Tipar ou não tipar? Compreendendo quais Fatores Influenciam a Escolha por um Sistema de Tipos

Carlos Souza, Eduardo Figueiredo, Marco Tulio Oliveira Valente

[1]Departamento de Ciência da Computação, UFMG, Brasil

carlos.garcia@dcc.ufmg.br, figueiredo@dcc.ufmg.br, mtov@dcc.ufmg.br

Abstract. *One of the most important features 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 a thousand projects written in Groovy, a programming language where one can choose, for each declaration, either to type it or not, 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.*

Resumo. *Uma das características mais importantes a serem consideradas na hora de se escolher uma linguagem de programação é o seu sistema de tipos, estático ou dinâmico. Essa questão tem se tornado cada vez mais importante graças à popularização recente de linguagens dinamicamente tipadas como Ruby e JavaScript. Este trabalho apresenta um estudo sobre quais fatores mais influenciam a escolha de um programador por um sistema de tipos. Uma análise do código fonte de mais mil projetos escritos em Groovy, uma linguagem de programação onde se pode escolher, para cada declaração, usar tipos ou não, permite visualizar em quais situações um sistema de tipos é preferido. Resultados deste estudo apontam que a experiência prévia do programador, tamanho do projeto, complexidade dos módulos, escopo e visibilidade das declarações são alguns dos fatores mais importantes para essa decisão.*

1. Introdução

Ao se escolher uma linguagem de programação para um projeto, um desenvolvedor deve considerar algumas características desta linguagem, sendo uma das mais importantes o sistema de tipos. Este, que pode ser estático ou dinâmico, define em que momento o tipo de uma declaração deve ser definido [Pierce 2002]. Declarações em linguagens com tipagem estática, como Java e C#, devem ser acompanhadas pela definição de um tipo, que pode ser usado pelo compilador para checar a corretude do código. Já em linguagens dinamicamente tipadas, como Ruby e JavaScript, a definição do tipo só é realizada em tempo de execução.

A discussão sobre qual sistema de tipos é melhor tem se tornado cada vez mais relevante nos últimos anos graças à rápida popularização de linguagens dinamicamente tipadas. De acordo com o TIOBE Programming Community Index[tio], um conhecido

ranking que mede a popularidade de linguagens de programação, 27% das linguagens de programação adotadas na indústria possuem tipagem dinâmica. Em 2001, esse número era de apenas 17%. Entre as 10 linguagens no topo do ranking, 4 possuem sistemas de tipos dinâmicos: JavaScript, Perl, Python e PHP. Em 1998, nenhuma dessas linguagens estava entre as 10 primeiras do ranking.

Diversos fatores podem ser considerados na escolha por uma linguagem de programação com sistema de tipos dinâmico ou estático. Linguagens com tipagem dinâmica , por serem mais simples, permitem que programadores executem suas tarefas de desenvolvimento mais rapidamente[Pierce 2002]. Ainda, ao removerem o trabalho de declarar os tipos das variáveis, estas linguagens permitem que seus usuários foquem no problema a ser resolvido, ao invés de se preocuparem com as regras da linguagem [Tratt 2009].

Por outro lado, sistemas de tipos estáticos, também possuem suas vantagens. Estes conseguem prevenir erros de tipo em tempo de compilação [Lamport and Paulson 1999]. Declarações de tipos aumentam a manutenibilidade de sistemas pois estas atuam como documentação do código, informando ao programador sobre a natureza de cada variável [Cardelli 1996, Mayer et al. 2012] . Sistemas escritos a partir destas linguagens tendem a ser mais eficientes, uma vez que não precisam realizar checagem de tipo durante sua execução[Bruce 2002, Chang et al. 2011]. Por fim, ambientes de desenvolvimento modernos, tais como Eclipse e IDEA, quando possuem conhecimento sobre o tipo de uma declaração, são capazes de auxiliar o programador através de funcionalidades como documentação e complemento de código [Bruch et al. 2009].

Este artigo apresenta um estudo com o objetivo de entender quais dos fatores descritos acima influenciam de fato a escolha de um programador por tipar ou não as suas declarações. A fim de obter resultados confiáveis, essa questão foi estudada tendo como base código desenvolvido por programadores no contexto de suas atividades cotidianas através da análise de uma massa de dados composta por mais de mil projetos. Esses projetos foram escritos em Groovy, uma linguagem com sistema de tipos híbrido, que permite escolher, para cada declaração, tipá-la ou não. Assim, através de uma análise estática dessa massa de dados, é possível visualizar quando programadores escolhem cada sistema de tipos e, a partir daí, enteder quais são os fatores que influenciam essa decisão.

O restante deste artigo está organizado da seguinte forma. A seção 2 introduz os principais conceitos da linguagem de programação Groovy. As seções 3 e 4 descrevem a configuração do estudo e seus resultados. Ameaças à validade deste trabalho são discutidas na seção 5 enquanto alguns trabalhos relacionados são apresentados na seção 6. Por fim, a seção 7 conclui este trabalho e levanta alguns trabalhos futuros.

2. A Linguagem Groovy

Groovy é uma linguagem de programação orientada a objetos projetada para ser executada sobre a plataforma Java, mas com características dinâmicas semelhantes às de Ruby e Python. Sua adoção tem crescido de maneira notável nos últimos anos e, apesar de ter sido lançada há apenas 6 anos, Groovy já é a 36ª linguagem mais popular da indústria de software[tio].

Em Groovy, um programador pode escolher tipar suas declarações ou não. Tipagem estática e dinâmica podem ser combinadas no mesmo código livremente. No algo-

ritmo 1, por exemplo, o tipo do retorno do método é definido, enquanto os parâmetros e a variável local são tipados dinamicamente.

Algoritmo 1 Um método escrito em Groovy

```
1.  Integer add(a, b) {
2.      def c = a + b
3.      return c
4.  }
```

A maior parte da linguagem Java também é válida em Groovy e código Groovy pode interagir diretamente com código Java e vice-versa. Esses fatores tem atraído um grande número de programadores Java que desejam utilizar suas funcionalidades sem ter que aprender uma linguagem completamente diferente ou mudar a plataforma de execução de seus sistemas. Outros recursos interessantes de Groovy são suporte nativo a coleções, possibilidade de se escrever scripts e metaprogramação.

3. Configuração do Estudo

Este trabalho analisa em quais declarações programadores preferem utilizar tipagem estática ou dinâmica a fim de entender quais fatores influenciam nessa escolha. Abaixo são descritos a massa de dados e o analisador estático usados para tal.

3.1. Massa de Dados

Os projetos utilizados neste estudo foram obtidos do GitHub, um serviço de controle de versão baseado em Git. Utilizando a API do GitHub, foi possível obter o código fonte de quase dois mil projetos Groovy. Após descartar projetos privados e duplicados, restaram 1112 projetos com um total de 1,67 milhões de linhas de código, considerando apenas a última versão de cada projeto. A distribuição do tamanho destes projetos é mostrada na figura 1. Estes sistemas foram desenvolvidos por um total de 926 programadores.

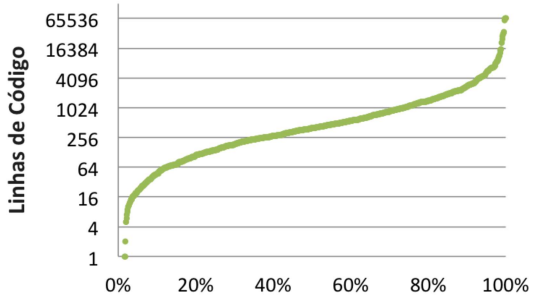

Figura 1. Distribuição do tamanho dos projetos

3.2. Analisador de código estático

O analisador de código estático utilizado neste trabalho é baseado na biblioteca de metaprogramação de Groovy. Com esta biblioteca, é possível criar uma árvore sintática abstrata(AST) a partir de código fonte utilizando uma das diversas fases do próprio compilador de Groovy. A fase escolhida foi a de de conversão, que possui informação suficiente

para determinar o sistema de tipos de cada declaração. Essa fase acontece antes do compilador tentar resolver quaisquer dependências externas, tornando possível analisar cada arquivo separadamente sem que seja necessário compilar o projeto.

Os seguintes tipos de declarações podem ser analisados

- Retorno de Métodos
- Parâmetros de Métodos
- Parâmetros de Construtores
- Campos
- Variáveis Locais

Para cada item listado acima, é possível obter ainda as seguintes informações

- A declaração é parte de um script ou de uma classe?
- A declaração é parte de uma classe de testes?
- Visibilidade (exceto para variáveis locais)

4. Resultados

A seguir são apresentados os resultados deste trabalho.

4.1. Resultado Geral

Cerca de 60% das declarações são estaticamente tipadas, enquanto apenas 40% destas são dinamicamente tipadas. Dado que grande parte dos programadores Groovy eram previamente programadores Java e, portanto, estavam acostumados com tipagem estática, este resultado sugere que a experiência prévia de um programador é um fator importante na escolha do sistema de tipos.

4.2. Resultados por Tipo de Declaração

A figura 2 mostra que tipagem dinâmica é utilizada em declarações de variáveis locais com muito mais frequência que em outros tipos de declaração. Dado que variáveis locais possuem menor escopo e menor ciclo de vida, programadores provavelmente sentem menor necessidade de documentá-las através da definição de tipos e acabam optando pela maneira mais direta de declará-las.

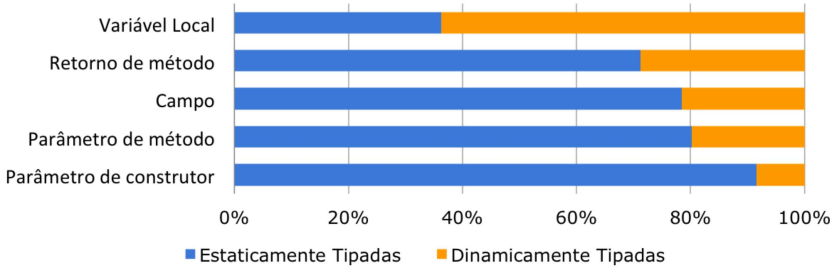

Figura 2. Sistemas de Tipo por Tipo de Declaração.

Por outro lado, parâmetros de construtores são o tipo de declaração mais frequentemente tipado. Pode se argumentar que há uma preocupação grande em tipar (e documentar) construtores uma vez que estes são importantes elementos da definição do contrato de um módulo.

4.3. Resultados por Visibilidade

De acordo com a figura 3, declarações com visibilidade pública ou protegida são as que, com mais frequência, utilizam tipagem estática. Essas são as declarações que definem a interface de um módulo e, ao tipá-las, programadores permitem que o compilador procure por erros de tipos na integração com outros módulos alem de documentar o contrato deste módulo para que clientes saibam como utilizá-lo.

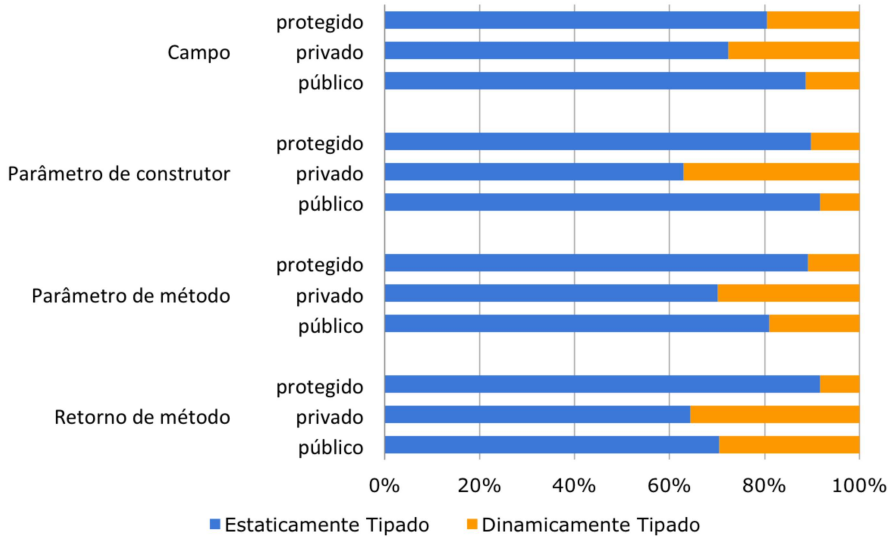

Figura 3. Sistemas de Tipo por Visibilidade da Declaração

No caso de retorno e parâmetros de métodos, o uso de tipos em declarações com visibilidade protegida chega a superar o de declarações com visibilidade pública. Pode-se argumentar que métodos e campos protegidos estabelecem um contrato delicado, já que expõem elementos internos de uma superclasse para uma subclasse. Aparentemente programadores enxergam a necessidade de documentar bem o código que define esse tipo de contrato através do uso tipagem estática.

4.4. Resultados por Tamanho de Projeto

A figura 4 mostra o uso de tipagem dinâmica em declarações públicas por tamanho de projeto. Cada barra desse gráfico representa um grupo de projetos agrupado por seu tamanho sendo que os limites de cada grupo são definidos sob cada barra. Por exemplo, um projeto com 1500 linhas de código se encontra na segunda barra, pois 1500 se encontra dentro do intervalo $]400, 1600]$.

Este resultado mostra que o uso de tipagem dinâmica em declarações públicas diminui à medida que o tamanho do projeto aumenta. Projetos com mais de 6400 linhas de código usam tipagem dinâmica com praticamente metade de frequência que projetos menores. Intuitivamente, quanto maior o projeto, maior a dificuldade de integração e a necessidade de manutenção, o que pode levar programadores a preferirem o uso de tipagem estática em elementos públicos, os mais críticos para esse contexto. Nenhum padrão pode ser observado em outros tipos de declaração, reforçando a idéia de que esse padrão está relacionado ao papel de elementos públicos em projetos grandes.

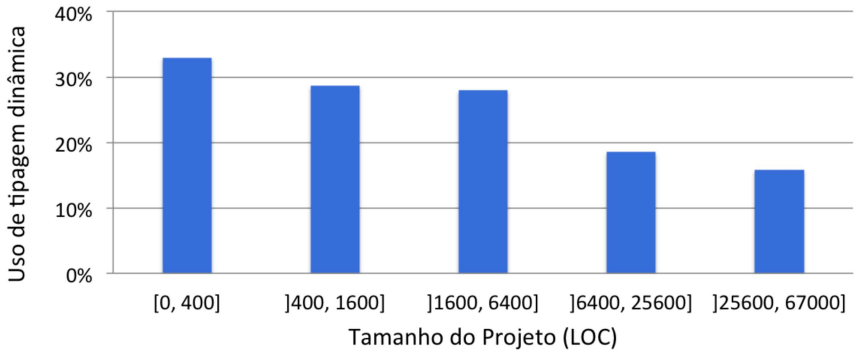

Figura 4. Sistemas de tipo de retornos e parâmetros de métodos públicos agrupados por tamanho de projeto.

4.5. Scripts e Testes

Scripts são, em geral, escritos para desempenhar tarefas simples e não se relacionam com muitos outros módulos. O mesmo pode ser dito a respeito de código de teste. Isso leva a crer que tipagem dinâmica seria utilizada com mais frequência nesses contextos uma vez que manutenibilidade e integração não são fatores críticos. O resultado da tabela 1 porem contradiz essa intuição mostrando que não há diferença significativa entre o perfil de uso dos sistemas de tipos nesses contextos.

Tabela 1. Todos os tipos de declarações agrupadas por classe/script e classes de teste/classes funcionais

	Tipagem Estática	Tipagem Dinâmica
Todas as Classes	61%	39%
Scripts	54%	46%
Classes Funcionais	62%	38%
Classes de Teste	57%	43%

5. Ameaças à validade

Como levantado na seção 4.1, programadores tendem a continuar usando o sistema de tipos com o qual já estão acostumados. Dado que grande parte dos programadores Groovy possuem experiência prévia com Java, uma linguagem estaticamente tipada, os resultados mostrados neste trabalho podem apresentar certa tendência ao uso de tipagem estática. Apesar disso, a análise por tipo de declaração apresentada na seção 4.2 mostra a predominância de tipagem dinâmica em variáveis locais, indicando que, apesar da experiência anterior com Java, programadores são capazes de aprender a usar tipagem dinâmica onde julgam necessário.

Alguns arcabouços impõem o uso de um dado sistema de tipos em certas situações. Spock, por exemplo, um arcabouço de testes automatizados, requer que o retorno de métodos que implementam testes seja dinamicamente tipado. Porem, graças à heterogeneidade e ao grande número de projetos analisados, acredita-se que não haja nenhum arcabouço com utilização tão extensa a ponto de influenciar os resultados gerais.

6. Trabalhos Relacionados

Há alguns trabalhos que realizam essa comparação através de estudos controlados. Em[Hanenberg 2010], o autor compara o desempenho de dois grupos de estudantes quando instruídos a desenvolver dois pequenos sistemas. Ambos os grupos utilizaram uma linguagem desenvolvida pelo autor, Purity, sendo que a única diferença entre eles é que um grupo utiliza uma versão desta linguagem com tipagem estática enquanto o outro utilizou uma versão com tipagem dinâmica. Resultados mostraram que o grupo utilizando a versão dinâmica foi significativamente mais produtivo. Assim como neste trabalho, o autor foi capaz de comparar dois sistemas de tipos diretamente, neste caso desenvolvendo sua própria linguagem. Porem, pode-se argumentar que esses resultados podem não representar bem situações do cotidiano da indústria de software, uma vez que esse foi um estudo de pequena duração onde estudantes são utilizados como exemplos de desenvolvedores e que, ainda, não possuem nenhum tipo de interação com outros programadores. Neste trabalho, tenta-se conseguir resultados mais relevantes ao analisar código fonte desenvolvido pro programadores durante suas atividades cotidianas.

Em uma continuação do estudo acima [Kleinschmager et al. 2012], os autores chegaram a conclusões opostas. Eles compararam o desempenho de dois grupos de desenvolvedores em tarefas de manutenção, um utilizado Java, uma linguagem estaticamente tipada, e o outro, Groovy, usado de forma a simular uma versão de Java dinamicamente tipada. Nesse caso, o grupo utilizando Java, a linguagem estaticamente tipada, foi muito mais produtivo. Essa contradição reforça o argumento que os resultados de estudos controlados podem não ser confiáveis para analisar essa questão.

Em experimentos conduzidos em [Daly et al. 2009], os autores comparam o desempenho de dois grupos trabalhando em pequenas tarefas de desenvolvimento. Um grupo utilizou Ruby, uma linguagem dinamicamente tipada, enquanto o outro usou DRuby, uma versão estaticamente tipada de Ruby. Resultados mostraram que o compilador de DRuby raramente conseguiu capturar erros que já não eram evidentes para os programadores. A maior parte dos envolvidos no estudo tinha experiência prévia com Ruby, o que leva a crer que programadores se acostumam com a falta de tipagem estática em suas declarações.

7. Conclusão e Trabalhos Futuros

Este trabalho estuda quais são os fatores mais importantes para a escolha de um sistema de tipos estático ou dinâmico. Existem trabalhos na literatura que analisam as vantagens de cada um através de estudos controlados. Os resultados apresentados aqui, porem, mostram quais são os fatores que de fato influenciam essa decisão através da mineração de um amplo conjunto de repositórios de software e da visualização do uso destes sistemas de tipos.

Quando a necessidade de manutenção e a complexidade de integração entre módulos são questões importantes, tipagem estática aparentemente é preferida por programadores Groovy. Nessas situações a integração com ferramentas de desenvolvimento e a documentação do código oferecidas pelo sistema de tipos estático são vantagens importantes consideradas por programadores. Por outro lado, quando essas questões não são tão críticas, a simplicidade de tipagem dinâmica parece ser preferida, como visto com

declarações de variáveis locais. Outro fator importante é a experiência prévia de programadores com um dado sistema de tipos.

Em trabalhos futuros deseja-se analisar a influência dos sistemas de tipos estático e dinâmico sobre a robustez de sistemas de software. Em particular, deseja-se entender se o uso tipagem dinâmica, que limita a capacidade do compilador em descobrir problemas de tipo, possui alguma correlação com a ocorrência de defeitos no sistema e se o emprego de testes automatizados é capaz de diminuir essa correlação.

Agradecimentos

Este trabalho recebeu apoio financeiro da FAPEMIG, processos APQ-02376-11 e APQ-02532-12, e do CNPq processo 485235/2011-0.

Referências

Tiobe programming community index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. Acessado em 23/06/2013.

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

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 foundations of software engineering*, pages 213–222. ACM.

Cardelli, L. (1996). Type systems. *ACM Comput. Surv.*, 28(1):263–264.

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):13–24.

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.

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):22–35.

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.

Lamport, L. and Paulson, L. C. (1999). Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526.

Mayer, C., Hanenberg, S., Robbes, R., Tanter, É., and Stefik, A. (2012). Static type systems (sometimes) have a positive impact on the usability of undocumented software: An empirical evaluation. *self*, 18:5.

Pierce, B. (2002). *Types and programming languages*. MIT press.

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