

# Robson Castilho

Código de qualidade

## Linguagens estaticamente ou dinamicamente tipadas?

Por quase toda minha carreira, trabalhei com linguagens de tipagem estática. De dois anos para cá, estive quase totalmente focado em linguagens dinâmicas (Ruby e agora Elixir). Tendo já uma boa base prática para comparações, trago neste artigo minha visão sobre ambos estilos de tipagem e o porquê de minha preferência por um deles.

### DEFINIÇÕES

**Tipagem estática** significa que os tipos das variáveis de um programa são explicitamente definidos no código e, portanto, conhecidos/checados em tempo de compilação. Exemplos de linguagens com essa característica: Java, C#, F#, Kotlin, Go, etc.

Na **tipagem dinâmica** é justamente o contrário: os tipos não são declarados no código e, portanto, conhecidos/checados em tempo de execução. Exemplos de linguagens: Ruby, Python, Clojure, Elixir, etc.

**ATENÇÃO:** Não confunda esses conceitos com tipagem forte e fraca. Tipagem dinâmica **NÃO** significa tipagem fraca! **Ruby possui tipagem dinâmica e forte, enquanto JavaScript possui tipagem dinâmica e fraca.**

### VANTAGENS DAS LINGUAGENS ESTATICAMENTE TIPADAS

**Identificação de erros em tempo de desenvolvimento.** A possibilidade de checar os tipos em tempo de desenvolvimento evita uma série de *bugs* em *runtime*, que podem, no pior caso, serem pegos apenas em ambiente de produção. Se seu método recebe um parâmetro do tipo X, é isso que precisa ser fornecido a ele.

Ainda com relação à corretude, é muito simples refatorar o código, por exemplo, alterando os parâmetros de um método ou renomeando uma classe, sem risco de quebrar algo em *runtime*.

**Clareza de código.** Entender o código é fácil, já que não precisamos fazer suposições sobre o que é cada atributo de um objeto e quais são as entradas e saídas esperadas de um método/função. O código se torna a documentação.

**Ferramental.** Graças a uma boa IDE, *refactoring* é trivial. Quanto maior o *codebase*, mais isso é visível: altere o nome de uma classe ou método e a IDE garante que tudo foi devidamente corrigido em todos os arquivos impactados. Ainda temos outras facilidades, como mover arquivos, navegar por eles (*Go To Definition*, *Go to Implementation*, *Find Usages*), *IntelliSense*,

*debugging* e ferramentas para **análise estática** de código.

## VANTAGENS DAS LINGUAGENS DINAMICAMENTE TIPADAS

**Menos verbosidade.** Código mais enxuto, para escrever e para ler.

**Sem espera de compilação.** O ciclo “coda – testa” torna-se imediato. Basta salvar e rodar os testes ou a aplicação.

**Decisões sobre tipos podem ser evitados ou adiadas.** Escrever uma função pode ser mais rápido, sem a obrigação de definir os tipos de sua assinatura. Num primeiro momento, pode ser que você nem tenha certeza sobre qual tipo usar e, numa linguagem estaticamente tipada, você seria obrigado a decidir de imediato, com o risco de ter que mudar no futuro. Outro caso é o uso de *duck typing* para a modelagem de objetos de mesmo comportamento, sem a necessidade de criar um tipo base explícito, como uma classe abstrata ou interface.

**Metaprogramação/monkey patching.** Novamente, rapidez (para se fazer *hacks*, acessar as definições e membros de um objeto podendo alterá-los em *runtime*).

## VANTAGENS MESMO?

Seguindo agora por um lado mais opinativo, vamos analisar as vantagens citadas para as linguagens dinâmicas.

Em se tratando de tempo de compilação, acredito que seja apenas questão de costume, já que houve muita evolução nesse sentido e o compilador é extremamente mais rápido do que nos primórdios do mesmo. A menos que você esteja trabalhando em um legado muito antigo – onde a compilação é penosa – o ciclo de desenvolvimento não é afetado em praticamente nada.

Com relação à menor verbosidade, hoje em dia, as linguagens estaticamente tipadas apresentam **inferência de tipo** – que é habilidade de detecção automática do tipo por parte do compilador – bastante evoluída, resultando em código bem menos verboso do que há 10 anos atrás.

Ainda em relação à verbosidade e às outras vantagens citadas, em softwares corporativos ou qualquer outro com certa complexidade e mudanças frequentes, essas ditas vantagens não se pagam no médio e longo prazo. Com o tempo, fica cada vez mais difícil de entender e manter o software, além de mais comuns os bugs em produção.

Por não ser uma exigência, o uso de tipos personalizados (classes, por exemplo) acaba desestimulado em certas situações. Um exemplo disso é o uso massivo de *hashes* no Ruby, que, embora simplifiquem a codificação, estimulam a violação de camadas, o desuso de termos de negócio (*Ubiquitous Language*) e a dificuldade de compreensão do código.

E, por fim, vale dizer que quanto mais fácil de se fazer algo, mais fácil de ser usado indiscriminadamente, produzindo código de baixa qualidade. E aqui me refiro com mais ênfase à metaprogramação e *monkey patching*, que, independente de linguagem, devem ser usados com parcimônia.

## E OS TESTES?

Outro assunto que aparece bastante quando falamos de tipagens estáticas x dinâmicas são os

testes: escrevemos mais ou menos testes com tipagem dinâmica? Ou não há diferença?

Fato é que quanto melhor o sistema de tipos, menor a chance de termos estados irrepresentáveis no software, simplesmente “de graça”. É impossível, por exemplo, atribuir *null* a um tipo inteiro, em tempo de compilação.

Como garantimos esse tipo de coisa em uma linguagem dinâmica? Resposta: você NÃO garante. Por mais robusta que seja, uma suíte de testes irá validar o correto comportamento do sistema, porém desconsiderando possíveis erros de *type mismatch*.

Já vi erros em produção acontecerem porque um método esperava um objeto “user” com um atributo, digamos “name”, mas recebeu em *runtime* outro “user”, que não possuía este atributo. Isso em um software com ótima cobertura de código (aprox. 90%).

Respondendo as questões iniciais: ou você escreve a mesma quantidade de testes e torce para que tudo fique bem ou você escreve mais testes em linguagens dinâmicas (pelo menos para uma ou outra área mais crítica).

Acredito que “tipagem estática + testes” se complementam e trazem muito mais segurança e robustez. Indo além, a verdadeira base da pirâmide de testes é o compilador.


## CONCLUINDO

Este artigo propôs mostrar as vantagens de cada estilo de tipagem e, por tudo que foi dito, concluo que, para desenvolvimento de aplicações comerciais, que tendam a escalar, de forma segurança, robusta e mais gerenciável o uso de uma linguagem estaticamente tipada é mais apropriada a médio e longo prazo.

As vantagens mencionadas das linguagens estaticamente tipadas superam as vantagens das dinâmicas, que não chegam a ser vantagens tão grandes, principalmente no cenário de softwares com as características acima. Acredito que linguagens de tipagem dinâmica sejam mais adequadas para scripts, demos, MVPs e softwares de escopo menor e mais fechado.

É claro que diversos fatores contam para a seleção de uma linguagem, como o ecossistema, mas procurei manter o artigo focado em seu propósito original, até porque, incluindo ecossistema (*frameworks*, *libs*, comunidades, documentação e suporte), minha escolha seria ainda mais forte por linguagens tipadas.

Qual sua opinião sobre o assunto? Concorda comigo? Discorda? Que pontos seriam cruciais quanto à tipagem dinâmica para escolhê-la? Comente aí!

 16/11/2019

Todos

discussões, linguagens, tipagem dinâmica, tipagem estatica

# 10 comentários em “Linguagens estaticamente ou

# dinamicamente tipadas?”

## 1. dkotvan

disse:

16/11/2019 às 19:03

Show, Robson. Muito bom o artigo.

Depois de vários anos trabalhando com Ruby, e agora trabalhando com Kotlin, a vantagem de uma IDE para fazer os refactorings é algo que faz diferença.

### 1. Robson Castilho

disse:

16/11/2019 às 19:33

Grande, Dimas! Obrigado.

Outra vida agora né? []s

## 2. Regis Hideki Hattori

disse:

16/11/2019 às 20:12

E aí, Robson, blz?

Nessa parte: “Tipagem estática significa que os tipos das variáveis de um programa são explicitamente definidos no código (...)”, acho que a palavra “explicitamente” conflita com o conceito que você citou mais a frente, de inferência de tipos.

Um problema com tipagem estática que me deparei apenas recentemente (talvez por estar acostumado com o fluxo de trabalho de tipagem dinâmica), é que às vezes ela dificulta a adoção de baby steps. Por exemplo, se eu altero uma classe para receber uma dependência a mais, eu preciso alterar todos os lugares que usam esta classe antes de rodar os testes. Mas eu gostaria de poder rodar apenas um testezinho para ver como fica e depois alterar o resto. Mas como o código não compila, isso não é possível.

No mais, estou gostando de voltar a trabalhar com tipagem estática (Kotlin). Mas às vezes ainda bate uma saudade do Ruby haha. Ainda não consegui separar muito bem o quanto dessa “saudade” tem a ver com estar acostumado com a stack e o ferramental (vim + tmux, por exemplo), e quanto tem a ver com o sistema de tipos da linguagem.

### 1. Robson Castilho

disse:

17/11/2019 às 12:53

E ai, Regis!

Cara, acredita que nos primeiros rascunhos tinha um (\*) nessa parte por causa disso!? Depois fui enxugando o texto e acabei retirando! Vou ver como fica melhor.

Nesse quesito do fluxo de trabalho, você tem razão. Particularmente, não acho que quebre o fluxo tanto assim.

Acredito que essa e outras pequenas desvantagens da tipagem se pagam ao longo do

tempo.

Obrigado por comentar!  
[]s

### 3. **Guilherme Barbosa Ferreira**

disse:

18/11/2019 às 11:58

Concordo bastante com seu ponto de vista, acho que valeria uma menção ao TypeScript, que vem sendo umas coisas mais bem sucedidas no ecossistema de JS e tem tudo a ver com o tópico.

#### 1. **Robson Castilho**

disse:

18/11/2019 às 22:47

Com certeza. TypeScript é vida!  
[]s, maninho

### 4. **awilliansd**

disse:

19/11/2019 às 10:09

Muito bom o artigo. Bastante informativo

#### 1. **Robson Castilho**

disse:

19/11/2019 às 21:12

Valeu, camarada. []s

### 5. **Fabício Risetto**

disse:

22/11/2019 às 11:51

Valeu pela menção do post da pirâmide. Concordo com teus pontos. É um assunto bem amplo se for aprofundar, como tu mesmo comentou, em “ecossistema” (libs, frameworks) e ferramental (IDEs, análise de código). Acho que linguagens dinâmicas tem algumas vantagens bem pontuais de “praticidade” como a que o Regis comentou acima, mas no geral também tenho preferência pelas estáticas principalmente para aplicações enterprises ou até mesmo POCs que tenham ambição de um dia evoluir pra algo maior.

Deixando um pouco linguagens de lado, gosto bastante da COMUNIDADE de linguagens dinâmicas e de algumas filosofias que foram disseminadas a partir delas, exemplo: Testes & TDD, o próprio Rails que ajudou a popularizar o padrão MVC.

#### 1. **Robson Castilho**

disse:

25/11/2019 às 23:45

E ainda tem outro ponto que deixei de lado, que é DI. Mesmo a prática tendo literaturas, como no Ruby, não é muito usada pra valer e compor as dependências no boot da aplicação é praticamente impossível porque, em geral, os frameworks não foram criados

levando DI em consideração.

Mesmo as “soluções” existentes – inclusive com uso de container de DI – que vi em Ruby e Elixir parecem não naturais (pra não dizer, gambiarrentas).

[]s e valeu por comentar.

**BLOG NO WORDPRESS.COM.**

**ACIMA ↑**