UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

VANIUS ZAPALOWSKI

# Análise quantitativa e comparativa de linguagens de programação

Trabalho de Graduação

Prof. Dr. Marcelo Soares Pimenta
Orientador

Porto Alegre, julho de 2011

*“Não é o fim que é interessante,*
*mas os meios para lá chegar.”*
— GEORGES BRAQUE

# AGRADECIMENTOS

Foram muitas as pessoas que me ajudaram na realização deste trabalho. Primeiramente e de forma mais enfática devo agradecer aos meus pais. Eles me deram grande apoio e suporte da forma incondicional que só os pais conseguem imaginar em fazer.

Todos que fazem parte do Instituto de Informática, pois ajudaram direta ou indiretamente, me ensinando ou fornecendo ferramentas para que eu pudesse adquirir o conhecimento necessário para execução deste trabalho. Em especial, o professor Marcelo Pimenta que foi diferenciado durante minha jornada acadêmica, contribuindo e influenciando fortemente com seus ensinamentos para a confecção deste trabalho.

Tenho muita gratidão a minha namorada, Paula, que soube entender meus momentos difíceis e complicados, além de ter ajudado no que pode para que este trabalho ficasse como está.

Mais do que colegas de faculdade, tive amigos e companheiros que me ensinaram muita lições durante minha jornada acadêmica, que só tenho a agradecer. Em especial ao Leonardo Borba pela leitura e revisões do trabalho e ao Gabriel Pereira por contribuições no trabalho.

# SUMÁRIO

# RESUMO

Existem mais de 8000 linguagens de programação disponíveis para desenvolvimento, em pouquissímas delas são realizados estudos para obtenção de métricas que forneçam dados relevantes sobre as implementações que nelas foram feitas. Este trabalho visa justamente suprir parcialmente esta lacuna, buscando desenvolver cinco implementações (*benchmarks*) de problemas relevantes, os quais por sua vez farão parte do grupo de execuções das métricas de linhas de código, tempo de processamento e eficiência relativa onde serão aplicadas. A intenção é com isso fornecer dados que poderão ser analisados quantitativamente. A partir de análises feitas sobre os resultados, podem ser extraídas informações necessárias e relevantes para comparar C, C++, Java, PHP, Python e Ruby, que são as linguagens selecionadas neste estudo.

**Palavras-chave:** Linguagens de programação, benchmark, métricas, engenharia de *software*, desempenho.

**Quantitative analysis of programming languages**

# ABSTRACT

There are over 8000 programming languages available for development, but in few of them have works focused to obtain relevant data. Thus, providing metrics on the implementations which were made in them. This paper aims to partially fill this gap, to develop implementations (benchmarks) of relevant problems, which will be part of execution group in which the metrics are applied. The intention is to provide data that can be analyzed quantitatively. From analysis about the results, we can be obtained relevant information needed to compare C, C++, Java, PHP, Python, and Ruby which are the selected languages for this work.

**Keywords:** programming languages,benchmark,metrics,software engineering, performance.

# LISTA DE TABELAS

# LISTA DE FIGURAS

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| ALGOL | Algorithmic Language |
| ENIAC | Electronic Numerical Integrator And Computer |
| Gb | Gigabytes |
| HTML | HyperText Markup Language |
| LDC | Linhas de Código |
| JDK | Java Development Kit |
| JVM | Java Virtual Machine |
| MLDC | Mil Linhas de Código |
| MVC | Model-View-Controller |
| PHP | PHP:Hypertext Preprocessor |
| RAM | Random-Access-Memory |
| RoR | Ruby on Rails |
| UML | Unified Modeling Language |
| YARV | Yet Another Ruby Virtual Machine |

# LISTA DE SÍMBOLOS

| | |
|---|---|
| O | Complexidade Média |
| $\varepsilon_X$ | Eficiência Relativa da linguagem x |
| $\uparrow$ | Iterador de exponenciação |

# 1 INTRODUÇÃO

## 1.1 Motivação

Com o exponencial aumento da capacidade de processamento e armazenamento é comum que desempenho não seja o único fator diferencial nas escolhas do desenvolvimento de *software*. Com a finalidade de se adequar as novas necessidades, são desenvolvidas cada vez mais novas linguagens de programação, tendo como objetivo obter um melhor desenvolvimento das soluções.

Um problema que surgiu a partir da criação dessas tantas maneiras de resolver um problema é saber qual dessas maneiras é a melhor para uma necessidade específica.

Independente de linguagem, é possível gerar muitos tipos de estatísticas para mostrar que o desenvolvimento é o melhor. As duas principais questões que temos de ter claro são como quantificar os dados gerados de modo a podermos comparar uma maneira de desenvolver com a outra, para com isso chegarmos a argumentos quantitativos dessa escolha e podermos, além de tudo, definirmos quais são os dados pertinentes para nossa avaliação quando vamos fazer alguma comparação entre linguagens.

Existem muitas formas de avaliar o processo de desenvolvimento de um *software*. Essas maneiras variam desde uma simples análise do código fonte gerado até uma detalhada verificação do processo de desenvolvimento, e umas das principais é analisarmos o desempenho das linguagens de programação utilizadas.

Cada linguagem tem características muito específicas que fazem com que ela se destaque como a linguagem que tem ou uma manutenabilidade, ou um desempenho, ou um consumo de *hardware* melhor do que suas concorrentes. Além do que, possuem métodos de execução diferentes também.

Tendo em vista essas características e o problema de medição de linguagens, esse trabalho irá trazer informações sobre as linguagens, suas características e as métricas para compará-las.

## 1.2 Contexto Histórico

Linguagens de programação são muito anteriores à existência dos computadores, logo, a definição de linguagens de programação não está ligada diretamente a computadores. As mais antigas são as diversas codificações criadas para minimizar ou codificar comunicação humana, como os exemplos do código Morse e da Pianola.

Existiam diversas codificações e durante toda a história sempre se buscou uma comparação entre elas, visando saber quais eram as melhores. Esse é um objetivo contínuo, ter resultados que possam ser comparados, e que é muito difícil de ser alcançado objetivamente.

As linguagens de programação como conhecemos hoje não foram diferentes da grande maioria dos códigos gerados anteriormente. Na década de 40 ocorreu a aparição do Assembly e seu conjunto de instruções, que foi quando se iniciou oficialmente a busca por menos tempo de processamento, ou diminuir ao máximo o tamanho do código fonte.

No intervalo de tempo entre o grupo de instruções do Assembly até o Java dos dias de hoje, ocorreram mudanças imensas na construção e medição das linguagens de programação. Somado a isso, hoje temos uma quantidade inimaginável na década de 40 de linguagens, por volta de 8000[1]. O objetivo final se manteve o mesmo, porém com metodologias diferentes. Para medições de linguagens existiam escassas ferramentas.

Uma das mais antigas métricas é a medição por linhas de código. Foi introduzida por volta dos anos 60 com o objetivo de medir o custo monetário do *software* originando a medida de dólar por linha de código. Em paralelo a isso, foram criadas as métricas de linhas de código por tempo de trabalho e defeitos de *software*.

Foram dois os principais motivos para essa evolução tão grande das linguagens e das métricas de *software*. A primeira foi a evolução do *hardware*, que ao ser melhorado tornava as máquinas mais poderosas em termos de processamento propiciando a evolução do *software*. A segunda é consequência da primeira. Com mais poder de processamento e armazenamento foi criado o foco em como estruturar e planejar melhor as soluções geradas, que foram as bases da engenharia de *software*.

Por mais que exista a evolução de *hardware*, as avaliações de desempenho e corretude serão sempre necessárias, visto que nossas necessidades estão sempre se modificando de forma a mudar como são desenvolvidos os programas. Métricas tradicionais, como o tamanho do código fonte gerado e o tempo despendido de processamento, se aliam a outras mais modernas, como portabilidade e legibilidade, com o intuito de proporcionar mais meios de analise tornando mais provável a escolha de construções melhores usando esses dados como argumentos.

## 1.3 Objetivo

Para podermos tomar algumas decisões de projeto, precisamos de dados conhecidos. Geralmente esses dados são contribuição de pessoas que já vivenciaram em algum momento uma situação semelhante à atual.

Em uma situação ideal, deveríamos ter os dados de projetos antigos para utilizar como previsões para projetos novos. O que é proposto para esse trabalho é gerar dados a partir de programas exemplos para, em seguida, realizar a análise desses mesmo dados. Assim, buscando as respostas para questões de definições com base em exemplos.

No trabalho serão debatidos os assuntos pertinentes à implementação dos cinco programas com características distintas, as seis linguagens de programação selecionadas presentes na linha de tempos da Figura 1.1 e as três métricas executadas nesse cenário.

A análise não se limitará a simplesmente mostrar os números gerados, mas também discutirá porque esses resultados se comportaram desta maneira.

Basicamente, serão comparadas as linguagens e as métricas escolhidas para este trabalho com a intenção de contribuir com resultados que sirvam como base de argumentação para escolhas futuras.

[1] HOPL: an interactive Roster of Programming Languages - http://hopl.murdoch.edu.au/

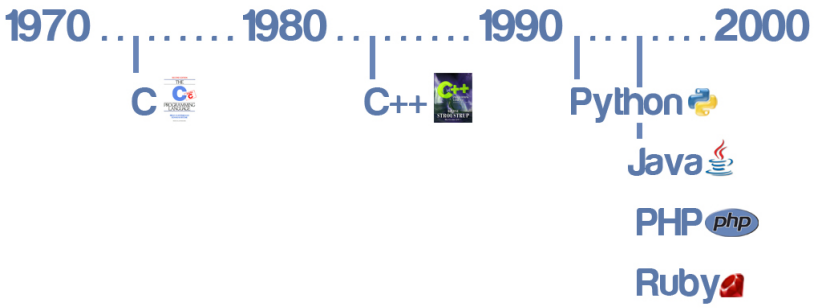


Figura 1.1: Linha do tempo das linguagens do estudo

## 1.4 Estrutura do Documento

Para uma melhor compreensão do presente trabalho, no Capítulo 2 são brevemente comentadas as linguagens de programação selecionadas e suas principais características. Sendo ressaltadas as características que foram diferenciais para a sua escolha

No Capítulo 3, são apresentados e definidos os problemas propostos.

Por sua vez, no Capítulo 4, são detalhadas as métricas alvo do estudo, que são de suma importância para o bom entendimento dos resultados finais.

A aplicação das métricas juntamente dos detalhes de desenvolvimento das soluções dos problemas propostos são descritos no Capítulo 5. Onde são mostrados detalhes de como foram aplicadas as métricas e decisões nas implementações das soluções.

Utilizando as saídas das medições das implementações, no Capítulo 6, são expostos os dados gerados.

Com base nos resultados gerados são feitas as comparações e análises da metologia geral do estudo no Capítulo 7.

# 2 LINGUAGENS DE PROGRAMAÇÃO

Atualmente existem mais de 700 linguagens de programação diferentes, apesar dessa variedade tão grande existe um grupo muito pequeno dessas que são realmente utilizadas.

A escolha das linguagens deste estudo foi dada priorizando as que possuem características diferentes. Características essas que serão citadas logo abaixo nos tópicos individuais de cada uma das linguagens. A necessidade de características diversas nas linguagens se faz necessário pelo fato de que é possível que uma delas tenha uma implementação melhor para um determinado tipo de métrica que será aplicada.

Seguindo essa ideia, para o presente estudo foram escolhidas as seguintes linguagens:

1. C
2. C++
3. Java
4. PHP
5. Python
6. Ruby

Segundo a Tabela 2, todas linguagens escolhidas estão entre as 12 mais populares. Além disso, a soma de todas as popularidades constitui mais de 55 por cento do total.

As linguagens selecionadas também foram selecionadas devido a um prévio conhecimento do desenvolvedor, assim viabilizando o trabalho.

Além de utilizar as linguagens populares, um objetivo é executar os experimentos com o intuito de comparar paradigmas diferentes (HARRISON et al., 1996).

Os principais conceitos abordados neste capítulo são encontrados em (BIANCUZZI; WARDEN, 2009), onde o tema principal é discutir e analisar o projeto e implementação das linguagens, A idéia é conhecer melhor o que trouxe a implementação da linguagem a tona, quem projetou e quando. Isso juntamente de informações técnicas de métodos e características principais faz deste capítulo um bom e breve resumo das linguagens.

Na seções 2.1, 2.2, 2.3, 2.4, 2.5 e 2.6 seguem os detalhes das linguagens que foram utilizadas nos testes práticos.

## 2.1 C

Dentre as mais populares atualmente, C é a mais antiga das linguagens. Surgiu em 1973 dentro dos laboratórios da Bell Telephone com forte influência de ALGOL e B. Foi

| Posição | Linguagem | Popularidade (%) |
|---|---|---|
| 1 | Java | 18.580 |
| 2 | C | 16.278 |
| 3 | C++ | 9.830 |
| 4 | C# | 6.844 |
| 5 | PHP | 6.602 |
| 6 | Basic | 4.727 |
| 7 | Objective-C | 4.437 |
| 8 | Python | 3.899 |
| 9 | Perl | 2.312 |
| 10 | Lua | 2.039 |
| 11 | Javascript | 1.501 |
| 12 | Ruby | 1.484 |

Tabela 2.1: TIOBE Programming Community Index em Junho de 2011(TIOBE, 2011)

desenvolvida e projetada principalmente por Dennis Ritchie e sua equipe (BERGIN JR.; GIBSON JR., 1996) para usuários do sistema operacional Unix.

Por muito tempo C (KERNIGHAN, 1988) foi a linguagem que imperava entre todas as outras, tanto que muitas das linguagens modernas tem como base conceitos e implementações de C. Das selecionadas, C++, Java e PHP são exemplos disso.

As principais características que fazem e fizeram a diferença para o surgimento e continuidade da linguagem são:

- *Paradigma imperativo procedural* - É possível ter uma estruturação dos programas através de procedimentos, assim possibilitando uma organização de código maior comparada às opções da época;

- *Compilação* - Existe a necessidade da utilização de um compilador para traduzir código escrito em C para uma linguagem que a máquina consiga interpretar (SEBESTA, 2003). O uso de compiladores tornou possível a criação de opções de compilação para maior adequação as necessidades do programa.;

- *Tipagem fraca* - Dados os tipos da linguagem, existem tipos que possuem os mesmos comportamentos. Como exemplo pode se citar o uso de caracteres como inteiros, sem necessidade de *typecast*.

- *Suporte a baixo nível* - O acesso total a memória da máquina tira qualquer preocupação da linguagem com o gerenciamento de memória.

Somando essas características foi obtida um linguagem de propósitos gerais, ou seja, é possível construir todas as variedades de programas com ela. Apesar da variedade de possibilidades de programas construídos, ela teve destaque devido ao desenvolvimento de programas que constituem a camada diretamente acima do sistema operacional, assim, criando uma maior abstração no código do que as outras linguagens da mesma época.

Devido as características de ter mais suporte a baixo nível e ser a linguagem mais próxima do núcleo dos sistemas operacionais dentre as utilizadas nesse estudo, são esperados os melhores resultados em relação a desempenho.

## 2.2 C++

Projetada por de Bjarne Stroustrup e sua equipe nos mesmos laboratórios que originaram a linguagem C, nas salas da Bell em New Jersey, porém uma década mais tarde, em 1983, foi criada C++.Ela surgiu da necessidade de novas funcionalidades, paradigmas e conceitos de programação que evoluíram durante a década que se passou.

Como fica perceptível no nome, C++ é fortemente baseada em C, tanto que inicialmente foi batizada de C com Classes, devido ao suporte a orientação a objetos que tinha sido acrescentado ao C para formação dessa nova linguagem. A nova linguagem é muito similar a antiga, contudo possui os novos conceitos (JONES, 2005) que facilitam muito a resolução dos problemas da época.

Os crescentes estudos e necessidades práticas da área de computação na década de 70 fizeram com que algumas características fossem necessárias nas linguagens que surgiriam dali em diante. Novos conceitos foram postos em prática na implementação de C++ (STROUSTRUP, 1994), assim como antigos conceitos foram mantidos e ambos são muito relevantes para os resultados gerados pela linguagem, como comentados a seguir:

- *Multiparadigma* - Com o estabelecimento de orientação a objetos como um bom paradigma de programação, esse foi um dos escolhidos para estar na nova implementação. Um exemplo gráfico de orientação objeto que pode ser implementada em C++ é exibido na Figura 2.1 em forma de UML. Contudo, era possível também gerar código imperativo procedural para essa mesma linguagem. Assim, sendo mantida a compatibilidade com C.

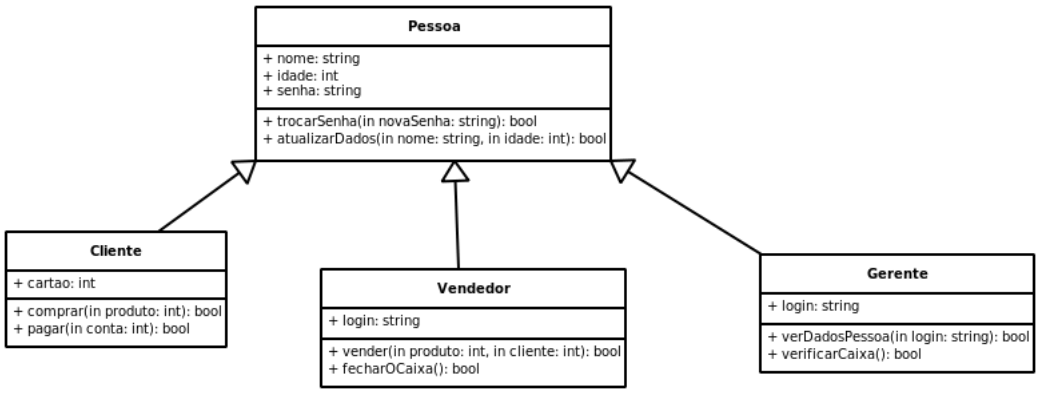


Figura 2.1: Exemplo de orientação a objetos através de UML

- *Compilação* - Existe a necessidade do compilador, assim como explicado na seção 2.1.

- *Tipagem forte* - Uma das novas características é a verificação de tipos, que evita erros de programação, contudo onera a compilação e o desempenho dos programas.

- *Compatibilidade com C* - Todos os programas anteriormente desenvolvidos em C são o melhor motivo para que essa característica seja importante. Essa compatibilidade não é total (STROUSTRUP, 2002), mas são raros os casos que ela não existe.

As aplicações desenvolvidas com C++ são as mais diversas. Devido ao suporte oferecido e a já familiaridade com o estilo de programação herdado do C, C++ se tornou uma linguagem que todas as camadas de aplicações utilizam, tanto o nível mais próximo do núcleo do sistema, quanto a interação final com usuário.Em parte, era isso que faltava para o C ter a possibilidade de gerar código para as camadas de sistema e ao mesmo tempo ter opções de grande abstração para aplicações de alto nível.

## 2.3 Java

Na linha de tempo das linguagens selecionadas, Java está entre as mais novas. Ela foi criada por James Gosling enquanto trabalhava na Sun Microsystems durante os anos anteriores a 1995. Inicialmente foi chamada de Oak por seu criador e em seguida rebatizada de Java, nome originado da lista de palavras randômicas em (RESEARCH, 1999). Seu projeto foi feito visando de ser um componente do núcleo da Sun Microsystems. Hoje em dia é uma das mais populares, como pode ser visto na tabela 2, devido aos fatores que serão abordados a seguir.

Durante o projeto que levou à criação do Java, ficaram claros alguns pontos que deveriam ser seguidos. Logo após o lançamento, Gosling informou, através dos artigos brancos (MICROSYSTEMS, 1997) do Java, os seguintes objetivos do seu projeto:

- *Simples* - O nível de simplicidade que era desejado é que tendo um programador qualquer, não seria necessário um treinamento extenso para começar a produzir soluções desenvolvendo em Java. Os programadores seriam produtivos desde o início da codificação, devido à facilidade de compreensão da linguagem (MICROSYSTEMS, 1997).

- *Orientada a Objetos* - Aplicar os conceitos dessa metodologia de forma eficiente e completa, oferecendo suporte a um sistema totalmente orientado a objetos (DEITEL, 2003).

- *Familiar* - Por sua similaridade com as linguagens já populares, é mais familiar ao usuário. Somando isso ao fato de serem removidas algumas das complexidades que C++ tem.

- *Livre de Arquitetura e Portabilidade* - Com a imensa variedade de *hardwares*, redes, sistemas presentes hoje, dar a liberdade de arquitetura aos projetos é uma das principais características de uma linguagem. Tanto que um dos slogans da Sun sobre o Java é “Escreva uma vez, execute em qualquer lugar.” (MICROSYSTEMS, 1996).

  O compilador Java é diferenciado, ele gera um arquivo *bytecode* o qual é chamado de arquivo de classe, independente de linguagem, que é interpretado pela máquina virtual Java (JVM), como exemplificado no diagrama 2.2.

  Sendo assim, quem se comunica com o sistema operacional é a JVM, que no caso se torna uma camada intermediária entre o *bytecode* e o sistema operacional. Tendo como resultado a portabilidade, pois onde quer que seja executado o código, ele terá sempre os mesmos comportamentos devido ao fato de ter a JVM como intermediário das operações.

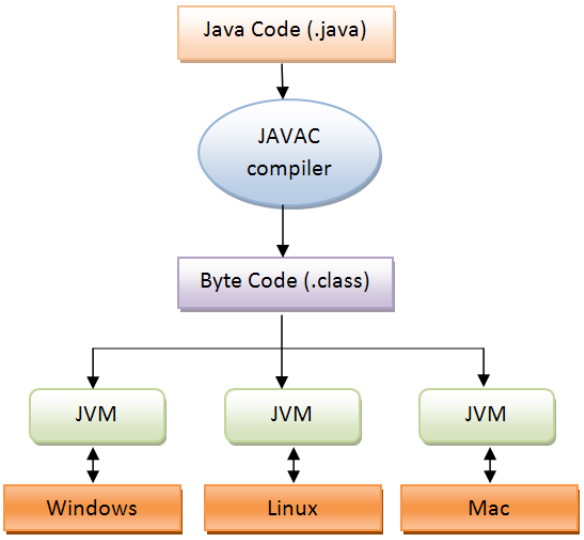


Figura 2.2: Arquitetura do Java com o uso da JVM

- *Compilada e Interpretada* - As linguagem citadas em 2.1 e 2.2 são compiladas e as que serão citadas em 2.4, 2.5, 2.6 são interpretadas. Java é dita híbrida (SEBESTA, 2003) ou mista nesse sentido, pelo fato de ter um código fonte que é sujeito a um compilador para poder ser interpretado pela JVM, como visto na Figura 2.2.

A maior abstração, a grande preocupação com simplicidade e um menor suporte a baixo nível acabaram fazendo com que grande parte das aplicações Java sejam as mais próximas da camada final de desenvolvimento de *software*.

Devido ao fato de Java ser uma linguagem mista, existem estudos que expõem o fato de que usar os dois métodos torna muito custoso em relação ao desempenho do programas (MARTIN, 1997). Existem, por outro lado, estudos que trazem fatores, como o coletor de lixo (W. APPEL, 1987), que apontam para melhorias de desempenho do Java. Espera-se fazer essa verificação através dos resultados e conclusões do presente estudo.

## 2.4 PHP

No mesmo ano em que Java surgiu, Rasmus Lerdorf considerou finalizada a primeira versão de sua linguagem, o PHP. Originalmente PHP era a sigla para “Personal Home Page”, mas hoje em dia o significado é outro, “PHP : Hypertext Preprocessor”, fazendo referência ao próprio nome antigo (THE PHP, 2011).

PHP foi desenvolvida com o intuito de ser uma linguagem de *script web* que fosse adicionada como um módulo a um servidor para gerar páginas *web* dinamicamente. O interpretador inclui código dentro do código HTML vindo do cliente e responde com um documento HTML já interpretado. Atualmente esse processo é realizado em mais de um milhão de servidores e com mais de 20 milhões de domínios de endereços na Internet (THE PHP, 2007).

A seguir são brevemente comentadas as principais características da PHP:

- *Interpretada* - Ser uma linguagem interpretada reflete diretamente no desempenho da linguagem. O código gerado pelo desenvolvedor é interpretado para linguagem de máquina somente no momento em que ele é atingido em tempo de execução.

Assim, não se tem o tempo de compilação para geração do programa e também não existe o tempo de recompilação, caso sejam feitas alterações sobre o código, mas existe sempre o tempo de tradução do interpretador para linguagem de máquina (SEBESTA, 2003).

- *Multiparadigma* - O comportamento multiparadigma usado em C++, como citado na seção 2.2. Sendo assim, é possível utilizar orientação a objetos, procedimentos e características funcionais em um mesmo contexto de programa PHP (DALL'OGLIO, 2009).

- *Padrão "de facto"* - Ser um padrão *"de facto"* significa que não existe uma especificação que rege todas as implementações de PHP, de forma a validar cada uma delas. Isso tem como consequência a possibilidade da existência de diferenças de quaisquer natureza entre cada implementação de preprocessador. Essa classificação foi dada porque existe um uso muito grande da linguagem, embora não exista uma padronização formal dela.

Como se pode ver na Tabela 2, PHP é a linguagem mais popular das puramente interpretadas. Isso em grande parte se dá devido aos pontos citados sobre ela anteriormente.

Somado às características citadas, existe o fator de ela ter sido projetada para gerar um documento HTML, que na maioria dos casos resulta em uma página web, podendo ser visualizada em qualquer sistema operacional e qualquer arquitetura desde que se tenha um navegador de Internet. Esse proposito foi de suma importância para a sua grande popularidade, pois com o aparecimento da Internet no fim da década de 90 para o grande público existia a necessidade de uma forma de gerar conteúdo dinâmico para Internet e em muitos casos PHP foi a escolhida codificação dessas páginas.

## 2.5 Python

Na tentativa de Guido van Rossum ter uma linguagem de programação muito simples e legível, foi lançada, em 1991, Python. Diferentemente de todas as linguagens anteriormente citadas que foram criadas com forte inspiração em C, Python é fortemente baseada em uma linguagem chamada ABC. Guido teve contato com essa linguagem que visava o aprendizado em programação, pois não havia necessidade de declaração de variáveis e era obrigatória a identação para blocos de código aninhados. Conceitos esses que são dos principais na linguagem de Guido (TELLES, 2008).

Um dos principais conceitos que foram levados em conta no projeto da Python é tornar o código dela legível. Para isso o núcleo do projeto era minimalista, com o objetivo de criar um base simples e dar um grande suporte através de bibliotecas. Junto desse núcleo básico foram pensadas palavras chave com significado, assim alcançando uma maior legibilidade.

A identação obrigatória, as palavras chave da linguagem com sentido e um núcleo simples que seu criador tinha mente. Com isso, Python ficou com as seguintes principais características:

- *Interpretada* - Os comentários feitos na Seção 2.4 sobre esse tópico são válidos aqui também.

- *Legibilidade* - Ponto tido como fundamental no desenvolvimento da linguagem era ter uma linguagem legível. Isso facilita muito o aprendizado e o desenvolvimento dos programas, pois tenta aproximar a codificação da linguagem natural.

- *Multiparadigma* - Dar liberdade ao desenvolvedor escolher o paradigma que quisesse é uma realidade da programação em Python. Existe o suporte a programação orientada a objetos, imperativa e funcional.

Hoje em dia Python é muito conhecida no desenvolvimento web devido aos novos *frameworks*. Django (THIAGO GALSEI, 2010) é um dos principais *frameworks* com muitos recursos e uso do padrão *Model-View-Controller*(MVC)(GAMMA et al., 2000).

## 2.6 Ruby

"Eu queria uma linguagem de script que fosse mais poderosa do que Perl e mais orientada a objetos do que Python. Por isso decidi projetar minha própria linguagem" (STEWART, 2001) essas foram as palavras de Yukihiro Matsumoto quando perguntado sobre porque teria decidido escrever sua própria linguagem. Essa decisão foi tomada no início da década de 90 e em 1995 estava lançada a primeira versão de Ruby.

O objetivo principal da linguagem é fornecer aos programadores uma alternativa que faça com que eles sejam mais produtivos e gostem de programar, assim como Python propôs anos antes.

As principais características são similares às de Python, porém existem alguns detalhes que devem ser apresentados, como :

- *Multiparadigma* - Os paradigmas são os mesmo de Python, descritos na Seção 2.5, contudo as principais influências diferem. A maior influência na orientação a objetos de Ruby é SmallTalk, a pioneira nesse paradigma, onde não existe tipos de dados que não são objetos. Por exemplo, não existem inteiros, booleanos e caracteres, ao invés disso são objetos inteiros, objetos booleanos e objetos caracteres.

- *Interpretada* - Os interpretadores se diferenciam por serem de uma ou de múltiplas passadas. Isso implica diferenças na ordem do código e no desempenho da linguagem, pois o interpretador de passada única lê serialmente o código. Sendo assim, todas referências utilizadas já devem ter sido lidas e criadas pelo interpretador. Ruby, na maioria de suas implementações, faz uso do interpretador de passada única, ao contrário de PHP que possui interpretador de múltiplas passadas.

- *Padrão "de facto"* - A exemplo de Python, Ruby possui esse tipo de padrão. Por causa disso, Ruby tem uma variedade grande de implementações, além da padrão implementada em C. Sendo as principais delas YARV ( *Yet Another Ruby VM* ) (OLSEN, 2011) e JRuby (Java Ruby) (EDELSON; LIU, 2008), onde são usados métodos alternativos de tradução para linguagem de máquina.

O sucesso e crescente popularidade de Ruby estão fortemente ligados ao crescimento de Ruby On Rails (RoR), seu principal framework amplamente conhecido (AKITA, 2006) para aplicações *web*.

Como se pode notar pelo já citado neste trabalho sobre Ruby e Python, existem muitos pontos em comum entre elas e igualmente justificativas similares.

# 3 PROBLEMAS PROPOSTOS

A decisão de quais problemas abordar é de suma importância para o presente trabalho, pelo simples fato de eles serem a construção que será testada e medida. Assim, influenciando fortemente nos resultados finais e conclusões obtidos.

Todas as linguagens são de propósito geral, sendo assim, para a maioria dos problemas, é possível criar soluções equivalentes, desde que exista o suporte na determinada linguagem para tal resolução.

Tendo em vista evitar qualquer benefício em virtude de características de determinada linguagem devido ao fato de ela ter uma implementação destinada a um nicho específico de aplicação, foram propostos problemas variados, dos mais simples aos mais complexos. Como consequência disso, são gerados cenários diferentes, nos quais são exploradas diferentes características.

Nas seções seguintes desse capítulo serão apresentados e detalhados os cinco problemas propostos.

## 3.1 Olá Mundo

É comum quando queremos aprender uma linguagem buscarmos por programas de exemplo para termos uma base de como é a linguagem e sua sintaxe. Para praticamente todas as linguagens, quando fazemos essa busca, nos é apresentado o programa Olá Mundo, pois esse é um problema que nos apresenta brevemente conceitos da linguagem sem a necessidade de estudamos a fundo sobre ela.

Mesmo que ele seja um problema que não é solução para projetos comerciais, ele é muito relevante para o presente estudo.

O objetivo desse problema é que, ao ser executado, ele nos retorne na tela a frase “Ola Mundo”, e seja finalizado com sucesso.

Por ser um problema que não tem variáveis de entrada, nem variáveis internas, somente a execução de um comando de exibir na tela, sua complexidade é constante.

A primeira vista esse programa não acrescenta muita informação, porém ele nos diz muito sobre a linguagem que está sendo trabalhada. Com diferentes paradigmas, formas de tradução para linguagem de máquina e características do código fonte, esse problema, que idealmente é para ser simples, fornece as informações citadas de forma direta e rápida ao desenvolvedor.

Neste problema será revelada principalmente a simplicidade de cada linguagem, pois, se é um problema simples, deve ser solucionado de forma também simples.

## 3.2 Laços Encadeados

Na linguagens modernas existe uma grande variedade de iteradores, com diferentes sintaxes e objetivos. Independente da escolha de linguagem de programação que é feita, é básico o uso de iteradores. Usando boas práticas de programação, é impossível desenvolver sem utilizar iteradores nas linguagens modernas.

Ao contrário do Problema 3.1, Laços Encadeados faz uma imensa diferença nas aplicações do cotidiano considerando um paradigma imperativo. A otimização da implementação desse tipo de comando é tida como uma das prioridades durante o desenvolvimento do núcleo da linguagem. Isso acontece porque cada nanosegundo ganho por iteração pode variar bastante o tempo de execução do programa.

O programa que resolve o problema desta seção deve ser capaz de receber um número inteiro como argumento e exibir na tela o número de iterações realizadas. Esse argumento do programa é o índice dos seis laços encadeados, de forma a gerar uma complexidade como a exibida na Figura 3.1.

$$\begin{aligned} Complexidade(Lacos) &= n*n*n*n*n*n \\ &= n^6 \\ &= O(n^6) \end{aligned}$$

Figura 3.1: Complexidade do problema Laços Encadeados

Embora seja sempre executada uma operação matemática simples de soma, o uso de processamento é grande visto que o número de execuções cresce sensivelmente conforme o aumento do argumento de entrada, como mostrado na Figura 3.1.

## 3.3 Função Ackermann

Uma das funções importantes na teoria da ciência da computação é a função de Ackermann (SUNDBLAD, 1971). A função tem um crescimento incrivelmente rápido, de forma que existe até uma notação específica criada para representação da função, os números de Ackermann.

A definição da função é regida pelas regras presentes na Figura 3.2, que é uma simplificação da função Ackermann de três argumentos (SUNDBLAD, 1971). Assim, a complexidade da função utilizando notação de Knuth (DEPT; KNUTH, 1976) é apresentada na Figura 3.3. Seguindo a definição da função, o programa que resolve o problema proposto nesta seção deve imprimir o resultado da função de Ackermann simplificada corretamente.

$$Ack(m,n) = \begin{cases} n+1 & \text{Se } m = 0 \\ Ack(m-1,1) & \text{Se } m > 0 \text{ e } n = 0 \\ Ack(m-1,Ack(m,n-1)) & \text{Se } m > 0 \text{ e } n > 0 \end{cases}$$

Figura 3.2: Definição da Função Ackermann

A função Ackermann é usualmente aplicada em testes de desempenho de chamada de procedimentos e métodos devido ao cálculo da função ser recursivo em relação às suas

$$\begin{aligned} Complexidade(Ack) &= 2 \uparrow^{m-2} (n+3) - 3 \\ &= O(2 \uparrow^{m-2} (n+3)) \end{aligned}$$

Figura 3.3: Complexidade da Função Ackermann

entradas. Obrigatoriamente para que esse nos de resultados relevantes para o estudo, não se pode usar outros métodos para resolução do problema a não ser recursividade, pois com o uso de métodos como programação dinâmica obteremos resultados melhores em relação a tempo de processamento, que não refletem o desempenho real das chamadas realizadas pela linguagem.

## 3.4 Cliente-Servidor de Echo

Amplamente usada e conhecida hoje em dia no contexto da Internet, a arquitetura Cliente-Servidor tem como propósito fazer a comunicação entre dois pontos. Invariavelmente utilizamos essa arquitetura no nosso cotidiano, seja ao realizar uma transação bancária, ao estabelecer uma conexão com a Internet, etc.

O comportamento padrão na arquitetura padrão de Cliente-Servidor possui quatro passos (COMER, 2007), que são eles:

1. O servidor é ligado e espera contato do cliente;
2. O cliente através da localização do servidor requisita um conexão com ele;
3. Estabelecida a conexão, trocam as mensagens necessárias;
4. Depois do encerramento de envio de mensagens, é enviado um sinal de fim de transmissão e a conexão é encerrada.

O problema aqui proposto segue as etapas citadas nesta seção, mas possui as particularidades de especificação:

1. O servidor e o cliente são rodados na mesma máquina, porém por processos ou threads diferentes;
2. Será transmitida do cliente para o servidor a mensagem “Ola Mundo”, a qual possuí 10 Bytes de tamanho;
3. O programa deve permitir como parâmetro um inteiro, cujo proposito é definir o número de vezes que a mensagem será transmitida.

Só existe uma única variável no problema que é a de número de envios de mensagens. Essa variação aumenta linearmente a quantidade de informação enviada, então se considerarmos n como número de envios e 10 o número de Bytes enviados, temos uma complexidade do problema como vista na Figura 3.4.

Esse problema tem foco na implementação de comunicação entre aplicações, assim sendo de grande relevância no resultado de desempenho das linguagens.

$$\begin{aligned} Complexidade(Echo) &= 10 * n \\ &= O(n) \end{aligned}$$

Figura 3.4: Complexidade do problema Cliente-Servidor de Echo

## 3.5 Contador de Frequência de Palavras

O Contador de Frequência de Palavras é um problema simples. Ele tem por objetivo verificar a repetição de palavras de um texto e informar essa repetição de forma ordenada para o usuário.

O programa capaz de resolver esse problema deve receber como argumento um texto e retornar a lista de palavras na ordem de frequência de ocorrência no texto.

Pensando na complexidade do problema, a principal variável é a quantidade de palavras do texto de entrada. Se for considerada a quantidade de palavras do texto como n, para desenvolver um programa que atenda a especificação deve-se verificar palavra por palavra do texto, o que é um custo linear sobre n, depois realizar uma ordenação sobre esse mesmo conjunto de palavras e por fim mostrar ao usuário o resultado. Seguindo essas instruções chega-se a complexidade do problema como mostrada na Figura 3.5.

$$\begin{aligned} Complexidade(Freq) &= n + n * \log n \\ &= O(n * \log n) \end{aligned}$$

Figura 3.5: Complexidade do problema Contador de Frequência de Palavras

A principal característica desse problema é a grande utilização dos comandos de entrada e saída da linguagem, visto que pode-se entrar com um texto com milhares de palavras, o que fará com que seja retornado também milhares de palavras, caso elas sejam todas distintas entre si.

# 4 MÉTRICAS DE LINGUAGENS

Existem muitas discussões entre desenvolvedores sobre qual das linguagens de programação é a melhor. Cada uma delas é defendida exaltando seus melhores pontos, mas como realmente ela deve ser avaliada é o questionamento que foi feito quando foram analisadas as métricas de linguagens.

O crescente número de linguagens gera a necessidade da comparação entre elas. As métricas são geralmente associadas aos programas gerados pelas linguagens, sendo o foco criar otimizações e melhorias dentro da aplicação em busca de um melhor resultado. Contudo, o objetivo deste trabalho é gerar dados quantitativos sobre as linguagens e não sobre aplicações implementadas nela.

Visto que o resultado produzido pela linguagem é gerado a partir do código desse programa, um dos pontos principais a serem analisados é como são codificados os problemas. Partindo disso, é necessário garantir uma equivalência básica na codificação e uma estruturação tão similar quanto possível entre as implementações, mas também respeitando as características da linguagem.

Muitas definições de projetos dependem fortemente de como se comporta a implementação gerada pelo desenvolvedor, que por consequência depende da implementação da linguagem. Então, quanto mais dados relevantes sobre qualquer uma das camadas de produção de *software*, melhores e mais precisas serão as especificações.

Nas seções 4.1, 4.2 e 4.3 serão abordadas as métricas as quais são apresentados os resultados no Capítulo 6.

## 4.1 Linhas de Código

As primeiras notícias de métricas de *software* vem por volta de 1960 quando houve a necessidade de se medir a produtividade, qualidade e custo do desenvolvimento de *software*. Produtividade era medida em linhas de código (LDC) por unidade de tempo. Qualidade era medida em defeitos por MLDC, onde “M” significa mil na sigla. O custo era medido em dólares por LDC (JONES, 2009).

Medir os programas desenvolvidos por tamanho do código fonte é muito simples, basta ter regras de padronização de desenvolvimento para todas as linguagens e verificar quantas linhas de código escritas os programas possuem. A variação de resultados na aplicação de LDC sobre soluções pode variar de acordo com as padronizações de desenvolvimento escolhida para cada linguagem, por exemplo, usar símbolos de início e fim de blocos sempre que possível, ou não usar sempre que possível.

Com o passar do tempo essa métrica acabou perdendo seus objetivos iniciais, visto que as linguagens acabaram mudando e evoluindo. Como proposto em (JONES, 2009), supondo um problema implementado em linguagens diferentes, Assembly e Java, temos

parte dos resultados retirados de (JONES, 2009), apresentados na Tabela 4.1.

| Linguagem | Assembly | Java | Diferença |
|---|---|---|---|
| Linhas de Código | 1000 | 200 | -800 |
| Custo | $5,000.00 | $5,000.00 | $0.00 |
| Tempo(Meses) | 1 | 0.25 | -0.75 |
| LDC por mês | 1000 | 800 | -200 |
| Custo de LDC | $10.00 | $31.25 | $21.25 |

Tabela 4.1: Comparativo de linguagens de baixo e alto nível

Como se pode notar na Tabela 4.1, a linguagem de alto nível tem um custo maior por LDC, mas são necessárias menos LDCs para codificar o mesmo problema, além de levar menos tempo. Sendo assim, o fator linguagem impacta de maneira significante sobre os resultados.

Uma consideração importante a ser feita sobre LDC é que ela avalia a simplicidade da linguagem. Ela não avalia diretamente o conteúdo do código fonte e os detalhes de programação, mas, para maioria dos casos, resolver um problema com menos passos é uma resolução mais abstrata, logo, mais simples e compreensível.

## 4.2 Tempo de Execução

Tempo de execução é um parâmetro chave no desenvolvimento e projeto de *software*, pois podem existir limites mínimos ou máximos de tempo que a aplicação deve executar para atender seu propósito. Por isso, essa métrica é tão importante e relevante.

Por muito tempo foi a principal métrica, visto as restrições de *hardware* que existiam. Hoje em dia essa dificuldade existe, mas não com a mesma intensidade que em décadas passadas, porque a tecnologia se desenvolveu de forma a nos fornecer uma infraestrutura melhor e mais barata.

Em um projeto de *software* é comum que ao ser abordado a melhoria de tempo de execução se pense em otimizações de código, contudo a escolha de uma linguagem que faz uma implementação mais eficiente pode reduzir muito o tempo de otimização que já são feitas nativamente por outra linguagem.

Buscando esses dados que trazem a diferença entre as linguagens foi proposto em (KENNEDY; KOELBEL; SCHREIBER, 2004) que a solução de um problema $P$, é dito que $T(P)$ o tempo total para solucionar $P$ . Sendo proposta a equação na Figura 4.1 retirada de (KENNEDY; KOELBEL; SCHREIBER, 2004). Considerando $E$ o tempo médio de execuções de uma solução do problema, $I$ o tempo de implementação do problema e $r$ um coeficiente de ajuste de importância entre o tempo de implementação e o tempo de execução.

$$T(P) \quad = I(P) + rE(P)$$

Figura 4.1: Tempo para solucionar um problema $P$.

Como resultado da aplicação da métrica citada na Figura 4.1, temos uma medida de produtividade, visto que envolve o tempo de desenvolvimento. Pela dificuldade de

medição do tempo de desenvolvimento das soluções, será analisada neste trabalho somente o tempo médio de execução da solução do problema, assim excluindo o tempo de desenvolvimento e o coeficiente de ajuste, torna-se uma métrica de desempenho.

Medir o tempo de execução implica em analisar também alguns fatores que influenciam os resultados de desempenho. Deve-se considerar uma mesma arquitetura, pois comparar linguagens que executam em máquinas com recursos de processamento diferentes levará a resultados diferentes. Comparar implementações que respeitem as características das linguagens, porque para que as soluções reflitam o desempenho real da linguagem deve-se considerar a aptidão do desenvolvedor para a determinada linguagem.

## 4.3 Eficiência Relativa

Partindo da métrica citada na Seção 4.2, o objetivo da Eficiência Relativa é criar um parâmetro de comparação, que por sua vez gera uma estimativa de tempo de execução da implementação em uma linguagem nova. Isso é possível através da utilização de um histórico de implementações. Sendo assim, não é necessário implementar o problema para ter uma boa estimativa de qual será o tempo de execução dela.

Usando a mesma notação da Seção 4.2, temos que a Eficiência Relativa é a divisão entre o tempo médio de execução de uma linguagem sobre outra (KENNEDY; KOELBEL; SCHREIBER, 2004), definido como como mostrado na Figura 4.2. Assim temos a eficiência relativa de uma linguagem em relação a outra.

$$\varepsilon_L = \frac{E(P_0)}{E(P_L)}$$

Figura 4.2: Definição de eficiência relativa do problema $P$.

Geralmente temos como regra que analisando duas linguagens, uma de baixo nível e outra de alto nível, teremos um $\varepsilon_L < 1$. Isso nos mostra o quanto uma maior abstração custa ao desempenho.

Assim, espera-se que com um conjunto razoável de problemas resolvidos em ambas as linguagens e aplicada a métrica de eficiência relativa teremos uma estimativa muito próxima da real para o tempo de execução de programas ainda não implementados em uma das linguagens. Por isso, pode-se dizer que a comparação entre o tempo de execução de duas linguagens depende somente da implementação de uma delas e um conjunto prévio de soluções de ambas, como mostrado na Figura 4.3.

$$\begin{aligned} E(P_L) &= E(P_0) * \frac{E(P_L)}{E(P_0)} \\ &= \frac{1}{\varepsilon_L} * E(P_0) \end{aligned}$$

Figura 4.3: Definição do tempo de execução em relação a eficiência relativa e implementação já realizada.

A grande questão sobre essa métrica é como garantir que a quantidade de problemas que fazem parte da $\varepsilon$ são suficientes para estimar o resultado em relação a outra linguagem. Isso é solucionado usando tantos programas quanto possível, pois utilizando a média entre eles distribuímos qualquer disparidade entre os programas implementados, assim tornando a métrica mais precisa.

Com o crescente número de repositórios *online* de *software* como, Github[1], SourceForge[2] e GoogleCode[3], existem vários códigos fonte a disposição para serem comparados.

[1] http://www.github.com
[2] http://sourceforge.net
[3] http://code.google.com

# 5 IMPLEMENTAÇÕES

Existem detalhes da implementação que devem ser comentador e especificados para uma compreensão adequada dos resultados que seguem no Capítulo 6. Assim, nas seções deste capítulo seguem esses detalhes.

## 5.1 Arquitetura

Como a infraestrutura utilizada influencia nos resultados gerados ela deve ser bem especificada. Sendo assim, necessária a exposição de cada um dos meios de tradução utilizados por cada linguagem.

Na Tabela 5.1 é mostrada a configuração da máquina na qual foram executados os testes.

| | |
|---|---|
| Processador | Intel E8500 |
| Memória RAM | 4 Gb DDR2 |
| Sistema Operacional | Kernel Linux 2.6.32-generic |

Tabela 5.1: Dados da máquina utilizado nos testes.

Na Tabela 5.2 são exibidos os compiladores e interpretadores de cada linguagem junto de suas respectivas versões.

| **Linguagem** | **Compilador / Interpretador** | **Versão** |
|---|---|---|
| C | GCC | 4.3.3 |
| C++ | GCC | 4.3.3 |
| Java | javac/OpenJDK Runtime Environment | 1.6.0-24/IcedTea6 1.9.7 |
| PHP | PHP | 5.3.2 |
| Python | Python | 2.6.5 |
| Ruby | Ruby | 1.9.2p136 |

Tabela 5.2: Versões dos compiladores e interpretadores.

## 5.2 Implementações dos Problemas

Os problemas foram implementados usando como práticas de programação os livros de referência de cada linguagem. As principais referências para cada linguagem são citadas abaixo.

1. C, (KERNIGHAN, 1988);
2. C++, (DEITEL, 2001);
3. Java , (DEITEL, 2003);
4. PHP, (HOLZNER, 2007);
5. Python, (THIAGO GALSEI, 2010);
6. Ruby, (FLANAGAN; MATSUM, 2009).

Sendo todos programas desenvolvidos para rodar através de linha de comando, sem interface gráfica.

## 5.3 Implementações das Métricas

O meio que as métricas são implementadas são relevantes e influenciam nos resultados. Por isso, abordadas nas subseções desta seção.

### 5.3.1 Linhas de Código

As linhas de código foram simplesmente contadas de forma manual. Sendo consideradas mesmo linhas em branco e comentários, visto que essas também fazem parte do programa.

### 5.3.2 Tempo de Execução

Foi desenvolvido um script que roda todos os resultados com as entradas e número de repetições escolhidas. Sendo assim, todos os programas foram executados 1000 vezes com os parâmetros que seguem na Tabela 5.3.

| **Programa** | **Parâmetro** |
|---|---|
| Ackerman | 9 |
| Cliente-Servidor de Echo | 10000 |
| Contador de Frequência de Palavras | texto com 30000 palavras |
| Laços Encadeados | 25 |
| Olá Mundo | |

Tabela 5.3: Parâmetros das implementações

### 5.3.3 Eficiência Relativa

Utilizando os dados gerados a partir da forma de implementação citada na Subseção 5.3.2, Assim, foram realizados os cálculos para obtenção da eficiência relativa de todas as linguagens sobre a linguagem C, escolhida por ser a de mais baixo nível dentre todas as utilizadas no estudo.

# 6 RESULTADOS

O resultados gerados serão apresentados neste capítulo. A maioria deles está em forma de gráfico para um melhor entendimento, sendo que quando houve necessidade de detalhamento foram expostas de uma melhor maneira.

Sendo todas as implementações juntamente de seus *scripts* de medição e resultados gerados estão disponíveis em (ZAPALOWSKI, 2011).

## 6.1 Linhas de Código

### 6.1.1 Ackermann

Para a resolução deste problema, não houve uma variação muito grande no número de linhas dos programas, como se pode ver no Gráfico 6.1.

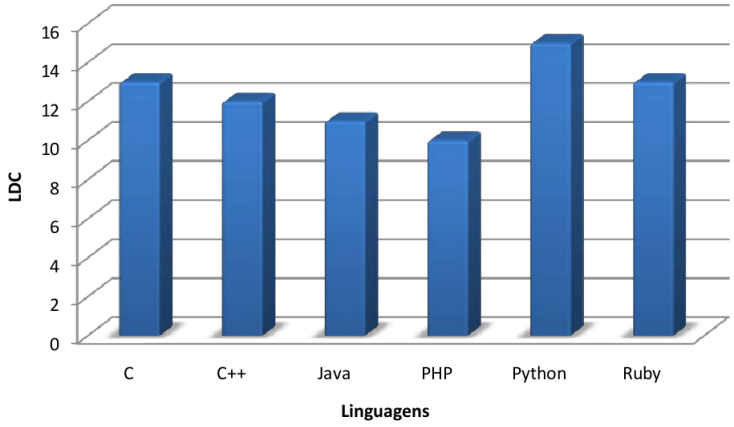


Figura 6.1: Linhas de código por implementação em cada linguagem do programa Ackermann

O Gráfico 6.1 reflete a semelhança entre as sintaxes das linguagens, pois o problema é estruturado de maneira muito semelhante em todas elas, de forma a ser facilmente implementado e entendido independentemente da linguagem dentre as selecionadas para o estudo.

Além da sintaxe similar, o fator de o problema ser simples tem papel diferencial no resultado. O problema não exige uma estrutura muito complexa para sua resolução, com isso não tem muitas variações de resolução em diferentes paradigmas.

### 6.1.2 Cliente-Servidor de Echo

Neste problema já surge com mais clareza como a linguagem de alto nível tem vantagem em relação a linguagem de mais baixo nível, nesta métrica.

Analisando o Gráfico 6.2 podemos ver que as três linguagens de alto tem uma resolução com um código sensivelmente menor que as duas outras de mais baixo nível. É visível a decrescente em relação ao número de linhas escritas

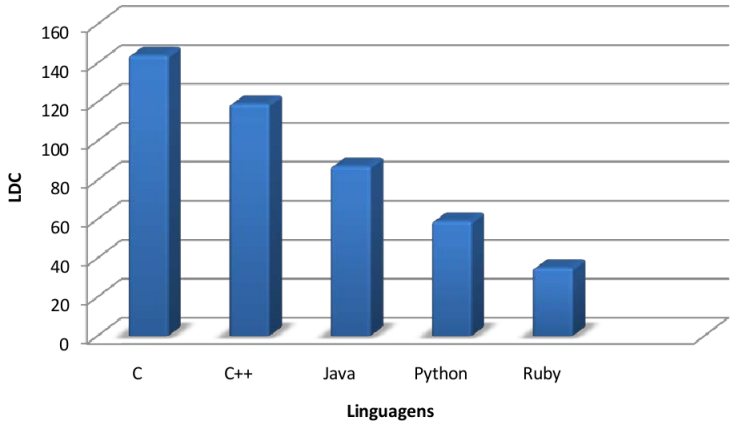


Figura 6.2: Linhas de código por implementação em cada linguagem do programa Cliente-Servidor de Echo

A complexidade do problema foi o fator que mais influenciou o resultado, pois as implementações seguem os passos citados na Seção 3.4. Contudo, conforme a crescente do nível de abstração, as linguagens fornencem mais funções nativas que facilitam a programação, tornando assim as soluções menores.

Houve uma empecilho na implementação do problema na linguagem PHP. Visto que ela não tem suporte a criação de processos, não possui a primitiva *fork*, ficando assim comprometida a comparação dela, pelo fato de não possuir construções semelhantes na linguagem. Por esse motivo, ela não consta no Gráfico 6.2.

### 6.1.3 Contador de Frequência de Palavras

Na resolução deste problema se pode constatar pelo Gráfico 6.3 um comportamento semelhante ao da implementação da Subseção 6.2, a exceção de C++ em relação a C. As linguagens de mais alto nível possuem uma codificação menor que as de mais baixo nível.

O que ocorreu com C++ foi que com a utilização de orientação a objetos o código acabou se tornando mais extenso, devido ao fato da necessidade de declaração e codificação das classes e métodos, que possuem praticamente as mesma operações que os código em C, mas de uma forma mais modular.

### 6.1.4 Laços encadeados

Um programa simples, porém que resultou em uma pequena variação no número de linhas codificadas, não refletindo o nível de abstração em relação ao tamanho do fonte, como pode ser visto no Gráfico 6.4.

As implementações não se diferenciam muito uma da outra porque as sintaxes dos comandos de iteração são muito similiares entre elas. A variação que existe no número de linhas é devido a sintaxe obrigatória referente a cada linguagem, como declaração da

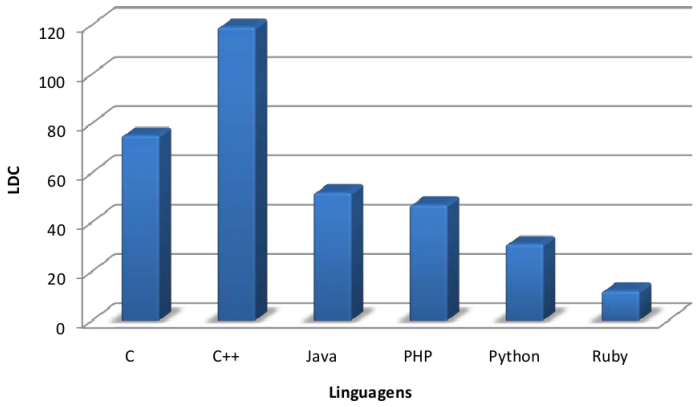


Figura 6.3: Linhas de código por implementação em cada linguagem do programa Contador de Frequência de Palavras

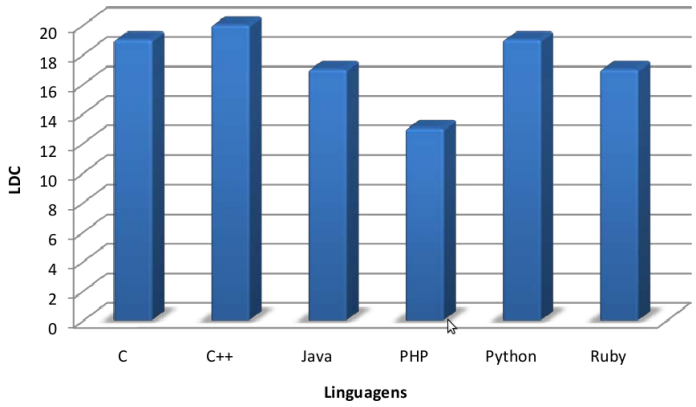


Figura 6.4: Linhas de código por implementação em cada linguagem do programa Laços Encadeados

função *main*, fechamento de bloco de comandos e declaração de variáveis.

### 6.1.5 Olá Mundo

O problema mais simples dos implementados mostra principalmente a diferença entre as linguagem que usam um compilador para as que usam um interpretador, como pode ser visto no Gráfico 6.5. Sendo que Java, que faz uso de ambos, tem um comportamento mais próximo das linguagens compiladas.

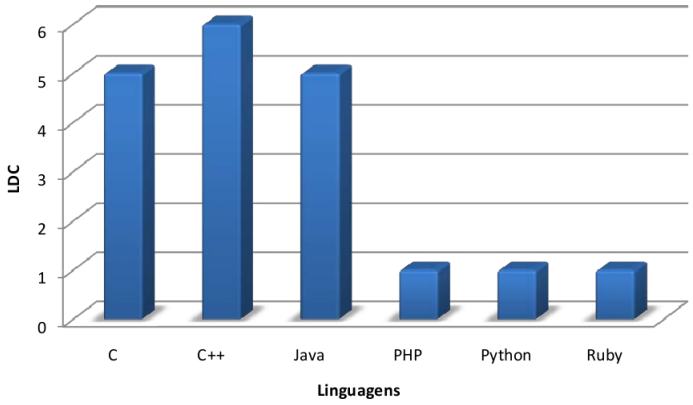


Figura 6.5: Linhas de código por implementação em cada linguagem do programa Olá Mundo

A sintaxe necessária para a compilação faz com que as soluções quando codificadas tenham mais linhas, ao contrários das linguagens interpretadas que fazem somente a chamada a função de imprimir a frase na tela. Isso, simplifica bastante o programa construído.

## 6.2 Tempo de Processamento

Nesta seção seguem os resultados gerados pelas medidas dos tempos de processamento de cada um dos problemas. Sendo os gráficos com os tempos de processamento em modo sistema, usuário e o tempo real. O tempo real de processamento não é a soma direta do tempo de processamento em modo de usuário e de sistema porque o ambiente possui um processador com múltiplos núcleos.

### 6.2.1 Ackermann

Através do Gráficos 6.6, que foram separados para uma melhor visualização dos dados, percebe-se uma variação enorme entre os tempos das implementações. O tempo de PHP comparado ao de C tem um diferença de aproximadamente 350 vezes.

Claramente o desempenho das linguagens que usam compiladores é melhor do que as que usam interpretadores. Sendo Java um meio termo entre os dois métodos, tendo um desempenho melhor que todas as linguagens interpretadas, mas tendo um desempenho pior que todas as linguagens compiladas, fazendo jus a classificação de linguagem híbrida.

Um destaque tem de ser dado a C++ no Gráfico 6.6, visto que é a única das linguagens que tem um tempo de processamento em modo usuário maior do que o seu tempo real. Esse fato se da por causa do compilador fazer otimizações que resultam em um melhor aproveitamento dos múltiplos núcleos do ambiente, assim gerando um menor tempo real

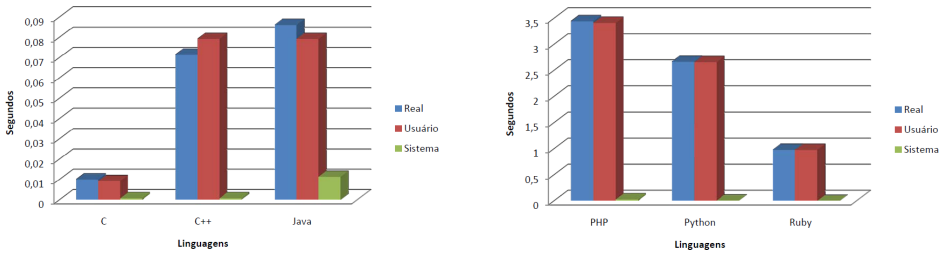


Figura 6.6: Linhas de código por implementação em cada linguagem do programa Ackermann

do que de processamento total.

### 6.2.2 Cliente-Sevidor de Echo

Diferentemente dos outros problemas o desta seção é regido pelo tempo de processamento em modo sistema ao invés de pelo tempo de processamento em modo usuário, como pode ser visto no Gráfico 6.7, a exceção de Java que devido a sua JVM possui um comportamento diferente.

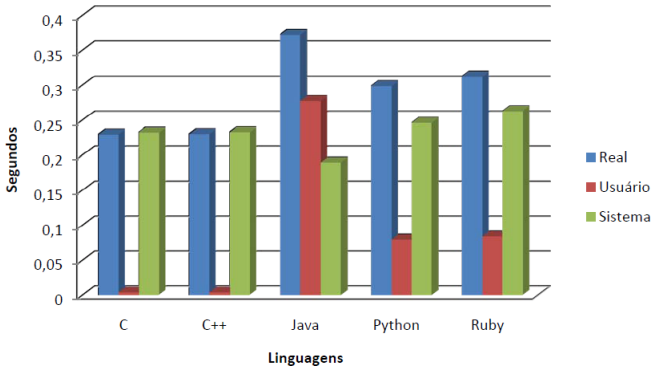


Figura 6.7: Linhas de código por implementação em cada linguagem do programa Cliente-Servidor de Echo

Neste contexto Java se comporta de maneira mais parecida com as linguagens interpretadas do que as compiladas, visto que o seu tempo real é maior que o de sistema. Isso acontece por causa da utilização da máquina virtual que fica na camada acima do sistema, assim sendo considerado processamento de usuário. Essa camada de abstração a mais deixou Java com o pior desempenho entre as linguagens.

Além disso, os resultados mostram um tempo muito similar entre todas as implementações. Sendo isso consequência dos programas serem fortemente dependentes das chamadas de sistema para comunicação de rede.

Sendo o motivo para a ausência de PHP no gráfico, já citada na Seção 6.1.

### 6.2.3 Contador de Frequência de Palavras

Como pode ser visto no Gráfico 6.8, o tempo de processamento foi distribuído entre os processadores de uma melhor forma na maioria das linguagens, a única exceção foi PHP.

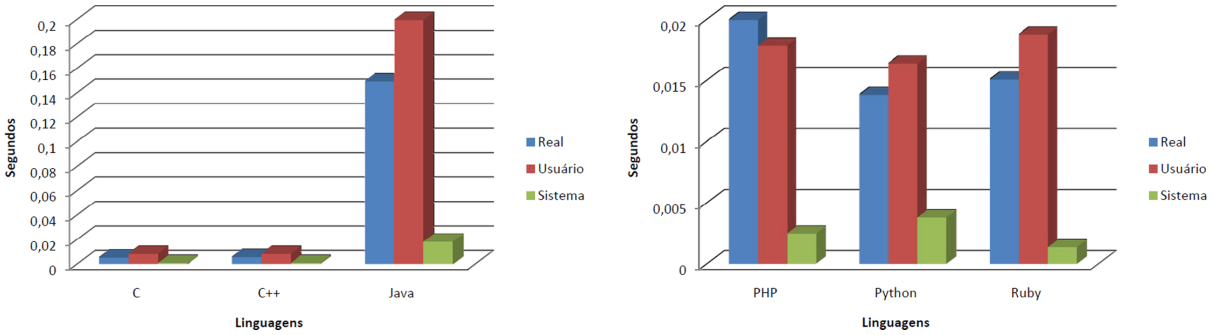


Figura 6.8: Linhas de código por implementação em cada linguagem do programa Contador de Frequência de Palavras

No Gráfico 6.8, chama muito a atenção o desempenho muito baixo de Java. Java obteve esse tempo por causa de seu desempenho em relação a entrada e saída de dados[1] e a forma como é implementada a classe *HashMap*[2].

É válido ressaltar as linguagens interpretadas cujos tempos são muito parecidos e muito maiores que os das linguagens compiladas. Isso é resultado das otimizações feitas pelo compilador as quais não existem nos interpretadores.

### 6.2.4 Laços encadeados

Neste problema é nítida a vantagem do Java em relação as outras linguagens, como se pode ver pelo tempo de execução no Gráfico 6.9

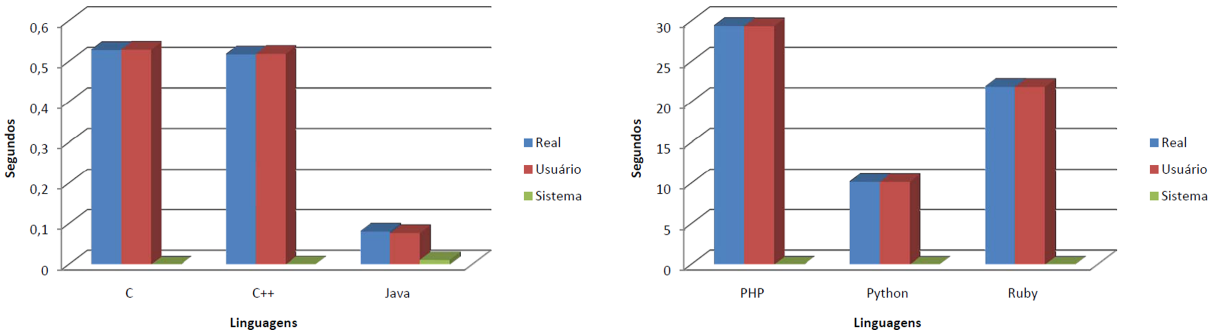


Figura 6.9: Linhas de código por implementação em cada linguagem do programa Laços Encadeados

Aqui as linguagens compiladas levam muita vantagem perante as interpretada, isso porque o compilador faz uma previsão de que o laço será executado, assim conseguindo um redução do tempo.

### 6.2.5 Olá Mundo

Este problema mostra quanto tempo é preciso para carregar o ambiente mínimo para execução do programa, pois o tempo de execução do comando de imprimir na tela a frase é praticamente zero.

[1] An In-Depth Examination of Java I/O Performance and Possible Tuning -

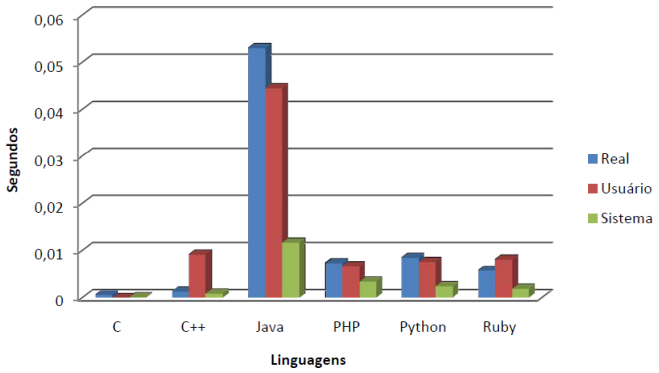


Figura 6.10: Linhas de código por implementação em cada linguagem do programa Olá Mundo

As linguagens que precisam somente do compilador são muito mais rápidas comparadas as que precisam de um interpretador. Isso ocorre devido a previsão de abrir as iterações feita pelos compiladores, que os interpretadores não possuem.

Java que tem toda estrutura da JVM para ser carregada para poder executar, independentemente de o que será executado tem um desempenho bem inferior.

Este problema mede o *overhead* mínimo necessário para a execução de qualquer problemas na determinada linguagem.

## 6.3 Eficiência Relativa

Para essa métrica é necessária a decisão de utilizar como medida relativa a linguagem que possui o melhor desempenho dentre todas. Tendo em vista os resultados gerados na Seção 6.2, foi escolhida a linguagem que obteve o melhor desempenho, a linguagem C, para utlizar como linguagem base para fazer referência as outras.

Utilizando os dados gerados na Seção 6.3, foi calculada a eficiência relativa média de C em relação a todas as linguagens. Tendo como base todos os programas implemetados, assim demonstrando quão pior ou melhor as linguagens são em relação a de mais baixo nível escolhida nesse estudo.

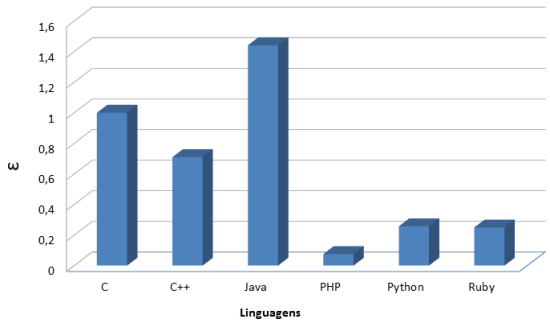


Figura 6.11: Eficiência relativa a C linguagem e as linguagens do estudo

O ideal para essa métrica ser precisa é ter programas complexos que englobem as

pages.cs.wisc.edu/ guo/projects/736.pdf

[2]HashMap Class - http://download.oracle.com/javase/1.4.2/docs/api/java/util/HashMap.html

execuções reais. Apesar dos problemas relacionados não serem em sua maioria deles complexos, eles tem diferentes características que contribuem para uma maior abrangência da métrica, atingindo pequenos contextos de programas reais.

Como esperado para esta métrica, foram encontradas menores eficiências relativas para as linguagens interpretadas. Sendo também essas as linguagens mais voltadas para uma codificação mais simples e intuitiva, como visto nos resultados da métrica de tamanho do código na seção 6.1.

Nas linguagens parecidas com C, em seus conceitos, foram obtidos, por consequência, uma eficiência relativa próxima a de C.

## 6.4 Análise Geral dos Resultados

Não existe uma relação direta entre as métrica avaliadas no estudo, porém se forem analisados os resultados gerados nas seções 6.1 e 6.2 podemos chegar a características das linguagens que são pertinentes. Esses são os assuntos que serão abordados nessa seção.

*C*

Teve o melhor desempenho nas métricas, porém a complexidade e tamanho dos códigos gerados dificultam o desenvolvimento.

*C++*

Os programas e resultados gerados por C++ acabam sendo muito similares, tanto em desempenho quanto em codificação com o desenvolvimento em C.

*Java*

Mesclando paradigmas e métodos de C com linguagens de alto nível, obteve resultados próximos de C em relação a desempenho e resultados não tão distantes das linguagens de alto nível em relação ao código gerado. O diferencial para os resultados foi devido principalmente a JVM e suas otimizações sobre o *bytecode* gerado.

*PHP*

A linguagem interpretada de maior popularidade mostrou seus pontos fortes e seu ponto fraco, seu código simples e pequeno na maioria das soluções, porém um desempenho bem aquém de todas as linguagens do estudo. Esses fatores resultaram na pior eficiência relativa a C.

*Python*

É das melhores linguagens no comparativo de desempenho em relação ao tamanho do fonte. Possuí um fonte simples e pequeno os quais solucionaram problemas complexos. Isso junto de uma eficiência relativa de aproximadamente 0.5 em relação a C.

*Ruby*

Muito similar nos resultados a Python, porém com um código fonte menor e aproximadamente a mesma eficiência relativa em relação a C.

A exemplo do estudo (ORAM; WILSON, 2010), considerando todos os resultados citados durante este trabalho, pode-se dizer que não existe uma linguagem melhor que outra dentre as estudas, contudo exitem linguagens melhores em certas métricas.

# 7 CONCLUSÃO

Neste trabalho foi trazida a proposta de medir as linguagens de programação mais populares atualmente, segundo métodos e problemas variados. Este capítulo apresenta as considerações finais deste trabalho. Sendo organizado em resumo dos resultados e contribuições, limitações do trabalho e propostas de trabalhos futuros.

## 7.1 Resultados e Contribuições

Os resultados gerados começam com a importância da análise do alvo da avaliação, sendo vistos detalhes de cada linguagem. Em seguida, foram apresentadas as definições de avaliação, os problemas e as métricas, que foram aplicadas sobre o alvo. Por fim, apresentados os resultados de forma informativa e analítica.

Como abordado durante o trabalho, os dados gerados contribuem para:

- Fornecer dados relevantes a comparações entre as linguagens.
- Auxiliar na previsão de desempenho e tamanho de código fonte de projetos, usando este trabalho como base.
- Aumentar a confiança em previsões baseando-se em resultados práticos, principalmente em migrações de sistemas para linguagens diferentes.
- Verificar vantagens de métodos e paradigmas em relação a nichos de problemas.

Sendo os principais tópicos propostos por esse trabalho plenamente abordados.

## 7.2 Limitações do Trabalho

A proposta apresentada pelo presente trabalho tem como foco medir as linguagens de programação atuais. Porém houve dificuldade em definir o que usar para mensurar as linguagens e quais problemas produziriam soluções relevantes para as métricas em questão. Então foram definidos problemas que parcialmente forneceram informações a respeito do objetivo, visto que ficaram ausentes muitos nichos de aplicações que não foram avaliadas.

Outras limitações são baseadas na construção das soluções. Para esse ponto podem ser levantados as seguintes limitações:

- Variação do conhecimento inicial das linguagens por parte do desenvolvedor que implementou as soluções. Por consequência disso, variando a qualidade das soluções.

Sendo assim, o programador e seu conhecimento serem variáveis importantes em estudo de linguagens. Essa é uma variabilidade humana que deve ser considerada sempre (AUGUSTINE, 1997).

- Como abordado por Oram e Greg Wilson em (ORAM; WILSON, 2010) ao estudar o *Plat_forms*[1], por mais que esse o presente estudo tenha um número grande de implementações, esse número ainda assim é insuficientes para obtenção de números reais por causa dos tipos de aplicações que não foram abordadas. Além desta deficiência, existe o fator implementações a partir de variados algorimos que resolvam a solução.

- Falta de ferramentas de código aberto para medição de desempenho.

## 7.3 Trabalho Futuros

Devido as limitações citadas anteriormente, pode-se propor os seguintes possíveis trabalhos:

- Estender o presente trabalho produzindo mais implementações, utilizando os mesmos meios de medição para agregar maior precisão aos dados gerados.

- Desenvolver um *software* livre que implemente as métricas.

- Estender o presente trabalho medindo também a métrica de poder relativo(KENNEDY; KOELBEL; SCHREIBER, 2004) dos problemas citados para completar a avaliação de produtividade.

[1]The web development platform comparison - http://www.plat-forms.org

# REFERÊNCIAS


AKITA, F. **Repensando a web com Rails**. 1ª.ed. [S.l.]: Brasport, 2006.

AUGUSTINE, N. **Augustine's laws**. [S.l.]: American Institute of Aeronautics and Astronautics, 1997.

BERGIN JR., T. J.; GIBSON JR., R. G. (Ed.). **History of programming languages(volume II)**. 1ª.ed. New York, NY, USA: ACM, 1996.

BIANCUZZI, F.; WARDEN, S. **Masterminds of programming**. [S.l.]: O'Reilly, 2009. (O'Reilly Series).

COMER, D. **Redes de Computadores e Internet**. 4ª.ed. [S.l.]: Bookman, 2007. 48-49p.

DALL'OGLIO, P. **PHP**: programando com orientação a objetos. 2ª.ed. [S.l.]: Novatec, 2009.

DEITEL, H. **C ++ Como Programar**. [S.l.]: Bookman, 2001.

DEITEL, H. **Java**: como programar. [S.l.]: Bookman, 2003.

DEPT, S. U. C. S.; KNUTH, D. **Mathematics and computer science**: coping with finiteness. [S.l.]: Computer Science Dept., School of Humanities and Sciences, Stanford University, 1976. (Report (Stanford University. Computer Science Dept.)).

EDELSON, J.; LIU, H. **JRuby Cookbook**. [S.l.]: O'Reilly, 2008. (Cookbook Series).

FLANAGAN, D.; MATSUM, Y. **A Linguagem de Programação Ruby**. [S.l.]: Alta Books, 2009.

GAMMA, E.; JOHNSON, R.; VLISSIDES, J.; HELM, R. **Padrões de projeto**. [S.l.]: Bookman, 2000.

HARRISON, R.; SAMARAWEERA, L. G.; DOBIE, M. R.; LEWIS, P. H. Comparing Programming Paradigms: an evaluation of functional and object-oriented programs. **Software Engineering J.**, [S.l.], v.11, n.4, p.247–254, July 1996.

HOLZNER, S. **PHP**: the complete reference. [S.l.]: McGraw-Hill, 2007.

JONES, C. **Software Engineering Best Practices**: lessons from successful projects in the top companies. 2ª.ed. [S.l.]: McGraw-Hill, 2009. 537–550p.

JONES, D. M. **The New C Standard**: an economic and cultural commentary. 2005.

KENNEDY, K.; KOELBEL, C.; SCHREIBER, R. Defining and Measuring the Productivity of Programming Languages. **Int. J. High Perform. Comput. Appl.**, Thousand Oaks, CA, USA, v.18, p.441–448, November 2004.

RITCHIE, D. M. (Ed.). **The C Programming Language**. 2ª.ed. [S.l.]: Prentice Hall Professional Technical Reference, 1988.

MARTIN, R. **Java and C++** : a critical comparison. Acessao em 19 de junho de 2011, `http://www.objectmentor.com/resources/articles/javacpp.pdf`.

MICROSYSTEMS, S. **Portability Verification of Applications for the J2EE Platform**. Acessao em 19 de junho de 2011, `http://java.sun.com/developer/technicalArticles/J2EE/portability/`.

MICROSYSTEMS, S. **Design Goals of the Java Programming Language**. Acessado em 18 de junho de 2011, `http://java.sun.com/docs/white/langenv/Intro.doc2.html`.

OLSEN, R. **Eloquent Ruby**. [S.l.]: Pearson Education, Limited, 2011. (Addison-Wesley Professional Ruby Series).

ORAM, A.; WILSON, G. **Making Software**: what really works, and why we believe it. [S.l.]: O'Reilly Media, 2010. (O'Reilly Series).

RESEARCH, T. **Java Application Servers Report**. Acessado em 15 de junho de 2011, `http://www.fscript.org/prof/javapassport.pdf`.

SEBESTA, R. W. **Conceitos de Linguagem de Programação**. 5ª.ed. [S.l.]: Bookman, 2003. 38–43p.

STEWART, B. **An Interview with the Creator of Ruby**. Acessado em 19 de junho de 2011, `http://linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html`.

STROUSTRUP, B. **The design and evolution of C++**. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994.

STROUSTRUP, B. C and C++: case studies in compatibility. **C/C++ Users' Journal**, [S.l.], p.9–20, 2002.

SUNDBLAD, Y. The Ackermann Function. A Theoretical, Computational, and Formula Manipulative Study. , [S.l.], v.11, n.1, p.107–119, Mar. 1971.

TELLES, M. **Python power!**: the comprehensive guide. [S.l.]: Thomson Course Technology, 2008. (Safari Books Online).

THE PHP, G. **Usage Stats for April 2007**. Acessao em 19 de junho de 2011, `http://www.php.net/usage.php`.

THE PHP, G. **History of PHP and related projects**. Acessao em 19 de junho de 2011, `http://www.php.net/history`.

THIAGO GALSEI, O. S. **Python e Django**: desenvolvimento Ágil de aplicações web. [S.l.]: Novatec, 2010.

TIOBE, C. **TIOBE Programming Community Index**. Acessado em 13 de junho de 2011, http://www.tiobe.com/index.php/content/paperinfo/tpci/.

W. APPEL, A. **Garbage Collection Can Be Faster Than Stack Allocation**. Acessao em 11 de junho de 2011, http://www.cs.princeton.edu/~appel/papers/45.ps.

ZAPALOWSKI, V. **Resultados dos testes do trabalho**. Acessao em 19 de julho de 2011, http://github.com/vzapalowski/languagesbenchmarck.