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

figureido@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 must consider several characteristics of that language. One of the most important is its typing system. This, which can be static or dynamic, determines when the type of a statement should 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 betwen 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 changin 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 5 describe the configuration of the study and its results. Section 7 discusses threatens to validity and section 8 shows related work. Finally, section 9 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. It builds upon the strengths of Java, but has additional features inspired by dynamic languages such as Python and Ruby. 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. Groovy allows programmers to use both static and dynamic type systems, which makes it ideal for a comparative study between those two type systems.

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 with the even numbers of that list. Listing 1 shows the Java implementation while Listing 2 shows the Groovy counterpart.

Many of the boilerplate elements from Java aren't required in Groovy. As shown in Listing 2, 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, frequently used classes like those of the $java.util$ package are imported by default and thus don't need to be implicitly imported. Least, Groovy provides easy access to frequently used methos such as $System.out.println$ (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 heavily influenced by dynamic programming langues like Ruby and Python. It provides many modern programming features like those seen in Ruby and Python.

Although Groovy language is similar to Java, it was designed to be more expressive and concise. Algorithm ?? shows some

¿ Functional Programming style ¿ Operator ¿ Closures ¿ Scripting

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 }
```

¿ Metaprogramming
¿ Native Regex support

In Groovy, a programmer can choose between static and dynamic typing for each of their statements. Dynamic and static Typing can be combined flexibly. In the algorithm 8, for example, the method return is statically typed, while parameters and local variables are dynamically typed.

Listing 3 A class written in Groovy

```
List<Integer> result = []
[1, 2, 3, 5, 8].each { item ->
  if(item % 2 == 0) {
    result << item
  }
}
println result
```

Listing 4 A class written in Groovy

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

Listing 5 Groovy is a dynamic language

```
class DynamicGroovy {
  private String field_s
  private field_d = 1.0

  DynamicGroovy()

  def sum(param_d, int param_s) {
    def local_d = param_d + param_s
    return result
  }
}
```

TODO: Talk about gradual type systems [14–18]

Listing 6 A class written in Groovy

```
int thisIsAnInt = "this is a string"
```

Listing 7 A class written in Groovy

```
class MyRunnable implements Runnable {}

/*
ERROR MESSAGE

$ groovyc SubclassTypeError.groovy
org.codehaus.groovy.control.
    MultipleCompilationErrorsException:
    startup failed:
SubclassTypeError.groovy: 1: Can't have an
    abstract method in a non-abstract class.
    The class 'MyRunnable' must be declared
    abstract or the method 'void run()'
    must be implemented.
 @ line 1, column 1.
    class MyRunnable implements Runnable {}
    ^

1 error
*/
```

Listing 8 A class written in Groovy

```
@TypeChecked
class TypeCheckedGroovyClass {

  static int sum(int a, int b) {
    a + b
  }

  public static void main(String[] args) {
    println sum("1", "2")
  }
}

/*
olga:listings carlosgsouza$ groovyc
    TypeCheckedGroovyClass.groovy
org.codehaus.groovy.control.
    MultipleCompilationErrorsException:
    startup failed:
TypeCheckedGroovyClass.groovy: 11: [Static
    type checking] - Cannot find matching
    method TypeCheckedGroovyClass#sum(java.
    lang.String, java.lang.String). Please
    check if the declared type is right and
    if the method exists.
 @ line 11, column 11.
    println sum("1", "2")
    ^

1 error
*/
```

3. Study Setup

TODO: Right now I only have the static code analyzer here. I will refactor this section so I can list all the phases of my study. It's still not clear where I can describe this static code analyzer though.

3.1 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.

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)

4. Dataset

TODO: This is also just the VEM version. The purpose of this section is to present data about my dataset that is fundamental for my discussion. Also, I believe that having a dedicated section to my dataset is a good idea as it highlights one of the strong points of this article.

The projects used in this study were obtained from GitHub, a popular source control service based on Git. Using Github API, it was possible to obtain the source code of 7268 projects with a total of 9.8 million lines of code. The distribution of the size of these designs is shown in figure 1. These projects were developed by a total of 4806 developers and, together, they have approximately 169 thousand Groovy files.

Table 1. Distribution of the Number of Developers

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

5. Results

In this section we present the results of the analysis of the declarations of the projects described in section 4. These results will be discussed in more details in section 6.
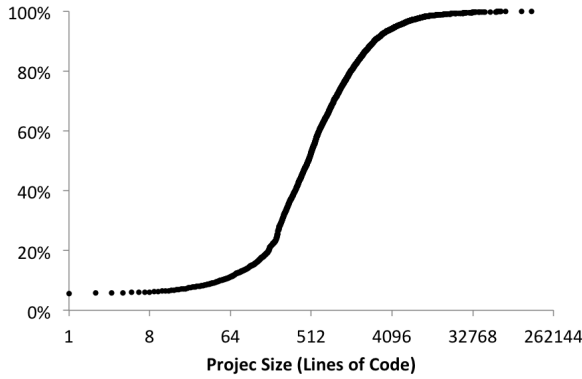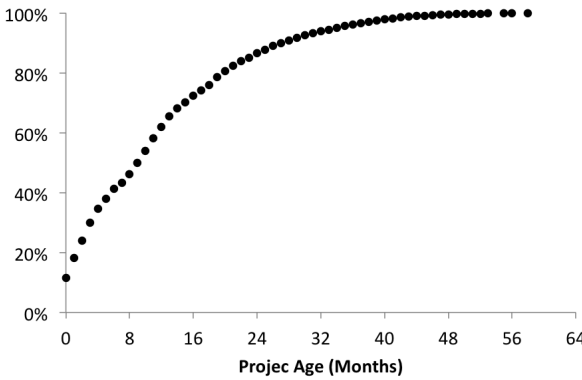
Figure 1. Project Size Distribution

Figure 2. Project Age Distribution

	Constructor Parameter	Method Parameter	Field	Method Return	Local Variable
Statically Typed (%)	91.7	81.8	78.5	67.7	29.2
Dynamically Typed (%)	8.3	18.2	21.5	32.3	70.8

■ Statically Typed (%)　■ Dynamically Typed (%)

Figure 3. Type Systems by Declaration Type.

Method Return Declaration by Visibility

	public	private	protected
Statically Typed (%)	67.26	64.44	91.75
Dynamically Typed (%)	32.74	35.56	8.25

■ Statically Typed (%)　■ Dynamically Typed (%)

Figure 4. Type Systems by Declaration Visibility

Method Parameter Declaration by Visibility

	public	private	protected
Statically Typed (%)	82.4	67.2	86.6
Dynamically Typed (%)	17.6	32.8	13.4

■ Statically Typed (%)　■ Dynamically Typed (%)

Figure 5. Type Systems by Declaration Visibility

5.1 Overall Result

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.

5.2 Results by Declaration Type

Figure 3 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.

5.3 Results by 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 4 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 5. On the other side, the return of protected methods, are almost always typed.

Figure 5 shows the results of the analysis of method parameter declarations. Compared to the results for method returns shown in figure 4, 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.
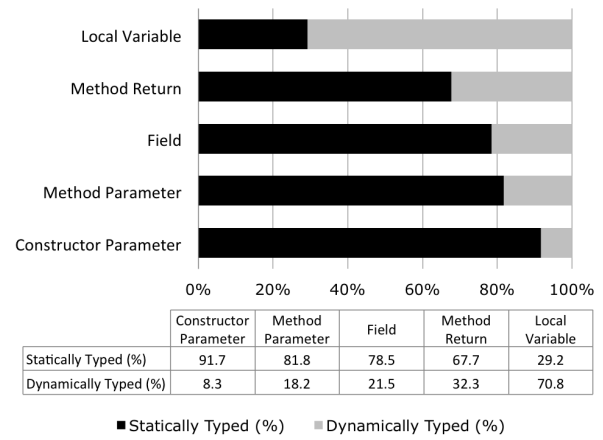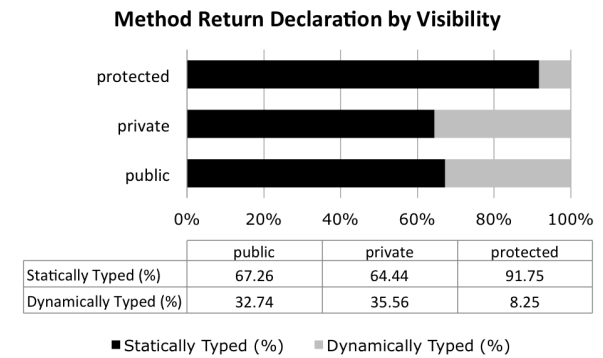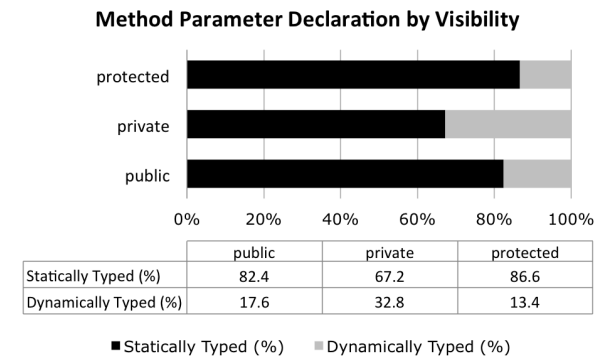
Results for constructor parameters are shown in figure 6. These results are close to the results observed for non-constructor method parameters above.

Figure 7 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
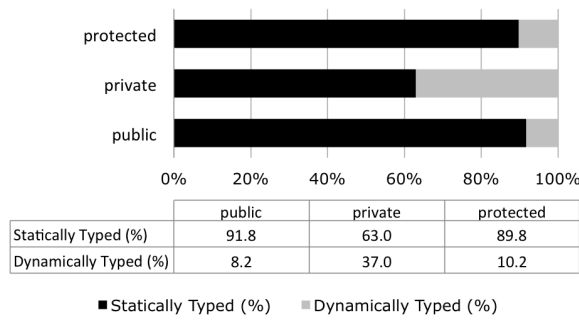
Constructor Parameter Declaration by Visibility

	public	private	protected
Statically Typed (%)	91.8	63.0	89.8
Dynamically Typed (%)	8.2	37.0	10.2

■ Statically Typed (%)   ■ Dynamically Typed (%)

Figure 6. Type Systems by Declaration Visibility

returns, method parameters and constructor parameters don't get typed more than 67% of the time.

Field Declaration by Visibility

	public	private	protected
Statically Typed (%)	88.8	72.4	80.5
Dynamically Typed (%)	11.2	27.6	19.5

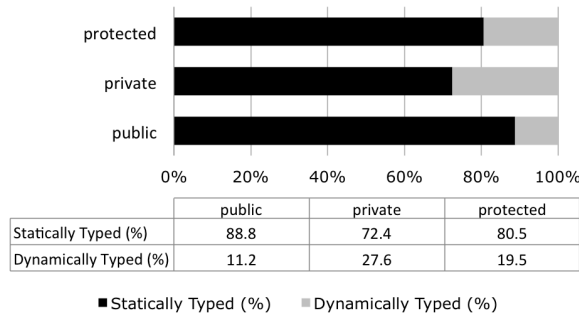■ Statically Typed (%)   ■ Dynamically Typed (%)

Figure 7. 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%.

5.4 Results by 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 8, 9, 10 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 9) and 10) 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 11. 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

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

5.5 Scripts

Table 3 shows that the usage of types in scripts is not significantly different form the overall usage shown in section 5.1.

Table 3. Type System used in different contexts

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

5.6 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%

6. Discussion

In this section we build on the results presented in section 5 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 sextion 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.
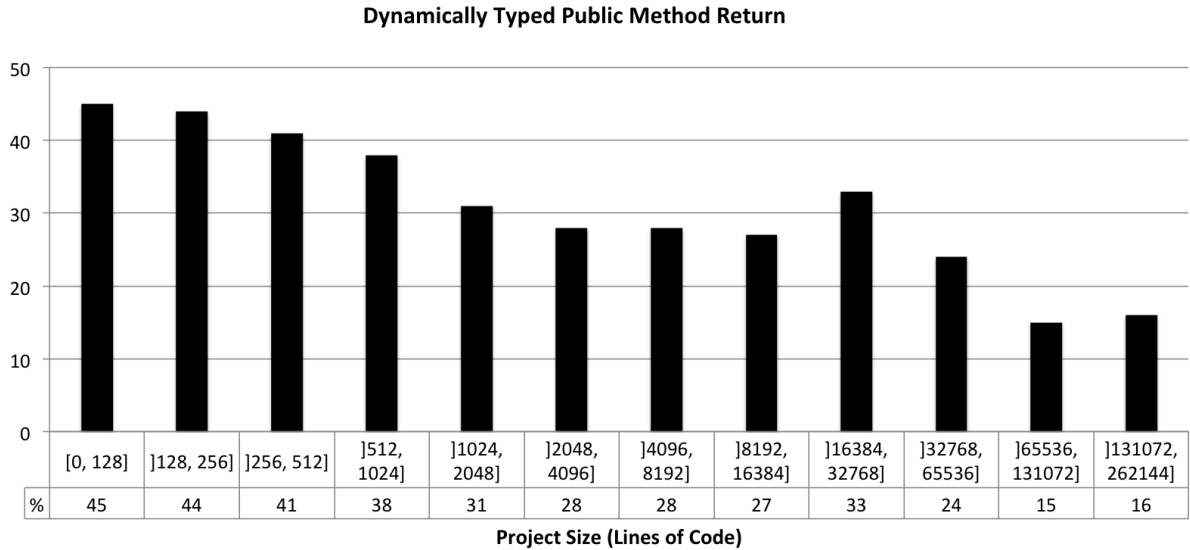
Dynamically Typed Public Method Return

%	[0, 128]	]128, 256]	]256, 512]	]512, 1024]	]1024, 2048]	]2048, 4096]	]4096, 8192]	]8192, 16384]	]16384, 32768]	]32768, 65536]	]65536, 131072]	]131072, 262144]
	45	44	41	38	31	28	28	27	33	24	15	16

Project Size (Lines of Code)

Figure 8. Dynamic Typing usage in local variables

Dynamically Typed Private Method Return

%	[0, 128]	]128, 256]	]256, 512]	]512, 1024]	]1024, 2048]	]2048, 4096]	]4096, 8192]	]8192, 16384]	]16384, 32768]	]32768, 65536]	]65536, 131072]	]131072, 262144]
	15	28	40	27	31	29	25	20	44	50	41	20

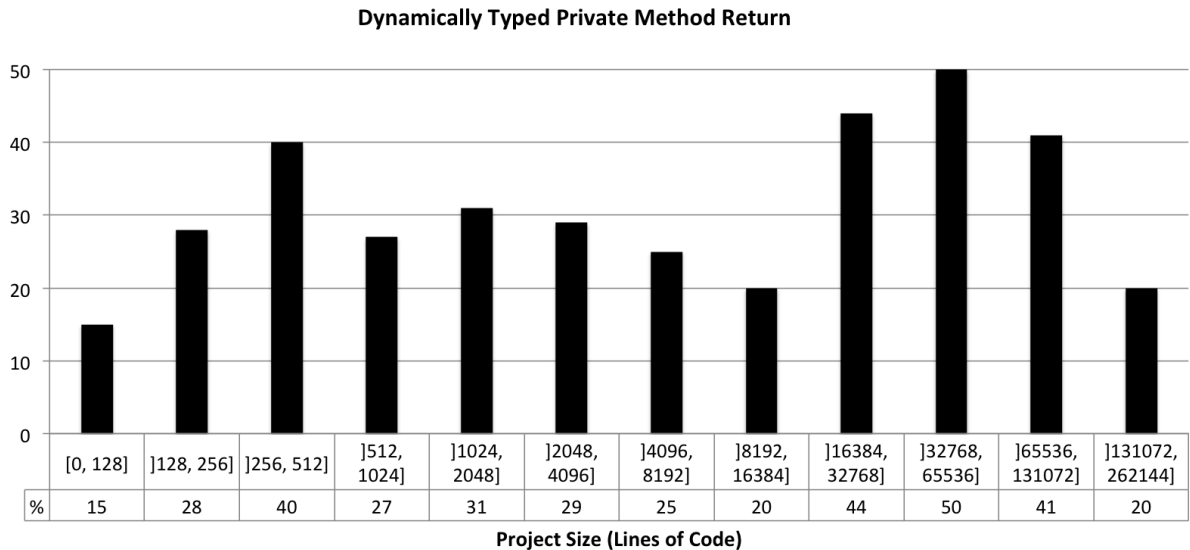**Project Size (Lines of Code)**

Figure 9. Dynamic Typing usage in local variables

6.1 Previous experience of developers

Section 5.1 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.

6.2 Scope and life cycle

Scope and life cycle seem to be

Figure 3 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.
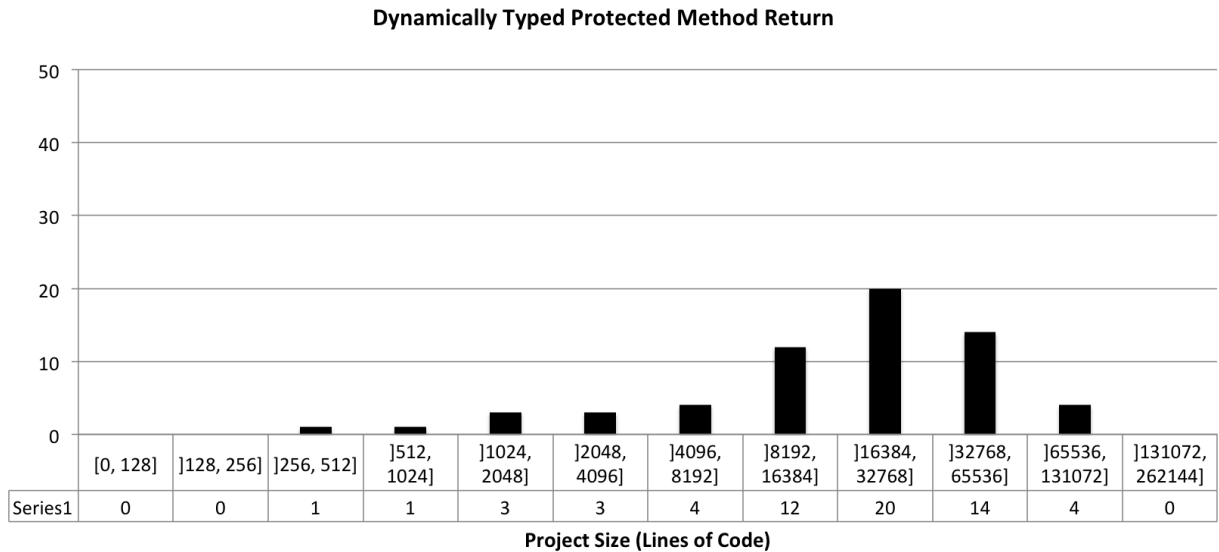
Dynamically Typed Protected Method Return

	[0, 128]	]128, 256]	]256, 512]	]512, 1024]	]1024, 2048]	]2048, 4096]	]4096, 8192]	]8192, 16384]	]16384, 32768]	]32768, 65536]	]65536, 131072]	]131072, 262144]
Series1	0	0	1	1	3	3	4	12	20	14	4	0

Project Size (Lines of Code)

Figure 10. Dynamic Typing usage in local variables

Dynamically Typed Local Variables

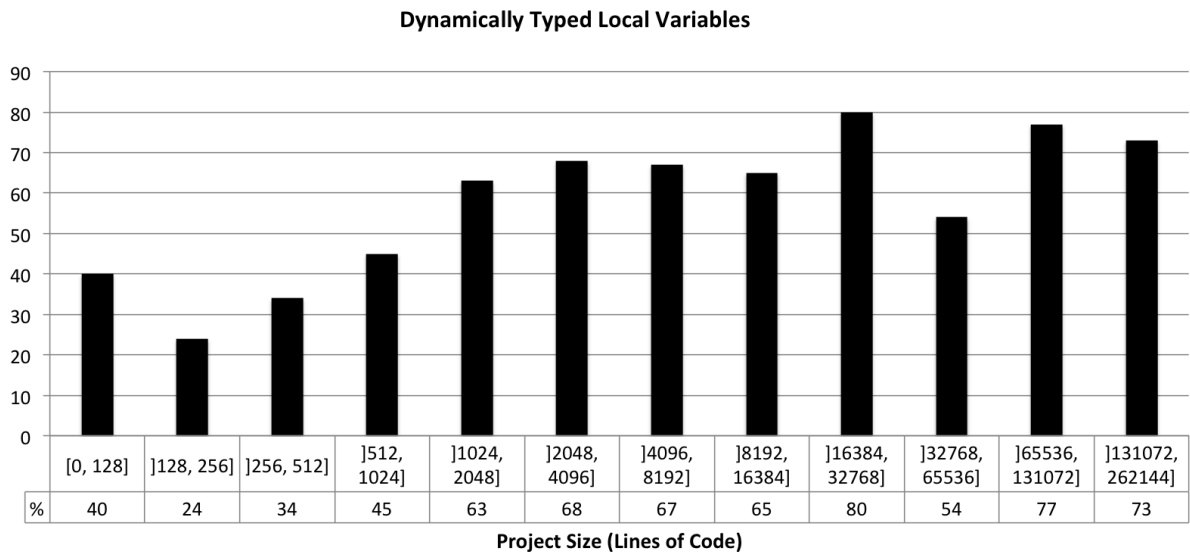|	[0, 128]	]128, 256]	]256, 512]	]512, 1024]	]1024, 2048]	]2048, 4096]	]4096, 8192]	]8192, 16384]	]16384, 32768]	]32768, 65536]	]65536, 131072]	]131072, 262144]
%	40	24	34	45	63	68	67	65	80	54	77	73

Project Size (Lines of Code)

Figure 11. Dynamic Typing usage in local variables

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.

6.3 Results by Visibility

According to figure 4, 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 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 document to avoid any problems in such a delicate context.

6.4 Results by Project Size

Figure 8 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.

6.5 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.

7.   Threats to Validity

As mentioned in section 5.1, 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 5.2 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 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.

8.   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.

9.   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.

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 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).Statictypesys- tems (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.