How Programmers Use Optional Typing? A Case Study

Carlos Souza
Federal University of Minas Gerais
carlosgsouza@gmail.com

Eduardo Figueiredo
Federal University of Minas Gerais
figureido@dcc.ufmg.br

Marco Tulio Valente
Federal University of Minas Gerais
mtov@dcc.ufmg.br

Abstract

One of the most important things to be taken into account when choosing a programming language is its typing system, static or dynamic. This question has become increasingly more important due to the recent popularization of dynamic languages such as Ruby and JavaScript. This paper studies which are the most influencing factors for a programmer when choosing between typing systems. An analysis of the source code of over seven thousand projects written in Groovy, a programming language where one can mix static and dynamic typing freely, shows in which situations programmers prefer a typing system over the other. Results of this study suggest that the previous experience of the programmer, project size, complexity of modules, scope and visibility of statements are some of the most important factors in this decision.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

When choosing a programming language for a project, a developer considers several characteristics of that language. One of the most important is its typing system, which can be static or dynamic. The typing system determines when the type of a statement is be defined[11]. 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 during run time.

The discussion about what is the best typing system has 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 in the industry are dynamically typed. In 2001, this number was only 17%. Among the 10 languages on top of the ranking, four are dynamically typed: JavaScript, Perl, Python and PHP. In 1998, none of these languages was among the top 10 ranking.

Several factors may be considered when choosing between a dynamcally or statically typed language. Since dynamically typed languages are simpler, they allow programmers to perform their tasks more quickly [11] and adapt to changing requirements [13]. Also, by removing the repetitive work of defining types, these languages allow their users to focus on the problem to be solved rather than spending time with the rules of the language [12].

Statically type languages also have their advantages. These allow compilers to look for type errors [9]. Typed declarations increase the maintainability of systems because they document the code, informing the programmer the nature of each variable [4, 10]. Systems built with these languages tend to be more efficient since they do not need to perform type checking during their execution [2, 5]. Finally, modern development environments, such as Eclipse and IDEA, are able to assist programmers with functionalities such as code completion based the information they get from statically typed declarations [3].

This paper presents a study aiming to understand which factors actually influence the decision of a developer for typing their declarations or not. This question was studied based on the analysis of a massive dataset with more than seven thousand projects. These projects were written in Groovy, a language with a hybrid typing system which allows developers to choose, for each statement, either to type or not. Through a static analysis of these projects, it was possible to understand when developers choose each type system and then extract what are the factors that influence this decision.

The remainder of this paper is organized as follows. Section 2 introduces the main concepts of the Groovy program language. Sections 3 and **??** describe the configuration of the study and its results. Section 6 discusses threatens to validity and section 7 shows related work. Finally, section 8 concludes this study and suggests future works.

2. The Groovy Language

Groovy is an object-oriented programming language designed to run on the Java Virtual Machine. 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. Groovy type system is flexible and allows programmers to optionally type their statements. This makes Groovy an ideal language for a case study about how programmers choose between using or not typing in their code.

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 22st 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 syntax 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. Below we show two implementations of a simple algorithm. 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.

Since Groovy has more expressiveness, it's able to reduce much of the boilerplate required in Java. Consider Listing 2. Note 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).

Listing 1 A simple algorithm written in Java

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class JavaFilter {
5    List<Integer> evenNumbers(List<Integer>
         list) {
6      List<Integer> result = new ArrayList<
           Integer>();
7      for(int item : list) {
8        if(item % 2 == 0) {
9          result.add(item);
10       }
11     }
12
13     return result;
14   }
15
16   public static void main(String[] args) {
17     List<Integer> list = new ArrayList<
           Integer>();
18     list.add(1);
19     list.add(2);
20     list.add(3);
21     list.add(4);
22
23     List<Integer> result = new JavaFilter().
           evenNumbers(list);
24     System.out.println(result);
25   }
26 }
```

Groovy's design 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 will 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

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 }
```

features of Groovy compared to Java. They allow a functional programming style of code, which is both expressive and powerful.

Listing 3 A class written in Groovy

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

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 [19].

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())
```

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 will 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 will reference an instance of the $Integer$ class. However, an exception will be thrown when we try to invoke the method $toUpperCase$ since the $Integer$ class doesn't 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 [14–18]. In this mode, the Groovy compiler will look for type errors and fail if it finds any.

Listing 7 shows an example of static typing in Groovy. Trying to compile the class $TypeCheckedGroovyClass$ will produce an

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

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

error since the method sum is supposed to receive two parameters with the type int, but is called with two parameters of the type $String$.

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 }
```

The $@TypeChecked$ annotation is reasonably recent and most Groovy programmers still don't use it. Typing annotations on the other side are very popular. Although they don't provide static type checking, they are capable of documenting the code and aiding in the integration with development tools.

3. Methodology

In this paper, we want to understand what are the factors that have an actual influence over the decision of a developer to type their declarations or not. In order to answer that question, we analyzed a series of Groovy projects. Groovy allows programmers to choose whether to type or not each of their statements. By finding out when programmers choose to use types or not, we can infer what is the factor that has influenced such choice.

A case study is the best approach to analyze this question since it provides us with actual data from programmers in their regular activities. In order to obtain reliable results, however, we need a large number of projects. Fortunately, it was possible to retrieve a massive dataset with more that 7 thousand projects from GitHub, a popular source control service. A static code analyzer was developed to analyze typing usage in these projects and these results were analyzed in conjunction with other data about these projects such as the project size and the programmers background.

In the remainder of this section, we formalize the research questions in subsection 3.1. The Data Collection procedure is described in subsection 3.2 while the dataset itself is discussed in subsection 3.3. Finally, we discuss the static code analyzer and the analysis procedure in subsection 3.4.

3.1 Research Questoins

- **Question Q1:** Do programmers use types to implicitly document their modules?
- **Question Q2:** Do programmers prefer not using types in simple code?
- **Question Q3:** Does the previous experience of programmers with other languages influence their choice for typing their statements? More specifically, do programmers with a background on statically typed languages tend to keep using types while those with a background on dynamically typed languages tend to keep their statements untyped?

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, and the metadata of all developers involved in that project. GitHub doesn't offer a listing of all hosted projects, but it offers two search mechanisms, a REST API and a web based search page. Unfortunately GitHub'sAPI is too limited for our requirements. It imposes a limit of one thousand results and doesn't allow filtering projects by their programming language.

In order to retrieve a expressive dataset, it was necessary to write a bot to interact with GitHub's webpage and search for projects. Some special care was necessary to make this strategy work. Because the number of results is limited to 1 thousand projects, we had to segment the queries. Multiple requests were made. Each request asked for the name of all projects created on a given month and results were combined. 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 GitHub's REST API to query for the metadata of all projects. Since the metadata of a project also contains the id of the developers of that project, it was possible to obtain the metadata of those developers using GitHub's REST API.

3.3 Dataset

Our dataset consists of 7268 obtained from GitHub. These projects add to 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. Projects have between 1 LoC and 150 KLoC. The age of a project, defined as the time between the the first and last commits, varies form 1 day to 5 years. The distribution of the size and age of the projects are shown in Figures 1 and 2 respectively.

The projects in our dataset were developed by 4481 prople. While most projects were developed by a single programmer (Table 1), there are projects with up to 58 developers. These developers have different backgrounds. Figure 3 shows what are the other languages used by these developers in other GitHub projects. Java

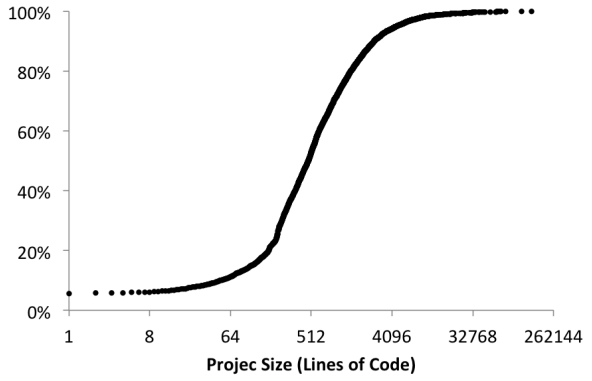


Figure 1. Cumulative Distribution of the Size of the Projects

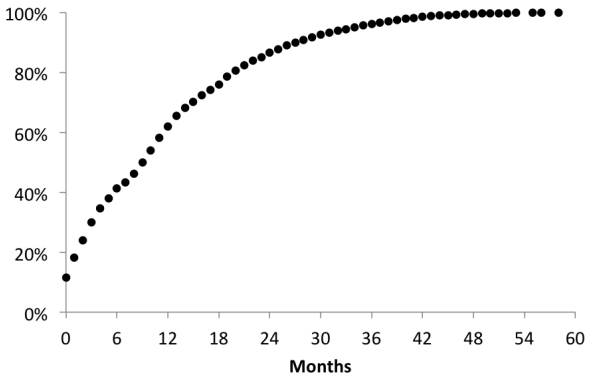


Figure 2. Cumulative Distribution of the Age of the Projects

is the most popular among them. More than 50% of the developers of projects in our dataset also have Java projects on GitHub. Figure 4 shows the background of these developers in terms of the type system of the other languages they have used.

Table 1. Distribution of the Number of Developers

Number of Developers	Fraction of Projects
1	84%
2	9%
3	3%
4 or more	4%

3.4 Analysis

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

- Return Methods
- Method Parameters
- Constructor Parameters
- Fields
- Local Variables

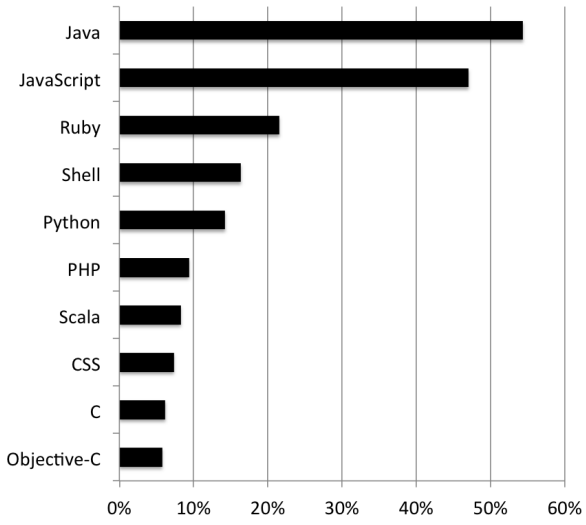
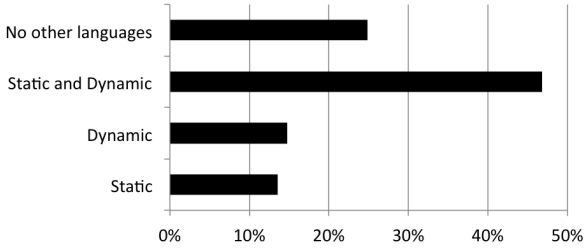


Figure 3. Usage of other languages by Groovy developers on GitHub



	Static	Dynamic	Static and Dynamic	No other languages
Developers	13.55%	14.73%	46.84%	24.88%

Figure 4. Type System of other languages used by programmers

In addition to that, it can answer the following questions for each statement

- 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 statement? (except for local variables)

We opted for not compiling projects. This would require us to resolve all dependencies, which many times is impossible. This is 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 the type system of each statement. 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't analyze Groovy code in conjunction with its dependencies. For example, it's 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, this was

fundamental in order to execute a study with such an expressive dataset. Nevertheless, as we will see in the next section, we were still able to obtain detailed and relevant results using this strategy.

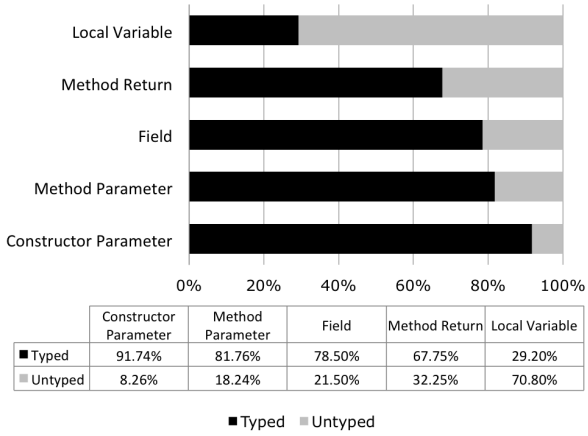
4. Results

In this section we present the results of the analysis of the declarations of the projects described in section **??**. These results will be discussed in more details in section 5.

The analysis shows that 61% of the statements are typed, while only 39% of these are dynamically typed. This results considers all declarations of the 7268 projects in the dataset.

4.1 Declaration Type

Figure 5 shows the amount of typed and untyped declarations for each type of declaration. Untyped declarations are far more frequent in local variables that in other types of statements. Fields, method returns and parameters on the other side are mostly typed. In particular, constructor parameters are typed in more than 90% of the declarations.



	Constructor Parameter	Method Parameter	Field	Method Return	Local Variable
Typed	91.74%	81.76%	78.50%	67.75%	29.20%
Untyped	8.26%	18.24%	21.50%	32.25%	70.80%

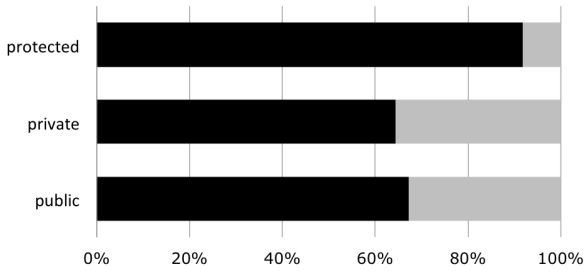
Figure 5. Type Systems by Declaration Type.

4.2 Visibility

In this section, we break down the results shown above by visibility. Figure 6 shows results for method returns. There is little difference in the usage of types on the declarations of the return of private and public methods, with 64.44% and 67.26% declarations typed respectively. On the other side, the return of protected methods, are almost always typed.

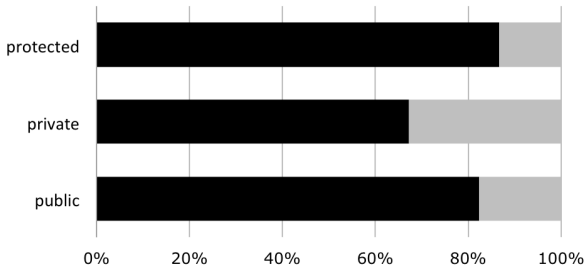
Figure 7 shows the results of the analysis of method parameter declarations. Compared to the results for method returns shown in figure 6, declarations of public method parameters are typed more often. While public method returns have types in 67.26% of the times, public method parameters have more than 80% their declarations typed. The usage Parameters of constructors, shown in Figure 8, present a similar behavior to that of parameters of methods.

Figure 9 shows the results for field declarations. Differently from previous analyzed declarations, protected fields are significantly less typed than public fields. While the first is typed in 88.8% of the cases, 80.5% of the latter are typed. Another difference is that private fields are slightly more typed than other private declarations. While this frequency is 72.4% for private fields, method returns, method parameters and constructor parameters don't get typed more than 67% of the time.



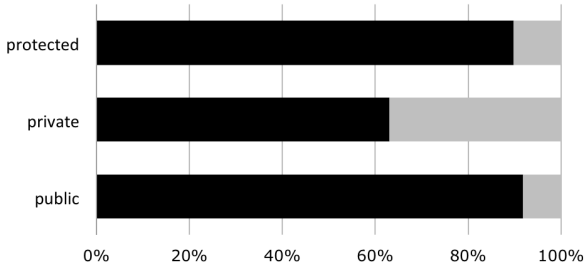
	public	private	protected
Typed	67.26%	64.44%	91.75%
Untyped	32.74%	35.56%	8.25%

Figure 6. Type Systems by Declaration Visibility



	public	private	protected
Typed	82.38%	67.17%	86.63%
Untyped	17.62%	32.83%	13.37%

Figure 7. Type Systems by Declaration Visibility



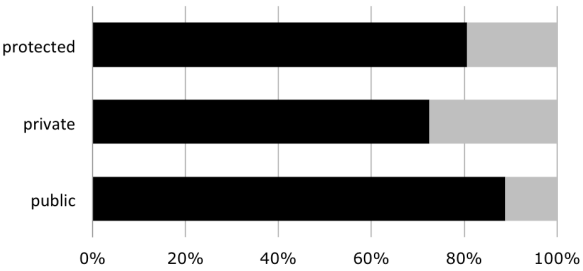
	public	private	protected
Typed	91.80%	63.01%	89.83%
Untyped	8.20%	36.99%	10.17%

Figure 8. Type Systems by Declaration Visibility

In summary, for all types of declarations, private declarations are less typed than the public and protected counterparts. Protected declarations, on the other side, are usually the most typed. The only exception to this are fields, where the usage of types in public declarations surpasses those of protected declarations by 8%.

4.3 Project Size

In this section we show how the use of types in declarations varies according to the project size. Project size is a simple metric to measure the complexity of a project. Usually, the larger the project, the greater the number of modules and and the need for maintenance. There are metrics far more effective for this purpose such as



	public	private	protected
Typed	88.77%	72.44%	80.54%
Untyped	11.23%	27.56%	19.46%

Figure 9. Type Systems by Declaration Visibility

[TODO]. However, this simple approach is ideal for this case study because it doesn't require projects to be compiled to determine the project complexity. As discussed in section 3.3, this would be impracticable considering the size of the studied dataset.

Results are shown for public, private and protected declarations (Figures 10, 11, 12 respectively) and local variables (Figure 13). In these graphs, each bar shows the average usage of untyped declarations of projects grouped by their size. The boundaries of each group are defined under each bar. For example, a project with 1500 lines of code is in the fifth group, since 1500 lies within the interval $]1024, 2048]$. Notice that we are using a logarithm scale for the project size in these graphs.

Types are less used in public declarations as project size increases. While projects with less than 1024 lines of code have almost 40% of their declarations untyped, this number is less than 20% in projects with over 65536 lines of code. Figures 11 and 12 show that, different from public declarations, private and protected declarations have no apparent correlation with the project size.

Conversely, the use of untyped declarations in local variables increases with the project size. This is shown in figure 13. The average frequency of untyped local variable declarations in projects with less than 512 lines of code is less than 40%. On the other side, in projects with over 65536 lines of code, the average frequency for this type of declaration exceeds 70%.

4.4 Test Classes

In this section, we analyze how types are used in test classes. Table 2 shows that declarations in this type of code are 14% less typed when compared to declarations in other types of classes.

Table 2. Type System used in different contexts

	Typed Declarations	Untyped Declarations
Test Classes	56%	44%
Other Classes	68%	32%

4.5 Scripts

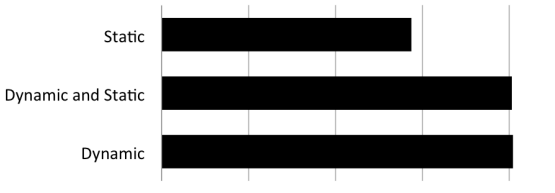
In this section we analyze how types are used in script files. Differently form last section, Table 3 shows that there is no significant difference between declarations in this type of code and declarations in class files.

Table 3. Type System used in different contexts

	Typed Declarations	Untyped Declarations
Scripts	63%	37%
Classes	60%	40%

4.6 Programmers Background

Figure 14 shows how programmers use types in their statements according to the type system of the languages they have used on GitHub. Developers are distributed in three groups, those with only statically typed languages, those with only dynamically dynamic languages and those with both. There is a clear distinction in the behavior of those whose all projects are written in statically typed languages. These programmers use types in about 28% of their statements compared to 40% of the other programmers. Notice that the behavior of those programmers with dynamic languages only on their portfolio is no different from those with statically and dynamically typed languages.



	Dynamic	Dynamic and Static	Static
Untyped Statements	40.42%	40.29%	28.73%

Figure 14. Programmers Background and Usage of Untyped Declarations

5. Discussion

5.1 Types as Documentation

In this section we discuss Q1, i.e, do programmers consider typing as a means to document their code? Well documented code plays an important role in making code more readable, hence improving the code overall maintainability [27]. There is evidence that Groovy programmers consider types as a mechanism to implicitly document their code. This is even more clear the definition of the interface of their modules. Given the results presented in section 4, we want to understand if there is a higher usage of typing in situations where readability is an important factor.

Documentation is the main reason why Groovy programmers type their code. As mentioned earlier in this study, the main advantadges of using statically typed languages are implicit code documentation, static type verification and execution performance. 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 runtime. Hence from the usual advantadges of using a statically typed language, only code documentation holds true for Groovy. Considering the result shown on previous section that 60% of all declarations are typed, this means that code documentation is an important matter for Groovy developers.

5.2 Types for modularization

Typing plays an even more important role on the documentation of modules. As mentioned in the literature [20–25], types greatly aid in the definition of the contract of a module. They retrict the pre and post conditions of methods and define the nature of the properties of that module. Clients of an well typed module will learn how to use it faster and make less mistakes since the contract of that module is clear to them.

In our analysis we were able to indentify an even higher usage of typing in those statements that define the interface of a module,

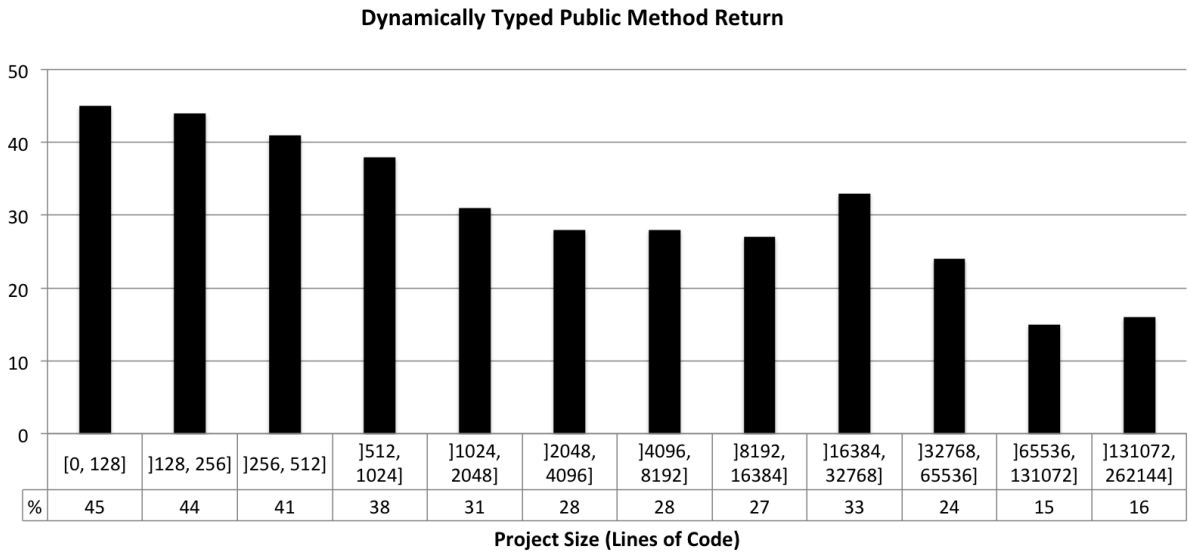


Figure 10. Dynamic Typing usage in local variables

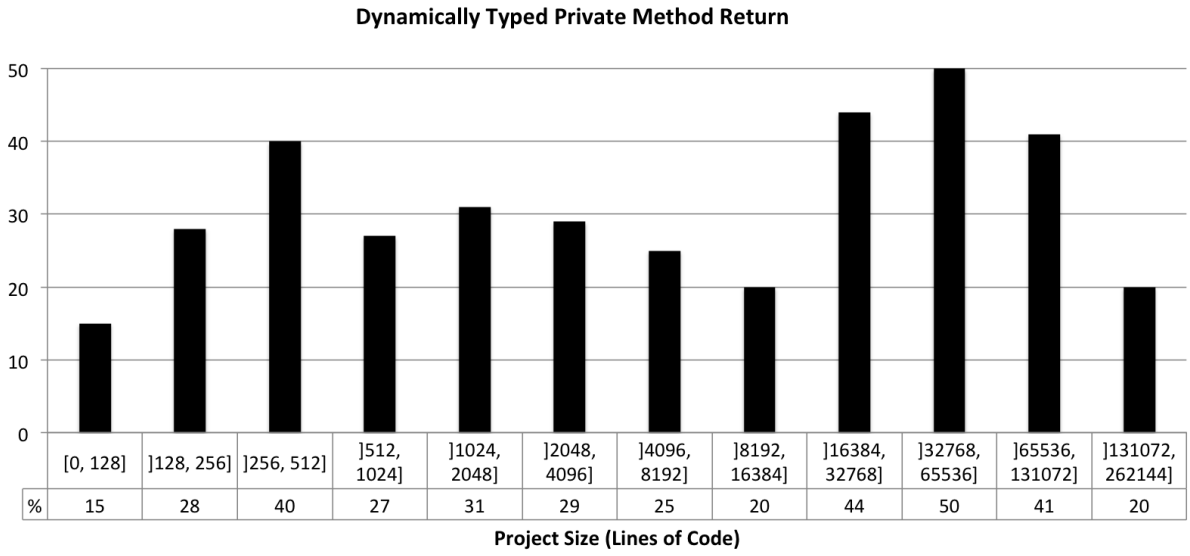


Figure 11. Dynamic Typing usage in local variables

i.e, fields, returns and parameters of methods and parameters of constructors. Consider Figure 5. While these elements are typed in the great majority of times, only 30% of the local variables, which don't 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

The results grouped by visibility, shown on Section **??**, help to confirm that types are indeed considered more often on the definition of modules. Figure 6, 7, 8 and 9 show that private declarations are always less typed than the public and protected counterparts. Similar to local variables, private statements don't 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 don't feel as compelled to use types.

Public and protected elements define the interface of a method and thus are widely typed. 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. Groovy developers use $protected$ declarations as a mechanism to restrict access of internal elements of a class allowing only subclassess to access them. It can be argued that this is a tighly coupled relationship between these classes [26] since they expose internal elements of a superclass to a subclass. Apparently programmers understand that

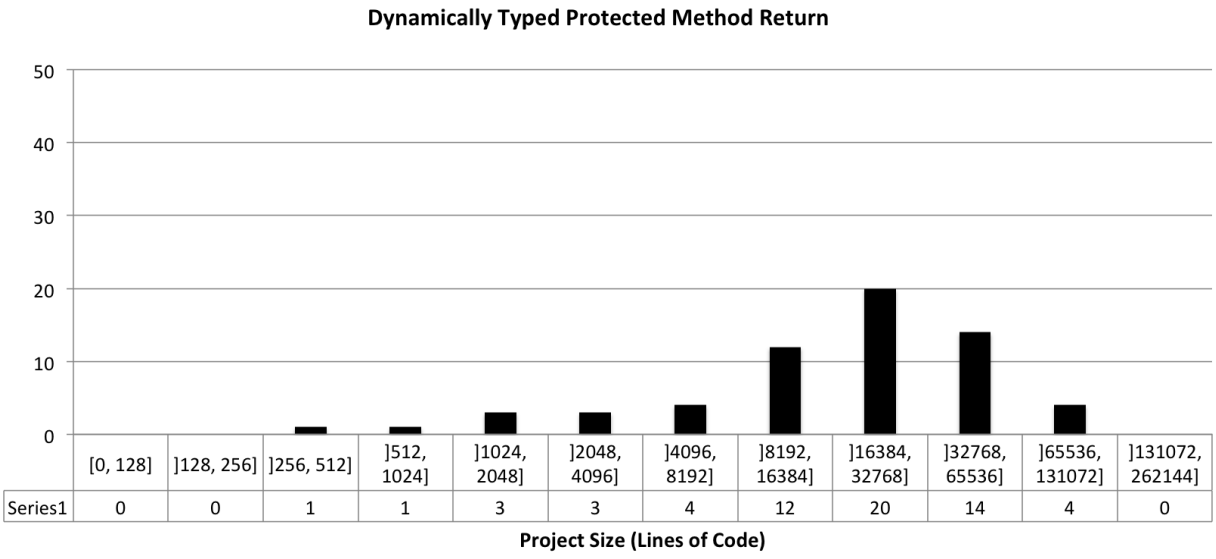


Figure 12. Dynamic Typing usage in local variables

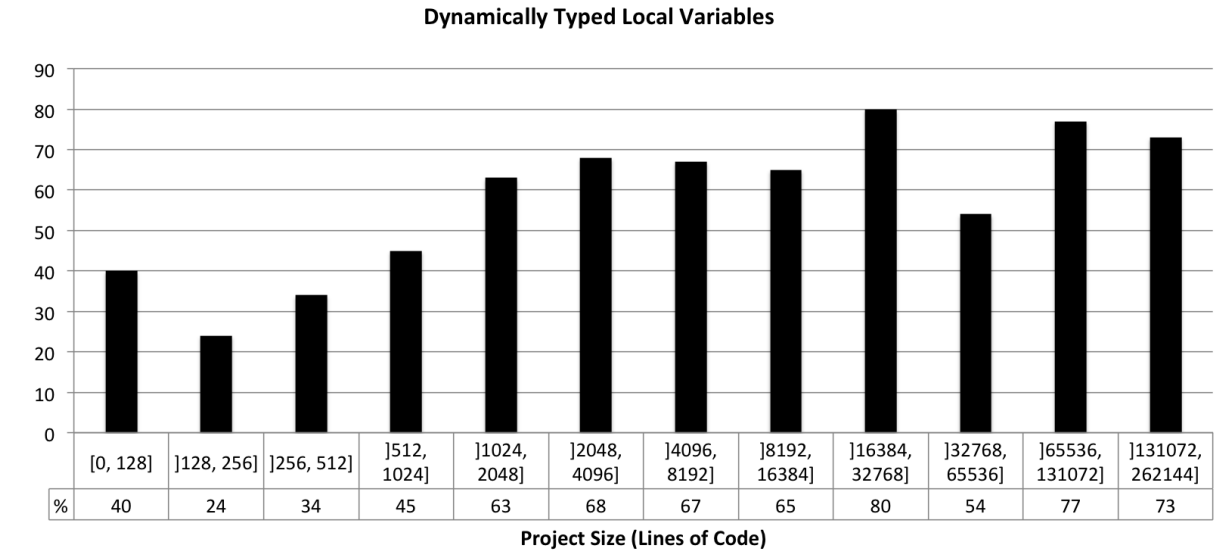


Figure 13. Dynamic Typing usage in local variables

this type of contract should be well document to avoid any problems in such a delicate context.

Results by project size also contribute to a positive answer to Q2. Figure 10 shows that, as projects grow, untyped public statements become less frequent. However, Figures 11 and 12 don't show the same behavior for private or protected statements. Intuitively, the larger the project, the greater the difficulty of integration and the need for maintenance. This may lead programmers to prefer the use types in public statements, the most critical statements in this context. Since no pattern can be observed in other types of statements, this reinforces the idea that this behavior is related to the role of public elements in large projects.

5.3 No types for fast code

Figure 5 shows that dynamic typing is used in local variables declarations with more frequency than in other types of statements. Since local variables have smaller scope and shorter life time, programmers probably feel less need to document them through the definition of types and end up choosing a simpler way to declare them, by using dynamic typing.

5.4 Programmers Background

In order to answer question Q3, we try to correlate how programmers use optional typing and the other languages that these programmers have experience with. Although Groovy is a dynamically

typed language, there is a significant number of programmers with experience in statically typed languages (Figure 4). Understanding how these programmers use optional typing is an important information for those creating languages with hybrid type systems.

Figure 14 shows a significant difference in how those programmers with experience in static languages only use optional typing.

5.5 Scripts e Testes

Scripts are usually written to perform simple tasks and do not relate to many other modules. The same can be said for test code. This suggests that dynamic typing would be used more often in such contexts since maintainability and integration are not critical factors for tests or scripts. The result shown in table **??** however contradicts this idea. There is no significant difference in the usage of typing systems in these contexts.

6. Threats to Validity

As mentioned in section **??**, programmers tend to continue using the type system they are used to. Given that most Groovy programmers have prior experience with Java, a statically typed language, the results shown in this work might present a certain tendency to favor static type systems. Nevertheless, the analysis by type of declaration given in section 4.1 shows the predominance of dynamic typing over local variable declarations, indicating that, despite the previous experience with Java, programmers are able to learn to use dynamic typing where they consider it necessary.

Some frameworks require the use of a particular type system in certain situations. Spock, for example, a automated testing framework, requires that the return of test methods to be dynamically typed. However, due to the heterogeneity and the large number of projects analyzed, we believe that there is no framework capable of having a significant influence on the overall results of this study.

7. Related Work

There are some works in the literature that compare static and dynamic typing systems though controlled studies. In [7], the author compares the performance of two groups of students when 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 type systems 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 this paper, we try to get more relevant results when analyzing source code developed by programmers during their normal activities.

In a follow up study [8], the authors got to opposite conclusions. They compared the performance of two groups of developers in maintenance tasks. First group used Java, a statically typed language, and the other used Groovy, but restricting developer to use only dynamic typing in order to simulate a dynamically typed Java. In this case, the group using the statically typed language, Java, was much more productive. This contradiction reinforces the argument that the results of controlled studies aren't reliable enough for this type of study.

In experiments conducted in [6], 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 static typing in their declarations.

8. Conclusions and Future Work

This study examines what are the most influencing factors in the choice for a static or dynamic type system. There are some controlled studies in the literature about the advantages of each one of these typing system already. However the results presented here focus on finding the factors that have an actual influence on the decision for one or another through the analysis of a wide range of source code repositories.

When maintainability and the complexity of the integration between modules are considered important, static typing is apparently preferred by Groovy programmers. In these situations, the advantages of static typing such as documenting the code or integrating with software development tools, are relevant advantages considered by Groovy programmers. Conversely, when these issues are not as critical, the simplicity of dynamic typing seems to be preferred, as seen on local variables declarations. 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

This work was partially supported by FAPEMIG, grants APQ-02376-11 e APQ-02532-12, and CNPq grant 485235/2011-0.

References

[1] Tiobe programming community index. http://www.tiobe.com/index.php/ content/paperinfo/tpci/index.html. Accessed in 23/09/2013.

[2] Bruce, K. (2002). Foundations of object-oriented languages: types and semantics. MIT press.

[3] 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.

[4] Cardelli, L. (1996). Type systems. ACM Comput. Surv., 28(1):263264.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Lamport, L. and Paulson, L. C. (1999). Should your specification language be typed. ACM Trans. Program. Lang. Syst., 21(3):502526.

[10] 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.

[11] Pierce, B. (2002). Types and programming languages. MIT press.

[12] Tratt, L. (2009). Chapter 5 dynamically typed languages. volume 77 of Advances in Computers, pages 149 184. Elsevier.

[13] Siek, Jeremy, and Walid Taha. "Gradual typing for objects." ECOOP 2007Object-Oriented Programming. Springer Berlin Heidelberg, 2007. 2-27.

[14] Gray, K. E. Safe cross-language inheritance. In European Conference on Object-Oriented Programming (2008), pp. 5275.

[15] Gray, K. E. Interoperability in a scripted world: Putting inheritance & prototypes together. In Foundations of Object-Oriented Languages (2011).

[16] 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.

[17] Siek, J.,Andtaha, W. Gradual typing for objects. In European Conference on Object-Oriented Programming(2007),pp. 227.

[18] Takikawa, Asumu, et al. "Gradual typing for first-class classes." ACM SIGPLAN Notices. Vol. 47. No. 10. ACM, 2012.

[19] Fowler, Martin. Domain-specific languages. Pearson Education, 2010.

[20] Meijer, Erik, and Peter Drayton. "Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages." OOPSLA, 2004.

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

[22] 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.

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

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

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

[26] Eder, Johann, Gerti Kappel, and Michael Schrefl. "Coupling and cohesion in object-oriented systems." Technical Reprot, University of Klagenfurt, Austria (1994).

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