

Understanding Which Factors Influence the Choice for a Typing System

Carlos Souza

Federal University of Minas Gerais
carlosgsouza@gmail.com

Eduardo Figueiredo

Federal University of Minas Gerais
figueido@dcc.ufmg.br

Marco Tlio 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 to 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 dynamically 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 on 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 4 describe the configuration of the study and its results. Section 6 discusses threats 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.

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

Listing 2 A simple algorithm written in Groovy

```

1  class GroovyFilter {
2      List<Integer> evenNumbers(List<Integer>
3          list) {
4          List<Integer> result = []
5          for(int item : list) {
6              if(item % 2 == 0) {
7                  result << item
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
16             ().evenNumbers(list)
17         println result
18     }

```

to every element of the list 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.

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.

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 5 Groovy is a dynamic language

```
1 class DynamicTyping {
2     private String typedField
3     private untypedField
4
5     DynamicTyping(untypedParam, String
6         typedParam) {}
7
8     def untypedMethod(untypedParam, int
9         typedParam) {
10         def untypedVariable = untypedParam +
11             typedParam
12         return result
13     }
14
15     int typedMethod() {
16         String typedVariable = ""
17         return typedVariable
18     }
19 }
```

Listing 6 A class written in Groovy

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

Listing 7 shows an example of static typing in Groovy. Trying to compile the class *TypeCheckedGroovyClass* will produce an 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 section we describe the dataset (subsection 3.4) and the methodology (subsection ??) adopted in this study.

3.1 Research Questoins

- **Question Q1:** Do programmers use types to implicitly document their code? More specifically, is it possible to observe

a higher usage of typing in statements that require more documentation? We consider two characteristics of statements to determine if they need to be more documented than others, the number of modules to which a statement is visible and the complexity of the contract defined by the statement. The higher the number of modules that can access a statement, the more it need to be documented since bad code can break the entire project and other people might need to see that statement. Same for contract complexity. If the contract is not defined correctly, things will explode.

- **Question Q2:** Are untyped declarations considered more often when maintainability or readability aren't important factors? If we are able to observe a higher usage of untyped declarations in this scenario, it means that programmers only use typing when they need maintainability and readability and in other cases, they will favor the objectivity of untyped declarations.
- **Question Q3:** Is the previous experience of a programmer a relevant factor in his or her decision for typing or not typing a statement? I mean, previous experience with typing systems. If yes, we can consider that programmers learn how to deal with type systems and can overcome advantages and disadvantages when they have to. This also imposes a threaten to the above two questions since a programmer might not use types because of the factors above, but because he likes it more.

3.2 Study Design

We will try to answer the questions proposed in section ?? by conducting a case study. We will use a large dataset of Groovy projects to analyze this data. Groovy allows programmers to combine typed and untyped statements flexibly in their code, which makes it possible to answer the questions, where programers use typed or untyped statements. We can cross reference that to the questions above and see if there is any correspondence between the use of types and the hypothesis above.

3.3 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's API 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.

3.4 Dataset

A total of 7268 projects developed by 4806 people was obtained from GitHub. These projects add to a total of 9.8 million lines of code and approximately 169 thousand Groovy files.

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

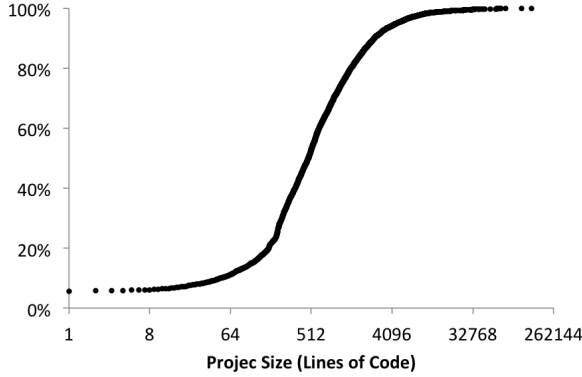


Figure 1. Cumulative Distribution of the Size of the Projects

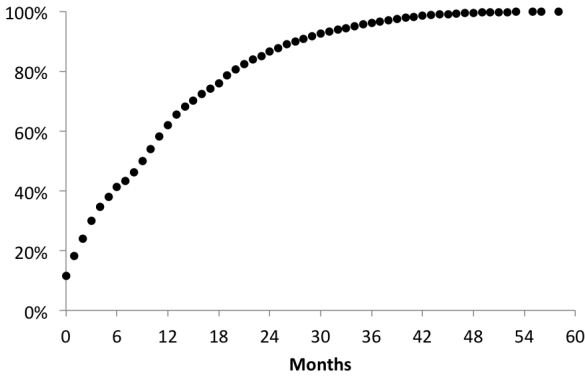


Figure 2. Cumulative Distribution of the Age of the Projects

Java is very popular among Groovy developers. Figure 3 shows that more than 50% of the developers of projects in our dataset also have Java projects on GitHub. In addition to that, 61% of all Groovy programmers have at least one project in another dynamically typed language, while 60% of these developers have at least one project in a statically typed language, including Java.

Most of Table 1 shows that most projects were written by a single developer while the project with most developers had 58 developers.

Table 1. Distribution of the Number of Developers

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

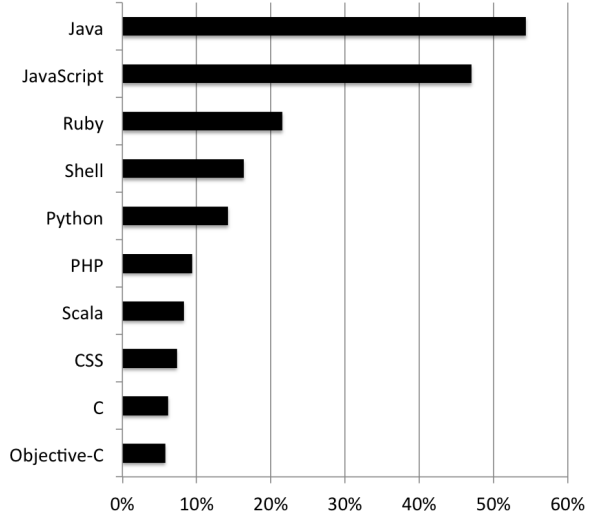


Figure 3. Usage of other languages by Groovy developers on GitHub

3.5 Analysis

The static code analyzer is capable of analyzing the following types of statements

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

For each item listed above, it is possible 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 statement? (except for local variables)

3.6 Static Code Analyzer

The static code analyzer used in this work is based on the Groovy metaprogramming library. With this library, it's possible to create an abstract syntax tree (AST) from Groovy source code using one of the various stages of the Groovy compiler. The phase chosen for this analysis was the conversion phase, which has sufficient information to determine the type system of each statement. It occurs before the compiler tries to resolve any external dependencies, making it possible to analyze each file separately without having to compile the whole project.

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 4 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.

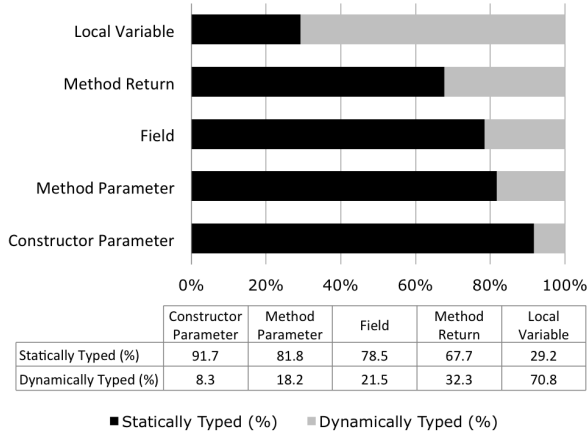


Figure 4. Type Systems by Declaration Type.

4.2 Visibility

In this section, we break down the results shown above by visibility. Our goal is to understand if the visibility of the declarations have any sort of influence on the choice for using types or not.

Figure 5 shows that there is little difference between the use of types in declarations of the return of private and public methods. With 64.44% and 67.26% typed declarations respectively, these results don't present any significant divergence from the overall results shown in section 4. On the other side, the return of protected methods, are almost always typed.

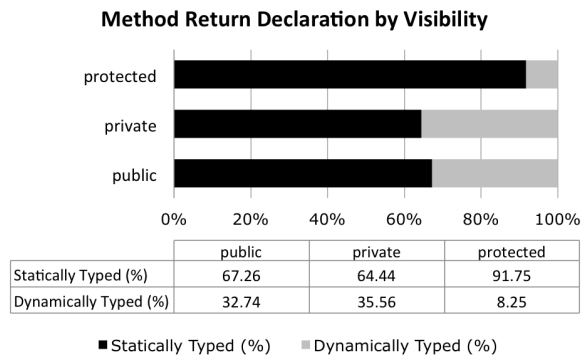


Figure 5. Type Systems by Declaration Visibility

Figure 6 shows the results of the analysis of method parameter declarations. Compared to the results for method returns shown in figure 5, parameters of public methods 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.

Results for constructor parameters are shown in figure 7. These results are close to the results observed for non-constructor method parameters above.

Method Parameter Declaration by Visibility

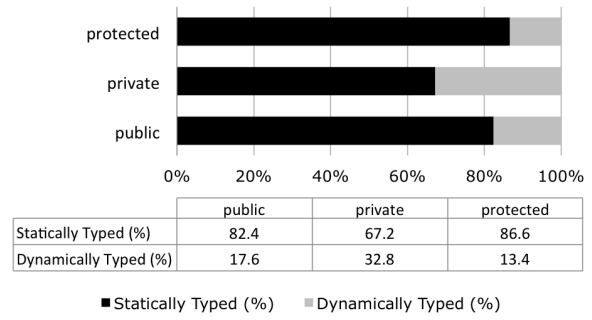


Figure 6. Type Systems by Declaration Visibility

Constructor Parameter Declaration by Visibility

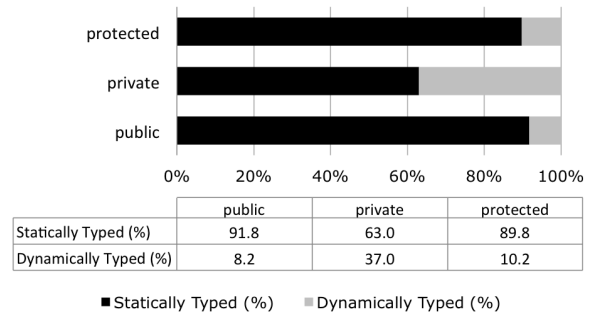


Figure 7. Type Systems by Declaration Visibility

Figure 8 shows the results for field declarations. Different 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.

Field Declaration by Visibility

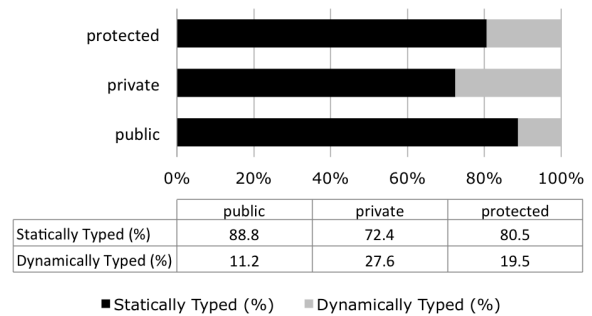


Figure 8. Type Systems by Declaration Visibility

In summary, for all types of declarations, private declarations are less typed than public and protected. Protected declarations, on the other side, are usually those typed in most cases. 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. The goal of this analysis is to understand if there is any correlation between the usage of types and the complexity of software projects.

Project size is a means to measure the complexity of a project. Usually, the larger the project, the greater the difficulty of integration and the need for maintenance. There are metrics far more effective for this purpose such as [TODO]. This simple approach is very useful in a study such as this because it doesn't require the source code to be compiled to determine the project complexity. Requiring projects to be compiled would drastically limit the number of projects

Results are shown for public, private and protected method returns (figures 9, 10, 11 respectively). We also show results for local variables (figure ??). Results for other types of declarations were omitted due to lack of space. The behavior for these other declarations is described in table 2, which shows the spearman ranking for the relation between size and dynamic typing for each type of declaration considered in this study.

In the graphs below, each bar shows the average use of dynamically typed 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. (TODO explain why)

The usage of untyped declarations of public method returns decreases with the project size. While projects with less than 1024 lines of code have almost 40% of their declarations untyped, this number is less than 20% in project with over 65536 lines of code.

Figures 10) and 11) show that, different from public method returns, private and protected method returns have no correlation with the project size.

Conversely, the use of untyped declarations in local variables increases with the project size. This is shown in figure 12. 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%.

TODO: Here I will add some text to explain the spearman ranking of these variables.

Table 2. Spearman Ranking for Dynamic Typing Usage over Size

Method Return	public	?
	private	?
	protected	?
Method Parameter	public	?
	private	?
	protected	?
Constructor Parameter	public	?
	private	?
	protected	?
Field	public	?
	private	?
	protected	?
Local Variable		?

4.4 Scripts

Table 3 shows that the usage of types in scripts is not significantly different from the overall usage shown in section 4.1.

Table 3. Type System used in different contexts

	Static Typing	Dynamic Typing
Scripts	63%	37%
Classes	60%	40%

4.5 Test Classes

There is a slight difference between the definition of types in tests classes and all other classes. This is shown in table 4

Table 4. Type System used in different contexts

	Static Typing	Dynamic Typing
Test Classes	56%	44%
Other Classes	68%	32%

5. Discussion

In this section we build on the results presented in section 4 to discuss what are the factors that we find to be influencing the choice for a static type system.

TODO Please, don't review this section yet. It's still not clear to me how to put together Results and Discussion section. I wanna discuss this a little more before I move on.

5.1 Previous experience of developers

Section 4.1 shows that about 61% of the statements are typed. This results show that, even though Groovy is a dynamic language, Groovy programmers prefer the convenience of typed declarations most of time. Groovy is a popular language among Java developers. Assuming that a significant part of these Groovy developers had some previous experience with Java and were use to static typing, this suggests that the previous experience of a programmer is an important factor in choosing the type system.

TODO This section is really weak without solid data that indicates that these developers have some experience with Java. I suggest getting the fraction of developers who also have Java projects in GitHub. Maybe we even could show the results above grouped by developers who have Java projects and other developers.

5.2 Scope and life cycle

Scope and life cycle seem to be

Figure 4 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.

Conversely, constructor parameters are the declaration type with the most static typing usage. It can be argued that there is a relevant concern in documenting these statements by the usage of static typing since these are important elements in the definition of the contract of a module.

5.3 Results by Visibility

According to figure 5, statements with public or protected visibility are those more frequently statically typed. These statements define the interface of a module and, by typing them, programmers allow the compiler to look for typing problems in the integration with other modules. In addition to that these statements should be well documented so clients of this module know how to use it making static typing a good option for that purpose.

In the case of parameters and returns of methods, the usage of types in statements with protected visibility even surpasses that of

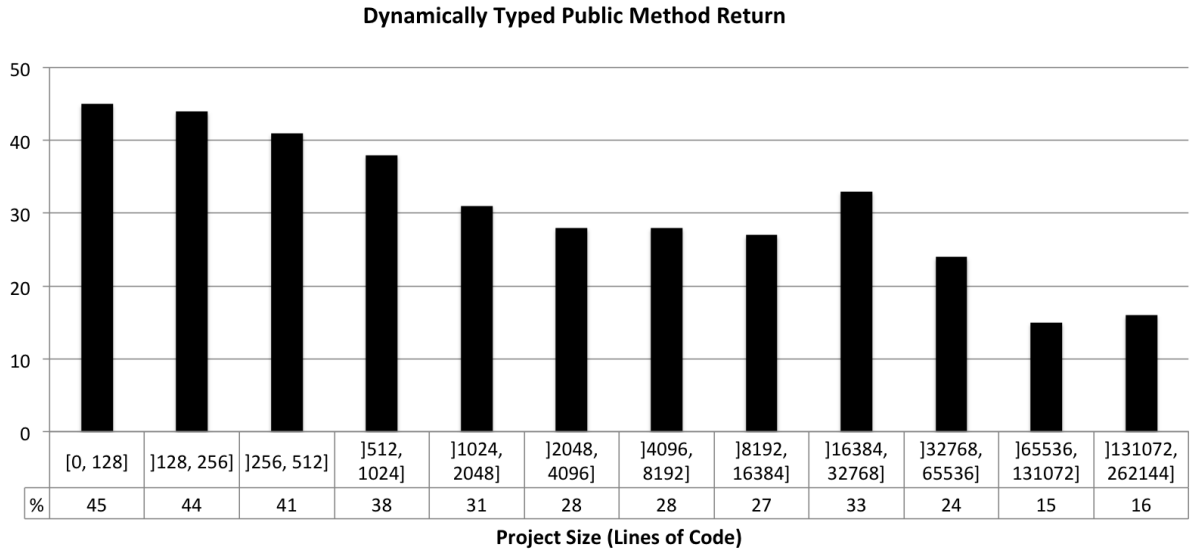


Figure 9. Dynamic Typing usage in local variables

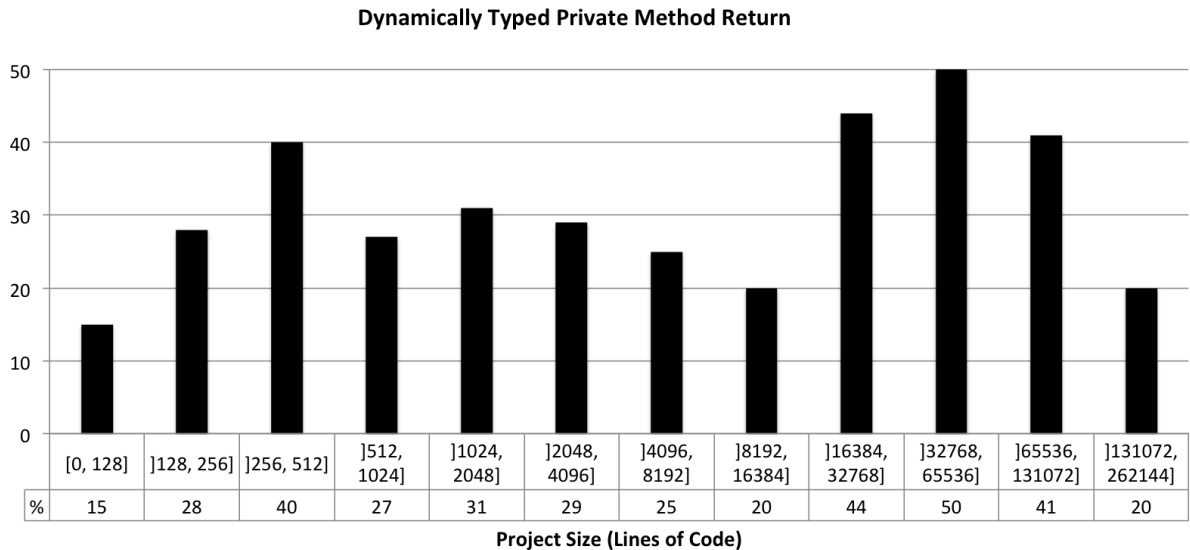


Figure 10. Dynamic Typing usage in local variables

statements with public visibility. It can be argued that protected fields and methods establish a tight contract, since they expose internal elements of a superclass to a subclass. Apparently programmers understand that this type of contract should be well documented to avoid any problems in such a delicate context.

5.4 Results by Project Size

Figure 9 shows the use of dynamic typing in public statements by project size. Each bar of this graph show the result for 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 second group, since 1500 lies within the interval]400, 1600].

This result shows that the use of dynamic typing in public statements decreases as the size of the project increases. The frequency of dynamic typing usage in projects with more than 6400 lines of code is almost half the frequency of smaller projects. Intuitively, the larger the project, the greater the difficulty of integration and the need for maintenance. This may lead programmers to prefer the use of static typing in public statements, the most critical statements in this context. No pattern can be observed in other types of statements, reinforcing the idea that this behavior is related to the role of public elements in large projects.

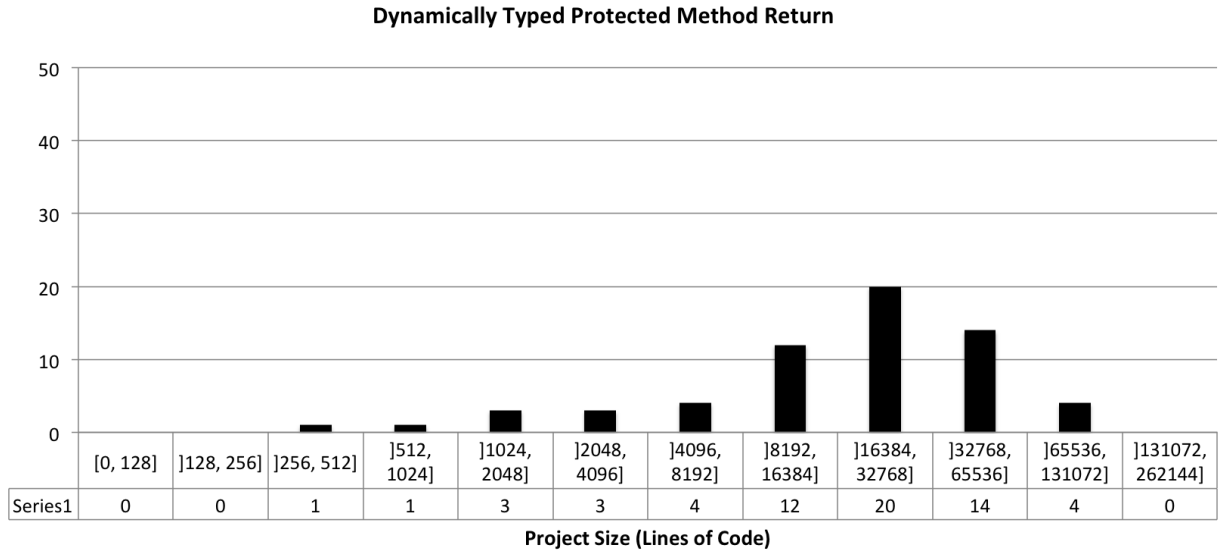


Figure 11. Dynamic Typing usage in local variables

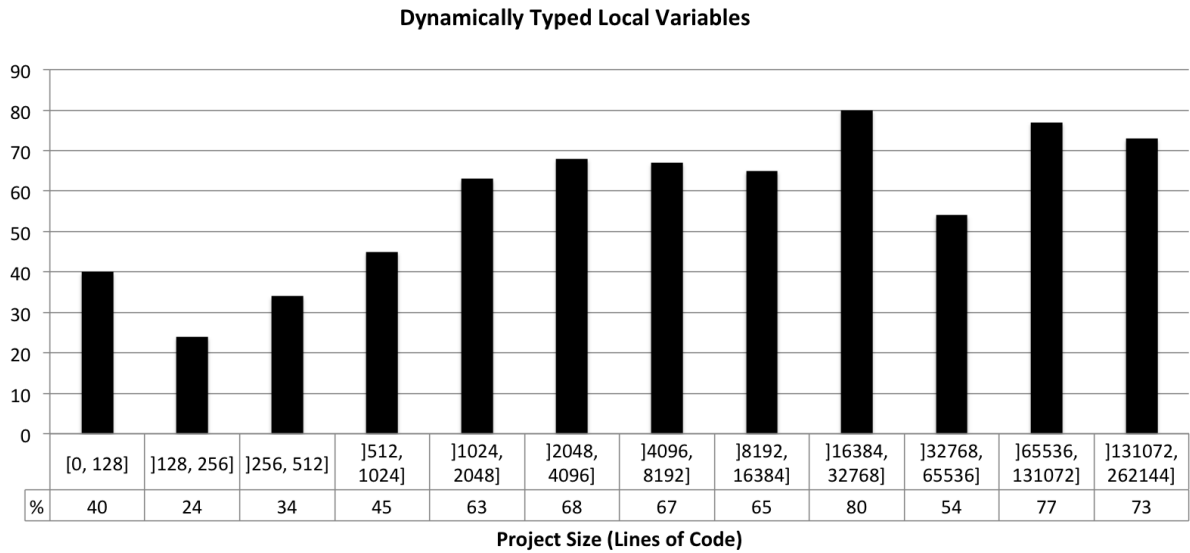


Figure 12. Dynamic Typing usage in local variables

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 4.1, 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 4.2 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 frame-

work, 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 through 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 23/06/2013.
- [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 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations 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] Lampert, 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., and Stefik, A. (2012). Static types 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 2007 Object-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., AND FLATT, M. Fine-grained interoperability through contracts and mirrors. In Object-Oriented Programming, Systems, Languages, and Applications (2005), pp. 231245.
- [17] SIEK, J., AND TAHA, 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.