interacts with other typed modules since we haven't resolved any dependencies to these modules. However, our choice was fundamental in order to execute a study with such an expressive dataset. Nevertheless, as shown in the next section, we were still able to obtain detailed and relevant results using this strategy.

4. Results

Subsection **??** shows the type usage in different sorts of declarations and visibilities. Subsection 4.5 shows results depending on the type of the file where a declaration was made, a script, a class or a test class. The effects of the programmers background and the project size are described in the Subsection 4.6 and 4.7. Finally Subsection 4.8 presents results according to the number of changes of files. These results will be further discussed in Section 5 in order to answer the research questions proposed in Section 3.1.

4.1 Overall Result

Figure 4 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 doesn't 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.
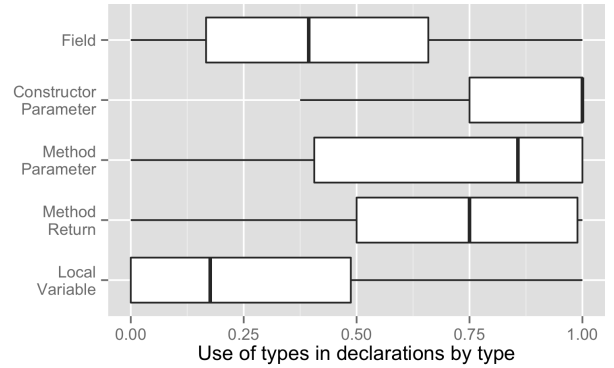
4.2 Declaration Type

The relative use of types per type of declaration is shown in Figure 4.2 along with the corresponding descriptive statistics. Each box plot in this graph describes relative use of types in five sorts of declarations: fields, parameters of constructors, parameters and returns of methods and local variables. It considers all declarations of all projects. Note that the value of n is different for each element in the table 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 declaration. Local
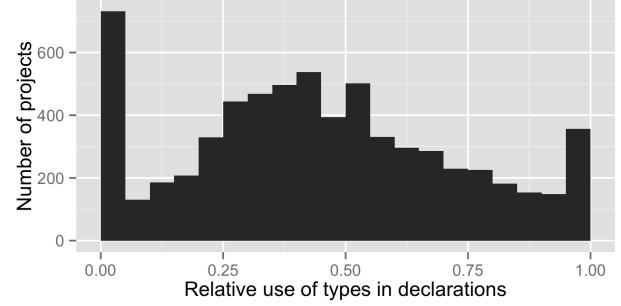
n	mean	std. dev.	Quartiles			
			1st	2nd	3rd	4th
6638	0.45	0.28	0.25	0.42	0.64	1.00

Figure 4. Usage of types in all declarations of all projects

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 5. Usage of types in all declarations by type of declaration

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.

In order to check whether there is a significant difference in the use of types in two kinds of declaration, a significance test should be applied. We start by defining a hypothesis below. A significance test can be applied to reject or accept the null hypothesis, H0, for any two declaration types.

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

We can't assume that the data presented in Figure 4.2 follows any particular distribution. In fact, an analysis of the data for declarations of local variables, methods and constructors shows that

these samples are heavily skewed and thus don't follow a Normal distribution. Because of this, we are required to use a non-parametric test to analyze this data.

An appropriate test for our scenario is the paired Mann-Whitney U test. This test computes a p value indicating whether two samples are significantly different form 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 addition to p, the Mann-Whitney U test also reports a confidence interval describing the difference of the medians of two populations. This confidence interval can be used to measure how different are two samples.

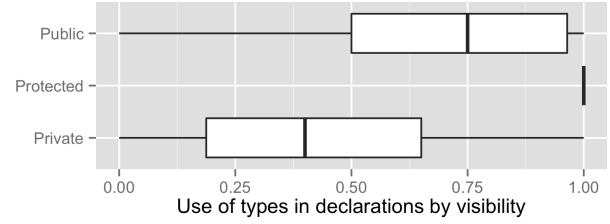|		p	conf. interval
Field	Constructor Parameter	0.000	(-0.49, -0.45)
	Method Parameter	0.000	(-0.32, -0.28)
	Method Return	0.000	(-0.30, -0.27)
	Local Variable	0.000	(0.13, 0.16)
Constructor Parameter	Field	0.000	(0.45, 0.49)
	Method Parameter	0.000	(0.07, 0.09)
	Method Return	0.000	(0.09, 0.12)
	Local Variable	0.000	(0.62, 0.67)
Method Parameter	Field	0.000	(0.28, 0.32)
	Constructor Parameter	0.000	(-0.09, -0.07)
	Method Return	0.005	(0.00, 0.00)
	Local Variable	0.000	(0.45, 0.49)
Method Return	Field	0.000	(0.27, 0.30)
	Constructor Parameter	0.000	(-0.12, -0.0)
	Method Parameter	0.005	(0.00, 0.00)
	Local Variable	0.000	(0.44, 0.47)
Local Variable	Field	0.000	(-0.16, -0.13)
	Constructor Parameter	0.000	(-0.67, -0.62)
	Method Parame	0.000	(-0.49, -0.45)
	Method Return	0.000	(-0.47, -0.44)

Table 2. Mann-Whitney U test results for the usage of types in declarations of all projects by visibility

Table 4.2 shows the result of the Mann-Whitney U test for the data displayed in Figure 4.2. Confidence intervals are calculated with a confidence of 95%. All p values are close to 0, allowing us to strongly reject the null hypothesis for all test executions. Note however that the confidence interval for the comparison between method parameters is negligible.

4.3 Declaration Visibility

Figure 4.3 shows how programmers use types depending on the visibility of a declarations. Results consider all sorts of declaration, except for local variables which don't have visibility. We follow the same approach from previous section and apply the Mann-Whitney U test to determine whether there is a significant difference between declarations of different visibilities. Results for the tests executions are displayed in Table 4.3. They show that the p value for all tests is equal to 0, which points out that all declarations visibilities are different from each other.

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 type all of their protected fields, methods and constructors. The confidence intervals reported by the U tests show a large difference

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 6. Usage of types in all declarations by visibility

		p	conf. interval
Public	Protected	0.000	(-0.21,-0.18)
	Private	0.000	(0.28,0.31)
Protected	Public	0.000	(0.18,0.21)
	Private	0.000	(0.54,0.57)
Private	Public	0.000	(-0.31,-0.28)
	Protected	0.000	(-0.57,-0.54)

Table 3. Mann-Whitney U test results for the usage of types in declarations of all projects by visibility

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.

4.4 Test Classes and Main Classes

This sections investigates whether programmers type their declarations differently when they are writing test code. A comparison between the usage of types in different declarations is shown in Figure 4.7. We also apply the Mann Whitney U test, but this time however we are not comparing declaration types to each other. Instead, we are comparing declarations from main classes and test classes. Results for the U test are shown in Table 4.7.

There is no significant difference in the usage of types in fields and constructor parameters across test classes and main classes. The p value reported by the U test comparing constructor parameters data is too high, 0.397, which doesn't allow us to reject the null hypothesis for this type of declaration. As for fields, p value for the test comparing fields is small enough, 0.002, but the reported confidence interval negligible.

Local variables and method parameters are significantly less typed in test classes. The confidence intervals reported in Table 4.7 show a difference of (-0.500, -0.389) and (-0.058, -0.030) respectively, both with p-value equal to 0. Method returns on the other side are significantly more typed. Figure 4.7 shows a median equal to 1.00, which means that at least half of the projects type all of their method returns. 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 out of the 5299 projects with test classes, requires the type of the return of a test methods to be *void*. Other popular test frameworks, such as TestNG, have similar requirements. This means that, in this case, developers type their methods not because they want to, but because they need to.

Figure 7. Usage of types in all declarations by visibility of declaration
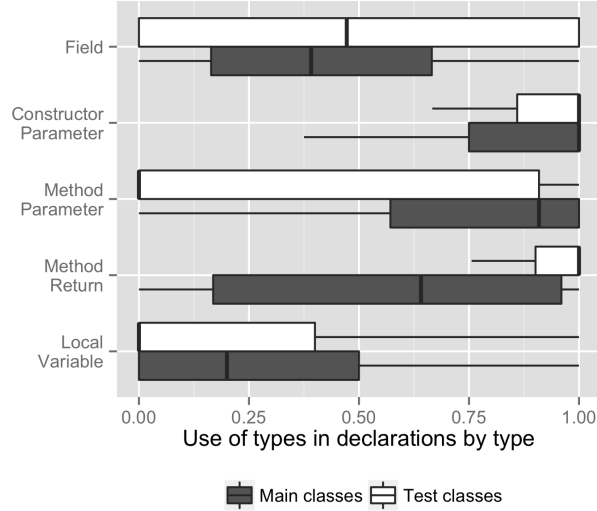
Declaration Type	Class Type	n	mean	median	std. dev.
Field	Test	1778	0.48	0.47	0.43
	Main	5857	0.43	0.39	0.33
Constructor Parameter	Test	130	0.77	1.00	0.40
	Main	1623	0.80	1.00	0.34
Method Parameter	Test	1546	0.34	0.00	0.44
	Main	4593	0.71	0.91	0.35
Method Return	Test	4348	0.84	1.00	0.31
	Main	5299	0.54	0.60	0.39
Local Variable	Test	2896	0.23	0.00	0.35
	Main	5548	0.30	0.19	0.32

Figure 7. Usage of types in all declarations by visibility of declaration

Declaration Type	p	conf. interval
Field	0.002	(0.000, 0.000)
Constructor Parameter	0.397	(0.000, 0.000)
Method Parameter	0.000	(-0.500, -0.389)
Method Return	0.000	(0.231, 0.250)
Local Variable	0.000	(-0.058, -0.030)

Table 4. Mann-Whitney U test for tests

4.5 Script Files and Class Files

We now analyze how programmers type their code in script files. Figure 8 shows the relative usage of types in declarations of local variables and parameters and returns of methods. Constructors and fields are omitted since there is no way to declare those elements in scripts. Table 4.5 shows the results of the Mann-Whitney U test for these elements.

The p value for all executions of the U test is equal to 0, allowing to reject the null hypothesis. The reported confidence interval for local variables however is too small and it can be argued that the difference in how programmers type these variables in scripts is insignificant. On the other side, parameters and returns of methods present a very different behavior. Different from classes, most of projects don't type these declarations very often in scripts. This is even more clear in method returns, where most of projects don't

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 8. Usage of types in all declarations by visibility of declaration

type any of these declarations. Note however that the value for the last quartile of these elements is very high, superior to 0.8. This indicates that there are a few projects that consistently type most of these declarations too.
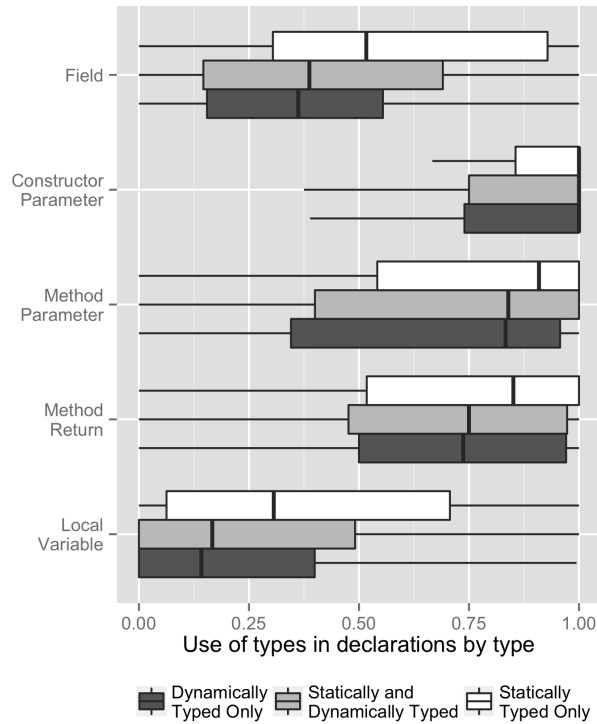
Declaration Type	p	conf. interval
Method Parameter	0.000	(-0.365, -0.250)
Method Return	0.000	(-0.500, -0.414)
Local Variable	0.000	(-0.016, 0.000)

Table 5. Mann-Whitney U test of scripts

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 contains 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, are included in the second group. Finally, the third group includes the project of those programmers with both dynamically and statically typed languages in their portfolio. Figures 4.6 and 4.7 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 for the three different background groups.

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 confidence interval is arguably small, between 0 and 0.051

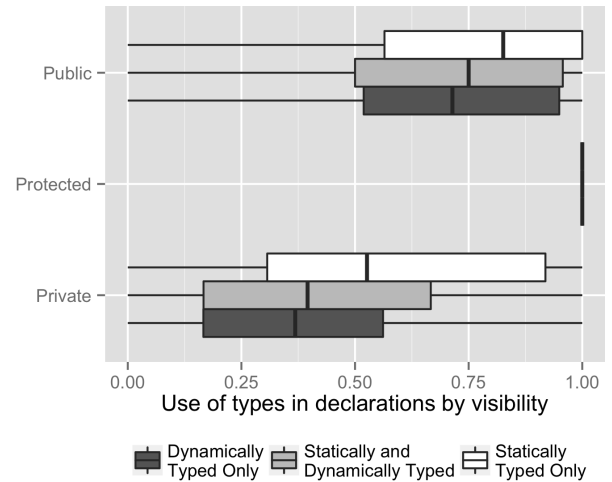| Declaration Type	Background	n	mean	median	std. dev.
Field	Static	782	0.56	0.52	0.35
	Both	3183	0.43	0.39	0.34
	Dynamic	2035	0.38	0.36	0.29
Constructor Parameter	Static	224	0.83	1.00	0.33
	Both	991	0.80	1.00	0.35
	Dynamic	455	0.80	1.00	0.34
Method Parameter	Static	662	0.73	0.91	0.34
	Both	2694	0.67	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 9. Usage of types in projects by declaration type depending on the background of the programmers of a projects

in both cases. Conversely, programmers with only statically typed languages in their portfolio tend to use types more often than others. There are only two situations where the p value reported by the Mann-Whitney U test is big enough so the null hypothesis can't be rejected, protected declarations and declarations of constructor parameters.

4.7 Project Size and Maturity

This section presents an analysis about how programmers use types in their code depending on the project size. We start by calculating the Spearman Rank Correlation Coefficient between the number of lines of code and the usage of types. This coefficient, which can go from -1 to 1, measures the correlation between two variables. A positive value means that, as the size of projects grows, so does

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 10. Usage of types in projects by declaration visibility depending on the background of the programmers of a projects

	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 6. Mann-Whitney U Test results for declaration types depending on the background of the programmers of a projects

the usage of types. A negative value means the opposite. Values close to 1 or -1 denote the strongest relationship while values above 0.5 or below -0.5 can be considered strong correlations. Table 4.7 shows the Spearman correlation for declarations grouped by type

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

Table 7. Mann-Whitney U Test results for declaration visibility depending on the background of the programmers of a projects

and visibility. There are no correlations in this table that seem to suggest a strong correlation.

We start by running an analysis In order for a project to be considered mature it has to have 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 two metrics. According to our criteria, there are 203 mature projects in our dataset. Together they a total of 83211 commits. The most changed project had a total of 6545 commits. More than 43 thousand files were changed by these commits.

| Declaration Type/Visibility	LoC	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 8. Spearman Correlation between the usage of types and the size, age and number of commits of projects

Declaration Type	p	conf. interval
Field	0.0000	(0.058,0.169)
Constructor Parameter	0.4889	(0.000,0.000)
Method Parameter	0.1041	(0.043,0.005)
Method Return	0.5880	(0.036,0.039)
Local Variable	0.0001	(0.015,0.077)

Table 9. Mann-Whitney U Test results comparing mature projects to others by declaration type

Declaration Visibility	p	conf. interval
Public	0.6330	(0.034, 0.036)
Private	0.0000	(0.000, 0.000)
Protected	0.0000	(0.060,0.167)

Table 10. Mann-Whitney U Test results comparing mature projects to others by declaration visibility
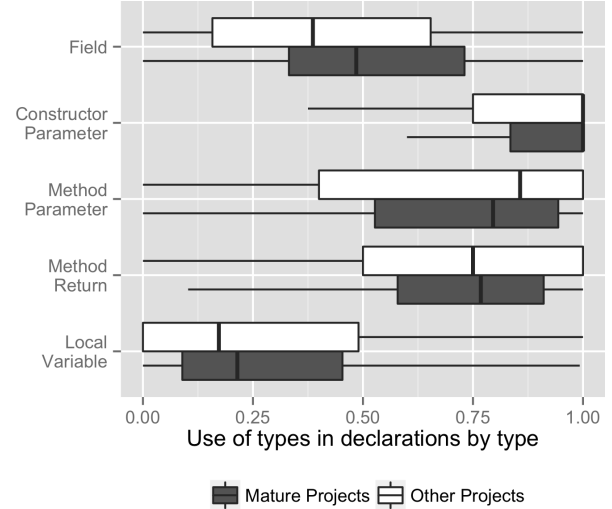
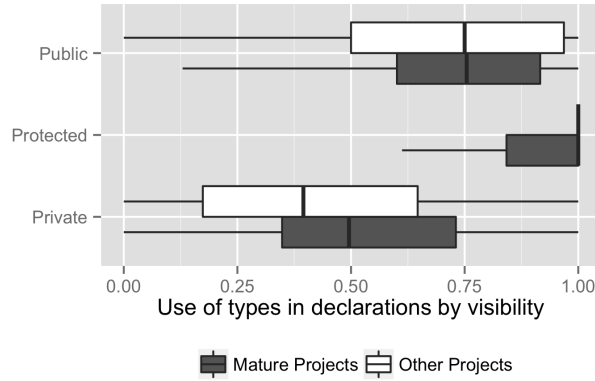| Declaration Type	Project Type	n	mean	median	std. dev.
Field	Other	5749	0.43	0.39	0.33
	Mature	251	0.53	0.48	0.27
Constructor Parameter	Other	1484	0.80	1.00	0.35
	Mature	186	0.84	1.00	0.29
Method Parameter	Other	4616	0.67	0.86	0.37
	Mature	251	0.69	0.80	0.30
Method Return	Other	5629	0.68	0.75	0.32
	Mature	252	0.71	0.77	0.25
Local Variable	Other	5592	0.29	0.17	0.32
	Mature	253	0.32	0.21	0.28

Figure 11. Usage of types in all declarations by visibility of declaration

4.8 Frequency of changes

The results in this section show how the frequency of changes in a file is related to the usage of types in the declarations in those files. For this analysis, we considered only mature projects since we would not be able to get significant results from small and young projects.

In order for a project to be considered mature it has to have 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 two metrics. According to our criteria, there are 203 mature projects in our dataset. Together they a total of 83211 commits. The most changed project had a total of 6545 commits. More than 43 thousand files were changed by these commits.

Figure **??** shows the average frequency of untyped declarations in files by the frequency of changes of those files. A file is considered in different groups depending on the relative number of commits that change this file in each project. Each group contains 10% of the files in a project such that group 10 contains the top 10% of files with the most changes of every project, while group 1 has the bottom 10% of the files with the least. This means that the most changed files of two different projects, one file changed in 50 commits and the other changed in 800 commits, are both part of group 10. We avoided using the absolute number of changes in a file since this value might have different meanings in different projects. What might be considered a high number of commits for

Figure 12. Usage of types in projects by declaration visibility depending on the background of the programmers of a projects

Declaration Type	Project Type	n	mean	median	std. dev.
Public	Other	5599	0.69	0.75	0.29
	Mature	253	0.71	0.75	0.24
Protected	Other	2186	0.94	1.00	0.19
	Mature	201	0.88	1.00	0.21
Private	Other	5772	0.43	0.40	0.32
	Mature	251	0.53	0.50	0.26
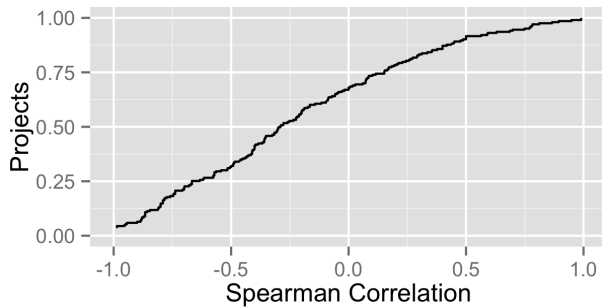
Figure 13. Spearman ranking for the relation between untyped declarations and frequency of changes for mature projects

one project, may be low for another. By using the relative number of changes, we avoid this pitfall.

It is possible to observe in Figure **??** a slight correspondence between the frequency of changes and the use of untyped declarations. While the most changed files of each project have, on average, 34.7% of their declarations untyped, this number is only 24% for the least changed files.

Figure 4.8 is capable of describing this relationship in more details. Each point in that graph represents the Spearman correlation between the number of changes in a file and the usage of untyped declarations for a project. A positive value denotes a correlation between those two metrics while a negative value denotes an inverse relationship, i.e, the higher the number of changes the less untyped declarations are used. Values close to 1 or -1 the strongest the relationship while values above 0.5 or below -0.5 can be considered strong correlations.

From the 203 mature projects in our dataset, only 65 present negative values for the Spearman ranking. Most of the projects present a positive correlation between the frequency of changes and the usage of untyped declarations. A total of 64 of these present a

strong relationship, i.e, a Spearman ranking value superior to 0.5, while only 20 present a strong inverse relationship.

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 statements and test classes in Section 5.2 supports the suspect that untyped declarations are used more frequently when readability or stability aren't 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 on 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 [29]. 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't 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 [21–26], 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 an well typed module learn how to use it faster and make less mistakes since the contract of that module is clear to them.

Statements 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 side, parameters of constructors, which are some of the most important elements of a contract definition, are almost always typed [21].

The results presented above above are further refined by visibility in Figures **??**, **??**, **??** and **??**. These results contribute to the understanding that types are indeed considered more often on the definition of modules. First, notice that private statements are always less typed than their public and protected counterparts. Private statements 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 statements with protected visibility surpasses that of statements with public visibility. In Groovy, protected declarations restrict access to internal elements of a class to its subclasses or other classes in the same pack-

age. It can be argued that the relationship between these classes is tightly coupled [28]. 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 statements. Unlike private or protected statements, the number of modules to which a public statement 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 aren't a concern

When readability or stability are not a concern, the advantages of typing become less apparent and programmers prefer the flexibility and objectivity of untyped declarations. This is what we found when we analyzed how typing is used in local variables, private statements and test classes. These are scenarios 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 and life cycle. 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 scenarios is not as critical making the flexibility of untyped declarations more apparent. 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 private statements. As shown in Figures **??**, **??**, **??** and **??**, untyped declarations are much more common in in these statements that in their public or private counterparts. Private statements 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 statements 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 keeping the quality of such code [27].

Test classes are also typed less often than the usual. Figure **??** shows that declarations made inside the main classes of a project are typed in 68% of the cases. On the other side, 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 aren't 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 side, 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 argument 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 statements untyped less often than other developers. Figure **??** shows that these developers have only 28.7% of their statements untyped compared to 40% of those developers with dynamic languages on their portfolio.

It is clear that the portfolio of a GitHub developer might not represent well his real experience with other programming languages. This developer might have projects hosted somewhere else or have projects in his portfolio that he has barely participated at. However, due to the scale of this study, it is difficult to collect more detailed metrics for every user.

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 [31].

Internal Validity

The main threats to the internal validity involve the fact that, in such a large scale empirical study, we can't analyze data into 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. However, these programmers might have projects hosted somewhere else or projects in his portfolio that he hasn't worked on.

In Section 4.7, we consider that the project size is a metric capable of representing the complexity of a project, but this is not always true [30]. However that was the most effective metric we could use without requiring projects to be compiled, which would drastically reduce the number of projects in our dataset.

There are other factors which might have influenced the 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 mock on 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 expressive 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, 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 different 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 typing 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 of 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 got to 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 aren't 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 weren't 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. It's also a valuable information 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 more that 7 thousand 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 statement visibility, the complexity of the contract defined by such statement and the size of the project where it was defined. Conversely, when readability or stability aren't a concern, the simplicity and flexibility of untyped declarations seems to be preferred, as seen on local variables, private statements and test classes. Another important factor is the previous experience of programmers with a given type system.

In future works 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

References

[1] Tiobe programming community index. http://www.tiobe.com/index.php/ content/paperinfo/tp-ci/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] Meijer, Erik, and Peter Drayton. "Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages." OOPSLA, 2004.

[22] Wadler, Philip, and Robert Bruce Findler. "Well-typed programs cant be blamed." Programming Languages and Systems. Springer Berlin Heidelberg, 2009. 1-16.

[23] 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.

[24] Flanagan, Cormac. "Hybrid type checking." ACM Sigplan Notices. Vol. 41. No. 1. ACM, 2006.

[25] Meyer, Bertrand. Object-oriented software construction. Vol. 2. New York: Prentice hall, 1988.

[26] Furr, Michael, et al. "Static type inference for Ruby." Proceedings of the 2009 ACM symposium on Applied Computing. ACM, 2009.

[27] Meszaros, Gerard. xUnit test patterns: Refactoring test code. Pearson Education, 2007.

[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.