Análise comparativa entre Groovy e Java, aplicado no desenvolvimento web

Vandir Fernando Rezende, Marcel Hugo

Departamento de Sistemas e Computação
Universidade Regional de Blumenau (FURB) – Blumenau, SC – Brasil

rezende.vandir@gmail.com,marcel@furb.br

Resumo. *Este artigo apresenta uma análise comparativa, no desenvolvimento de softwares web, entre as linguagens de programação Groovy e Java. Baseando-se na norma NBR ISO/IEC 9126-1 foram determinados os critérios de avaliação. Para permitir a comparação foi implementado um aplicativo estudo de caso em ambas as linguagens, com o intuito de medir a diferença de produtividade no desenvolvimento dos casos de usos. Para comparar o desenvolvimento, foi calculado a UCP dos casos de usos e posteriormente verificado o tempo e o desempenho em cada linguagem.*

1. Introdução

Com a necessidade de criar ferramentas que facilitassem o seu trabalho diário, o homem passou a aprimorar cada vez mais os computadores e seus sistemas (SAMPAIO, 1999). Diante da informatização dos processos, a linguagem de programação Java tem se mostrado importante, principalmente pelo fato de ser muito abrangente, pois contempla desde dispositivos móveis até *mainframes*. Para isso, a mesma utiliza-se de uma das suas principais características, a portabilidade (JAVA, 2010).

Segundo Oliveira (2009), Java conta com inúmeros *frameworks*, cada um especializado em um ramo distinto do desenvolvimento de software, incluindo desde a modelagem do banco de dados até a criação das interfaces visuais. Para ter esta flexibilidade, Java precisa ser especificamente configurado a cada funcionalidade, pois não conta com configurações predefinidas, ocasionando assim uma perda de produtividade no processo, em casos que o tempo levado para configurar a solução é maior que o tempo gasto com o desenvolvimento da regra de negócio. Pensando neste fato, foram propostas diversas soluções para a evolução da linguagem Java, principalmente focando no desenvolvimento ágil. Assim surge o JRuby e o Grails (Groovy on Rails), tentativas de aproximar a programação Java com a filosofia ágil (PETERSON, 2010).

Com o lançamento do *framework* Grails, para desenvolvimento de aplicações web, Groovy ganhou credibilidade e o interesse da comunidade Java, provando assim, que o desenvolvimento de aplicações web escaláveis são produtivas e fáceis. Grails utilizou desde recursos básicos do Groovy, até recursos complexos de aplicações web, como persistência em banco de dados, Ajax, *webservices*, relatórios, processamento em lote, e *plugins* que permitem aos desenvolvedores melhorar e criar novas ferramentas que auxiliam o desenvolvimento (JUDD; NUSAIRAT; SHINGLER, 2008).

Neste cenário, foi desenvolvido um aplicativo de gestão de projetos, sendo ele implementado em Java e em Groovy, com o intuito de comparar as linguagens. Tomou-se como base para a análise comparativa a norma NBR ISO/IEC 9126-1 (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 2003).

2. NBR ISO/IEC 9126-1

A NBR ISO/IEC 9126-1 fornece um modelo de propósito geral, o qual define as seis amplas categorias de características de qualidade de software, que tem por objetivo servir de referência básica na avaliação de produto de software: funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade.

Comparar linguagens de programação (LPS) não é exatamente o objetivo dessa norma, contudo é o documento técnico que mais se aproxima com o objetivo proposto. Desta forma, tornou-se necessário adaptar o modelo para a análise comparativa de linguagens de programação, elencando as características e subcaracterísticas que são relevantes a este tipo de comparação. Assim foram definidas as seguintes características como critérios de avaliação, a serem aplicados na análise comparativa:

a) produtividade/custo: modificando o modelo proposto, a fim de atender a análise comparativa, foi incluso o item produtividade na categoria funcionalidade, onde Sebesta (2003) alega que de todos os fatores que contribuem para os custos da linguagem, três são os mais importantes: desenvolvimento do programa, manutenção e confiabilidade – uma vez que essas são funções da capacidade de escrita e da legibilidade;

b) usabilidade: nas linguagens de programação pode-se avaliar a dificuldade de entendimento (inteligibilidade) dos padrões utilizados nela, assim como o custo de aprendizado (apreensibilidade) dos programadores que estão ingressando na mesma;

c) eficiência: através de programas gerados pela linguagem de programação, é possível medir o tempo de execução de determinado rotina ou caso de uso e quanto recurso de hardware foi utilizado para essa execução;

d) manutenibilidade: o quanto a estrutura da linguagem auxilia na detecção de falhas (analisabilidade), assim como na alteração de códigos existentes (modificabilidade);

e) confiabilidade: capacidade do programa em executar de acordo com as suas especificações sob todas as condições.

A definição por estas características da qualidade e o entendimento de cada uma delas guiou a seleção das características das LPS a serem empregadas na comparação.

3. Características das Linguagens de Programação

Para se tratar cientificamente a programação, deve ser possível especificar precisamente as propriedades necessárias dos programas. O formalismo não é necessariamente um fim por si só. A importância das especificações formais deve, no final, se justificar pela sua utilidade – sejam elas usadas ou não para melhorar a qualidade do software ou reduzir os custos de produção e manutenção de software (J. HORNING apud TUCKER; NOONAN, 2008).

As características de linguagens de programação avaliadas neste trabalho são:

a) ortogonalidade: uma linguagem é dita ortogonal se os seus comandos e recursos são construídos sobre um conjunto pequeno e mutuamente independente de operações primitivas. Quanto mais ortogonal uma linguagem, menos regras excepcionais são necessárias para escrever programas corretos. Assim, programas ortogonais tendem a ser mais simples e claros (TUCKER; NOONAN, 2008). Portanto, ortogonalidade é um número reduzido de construções primitivas e um conjunto consistente de regras para combiná-las. A ortogonalidade está estreitamente relacionada à simplicidade, pois quanto mais ortogonal é a linguagem, menos exceções as regras das linguagens exigirão, significando assim, um grau mais elevado de regularidade na mesma, tornando-a mais fácil de ser aprendida, lida e entendida (SEBESTA, 2003). Por outro lado, demasiada ortogonalidade pode resultar em prejuízo para a capacidade de escrita, devido a pouca quantidade de construções primitivas;

b) simplicidade global: é o resultado de uma combinação de um número relativamente pequeno de construções primitivas e uso limitado do conceito de ortogonalidade (SEBESTA, 2003). A simplicidade global afeta fortemente a legibilidade de uma linguagem. Uma linguagem com um grande número de componentes básicos é mais difícil de ser aprendida. Os programadores que precisam usar uma linguagem grande tendem a aprender um subconjunto dele e ignorar seus demais recursos, porém linguagens muito simples tornam os programas menos legíveis (TUCKER; NOONAN, 2008);

c) legibilidade: é a facilidade que os programas podem ser lidos e compreendidos. Um grande exemplo de uma estrutura de controle que afeta diretamente a legibilidade é o comando `GO TO`, pois determina onde o fluxo de instruções irá continuar. Logo, um programa que pode ser lido de cima a baixo é muito mais fácil de ser entendido do que um programa que exige o leitor pular de uma instrução a outra não-adjacente para seguir a ordem de execução (SEBESTA, 2003);

d) tipo de dados e estrutura: uma variável que represente verdadeiro ou falso é muito mais legível se puder atribuir `true/false` do que `1/0`. Quanto mais próximo da realidade for a estrutura, mais fácil dela ser entendida e representada corretamente;

e) sintaxe: é um conjunto de regras que define a forma de uma linguagem, estabelecendo como são compostas as suas estruturas básicas. Para Sebesta (2003), a sintaxe afeta diretamente a legibilidade da linguagem de programação. Restrições, como o limite de caracteres na definição de identificadores, prejudicam a interpretação do código escrito, porém recursos como palavras reservadas e identificação de início e fim de blocos, auxiliam para uma maior legibilidade;

f) capacidade de escrita: é uma medida de quão facilmente uma linguagem pode ser usada para criar programas para um domínio de problema escolhido. Comparar a capacidade de escrita de uma linguagem não é fácil, varia de acordo com o propósito de cada linguagem (SEBESTA, 2003);

g) abstração: significa a capacidade de definir e, depois, de usar estruturas ou operações complexas de uma maneira que permita ignorar muitos dos detalhes. A abstração é um aspecto fundamental do processo de projeto de programas. Os programadores gastam muito tempo construindo abstrações, tanto de dados quanto

procedurais, para explorar a reutilização de código e evitar reinventá-lo (TUCKER; NOONAN, 2008). Bibliotecas que acompanham linguagens modernas de programação comprovam a experiência acumulada de programadores na construção de abstrações;

h) expressividade: é a função entre a computação realizada e a quantidade de comandos utilizados, ou seja, quanto maior for a computação realizada e menor seja o programa, maior será a expressividade da linguagem (SEBESTA, 2003);

i) verificação de tipos: característica que a linguagem possui para testar se existem erros de tipagem de dados em determinado programa, seja pelo compilador ou durante a execução do programa (SEBESTA, 2003);

j) tratamento de exceção: capacidade de uma linguagem constatar que ocorreu algo inesperado e permitir tratar tal situação (SEBESTA, 2003);

k) confiabilidade: capacidade do programa de se comportar de acordo com as suas especificações sob todas as condições. É a necessidade de se projetar mecanismos apropriados de manipulação de exceções na linguagem. Além disso, linguagens que restrinjam o uso de *aliases* e vazamento de memória, que suportem tipagem forte, tenham sintaxe e semântica bem definidas e que suportem a verificação e validação de programas têm uma vantagem nessa categoria (TUCKER; NOONAN, 2008). Tanto a legibilidade como a capacidade de escrita influenciam a confiabilidade. Um programa escrito em uma linguagem que não suporta maneiras naturais de expressar os algoritmos exigidos usará, necessariamente, métodos não-naturais. Estes últimos têm menos probabilidade de estarem corretos para todas as situações possíveis (SEBESTA, 2003).

4. Meio de Avaliação dos Critérios

Com o intuito de quantificar os critérios determinados pela NBR ISO/IEC 9126-1, pode-se correlacionar estes com as principais características das linguagens de programação. Assim sugere-se as seguintes correlações apresentadas no quadro 1.

Quadro 1 – Correlação dos critérios com as características

	Custo	Usabilidade	Eficiência	Manutenibilidade	Confiabilidade
Ortogonalidade	X	X		X	X
Simplicidade global		X			X
Legibilidade	X			X	X
Tipo de dados e estrutura		X			
Sintaxe		X			
Capacidade de escrita	X		X	X	
Abstração		X	X	X	X
Expressividade	X		X	X	
Verificação de tipos	X				X
Tratamento de exceção	X		X		X

Desta forma, ao concluir que determinada característica em uma linguagem é mais relevante do que em outra, estarão sendo aplicados os critérios de avaliação estabelecidos pela norma NBR ISO/IEC 9126-1.

Estabelecidas as características das linguagens de programação a serem avaliadas, foi proposto o seguinte meio de avaliação para cada critério, onde se

classificou as características em dois grupos distintos, no qual foram denominados pelos autores de “estático” e “dinâmico”.

O grupo “estático” engloba as características que dispensam implementações para efetuar a comparação, onde analisando as linguagens e suas estruturas pode-se estipular uma métrica e verificar qual linguagem possui mais características que auxiliam a produtividade no desenvolvimento do software. Neste grupo estão atribuídos os seguintes critérios: usabilidade, manutenibilidade e confiabilidade. Estes critérios foram classificados como “estáticos”, pois estão diretamente relacionados a itens das linguagens, tais como: sintaxe, semântica, padrões, estrutura e documentação. Assim estes foram analisados através de um questionário para avaliar cada linguagem, obtendo um valor final para distingui-las.

Quanto aos critérios “dinâmicos”, diferentemente dos “estáticos”, necessitam da implementação do estudo de caso, para que os mesmos possam ser mensurados. Portanto os itens eficiência e produtividade estão inclusos neste grupo.

Para avaliar a produtividade de cada linguagem, é calculado o *Use Case Point* (UCP) por caso de uso e mensurado o tempo de desenvolvimento destes. Assim, a produtividade está em razão da seguinte função:

Quadro 2. Função estipulada para medir a produtividade.

Produtividade = (tempo de desenvolvimento) / UCP

Ao término do desenvolvimento, são obtidos os valores por caso de uso em cada linguagem, podendo assim compará-las. Além disto, é possível analisar a eficiência de cada linguagem, o tempo gasto e o hardware consumido em cada rotina executada.

5. Diferenças entre Groovy e Java

Para Doederlein (2006), a semântica da linguagem Groovy foi planejada para ser familiar aos programadores Java, pela sua similaridade com os fontes escritos em Java, reduzindo assim a curva de aprendizagem, porém há diferenças entre as linguagens, tais como (exemplificadas na Figura 1):

a) objetos: todas as variáveis são instâncias de `java.lang.Object` - ao contrário do Java em que há tipos primitivos e não primitivos, em Groovy tudo é considerado objeto;

b) tipagem dinâmica: ao definir uma variável em Groovy, declarar o tipo dela é opcional, pois Groovy utiliza *duck typing* para definir os tipos das variáveis. A definição de duck typing é: se age como um pato, é um pato. No caso do Groovy, se uma variável age como um determinado objeto (integer, string), ela será definida como tal;

c) ponto-e-vírgula: se houver apenas um comando na linha, o ponto-e-vírgula, como delimitador de fim de comando, torna-se opcional;

d) `return`: dada uma função, o valor de retorno do último comando corresponde ao valor de retorno da mesma;

e) igualdade: o símbolo `==` significa igualdade para tudo, diferentemente do Java, onde `==` é utilizado para tipos primitivos e `.equals()` para objetos. Em Groovy é sempre utilizado `==` para verificar igualdade;

f) conceito de verdade: Groovy considera verdadeiro qualquer valor que seja diferente de `null`, assim qualquer variável, quando testada, irá retornar `true` ou `false`, desta forma o famoso `NullPointerException` do Java, deixa de existir no Groovy.

g) concatenação de string: em Java, qualquer objeto para ser concatenado com string, antes deve ser convertido para string, já em Groovy isto não é necessário, basta utilizar a seguinte sintaxe: O resultado é `${nome_da_variavel}`;

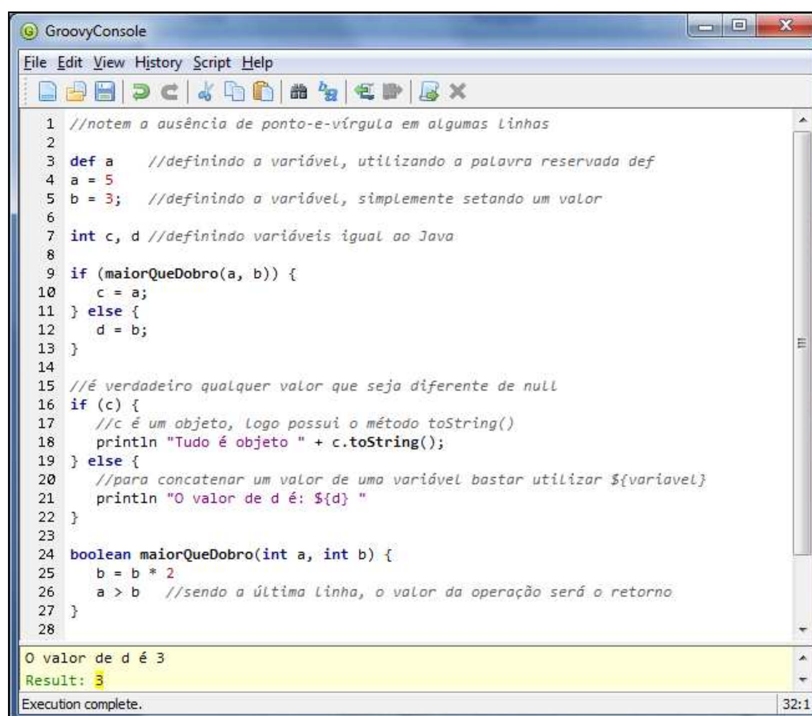

Figure 1. Código Groovy

h) laços de repetição: assim como no Java, em Groovy há várias maneiras de criar laços de repetições, onde o comando `while` é idênticos nas linguagens, porém o comando `for` possui variações.

i) Groovy Beans: para ganhar produtividade no desenvolvimento do código, o Groovy gera dinamicamente os métodos assessores (*getters & setters*).

j) GORM: Segundo DICKINSON (2008), Groovy abstrai o mapeamento objeto relacional e a persistência com o banco de dados através do *framework* GORM, o qual gera um ganho de produtividade no desenvolvimento, pois possui métodos padrões de acesso ao banco de dados, tais como: `save, delele, update, get` e `findBy`.

6. Estudo de Caso

Estabelecido o crivo de avaliação, realizou-se a análise do estudo de caso, com o objetivo de implementar as rotinas deste, tanto em Java quanto em Groovy, tendo como finalidade a comparação do desenvolvimento entre as linguagens supracitadas.

A análise deste foi modelada através de diagramas UML. Assim foram desenvolvidos os seguintes diagramas de caso de uso (figura 2).

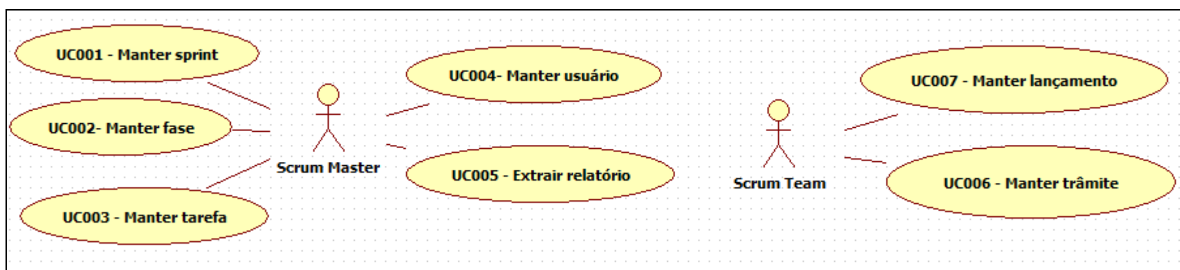

Figura 2. Casos de uso do aplicativo exemplo

A figura 3 apresenta o digrama de classes desenvolvido para o estudo de caso. Para cada classe no diagrama há métodos acessores (*getters & setters*), porém apenas os demais métodos serão exibidos, por questão de legibilidade do diagrama. Mais detalhes podem ser conhecidos em Rezende (2011).

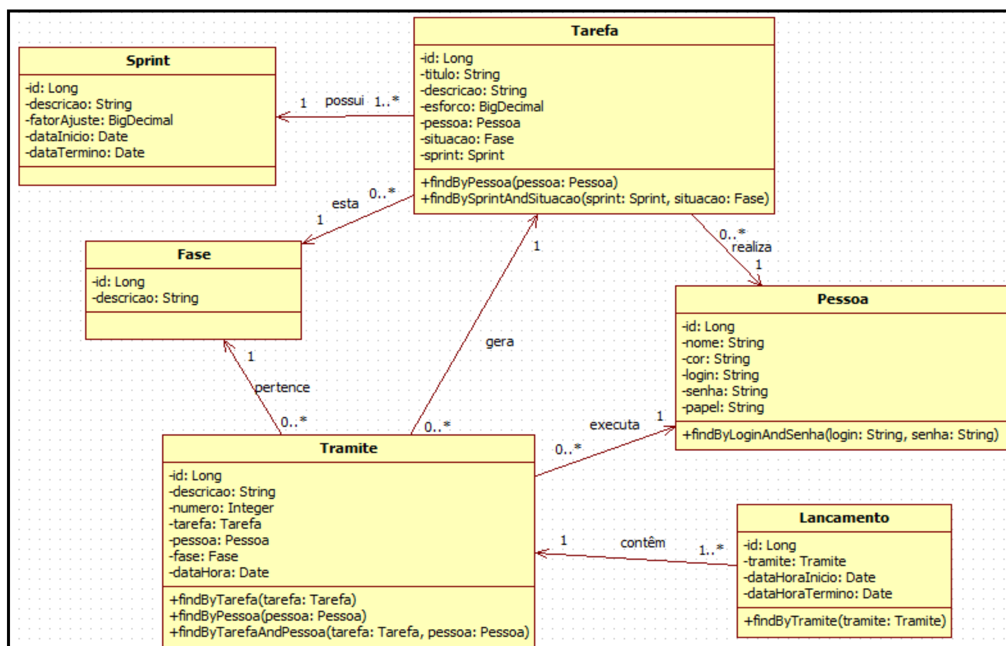

Figura 3. Diagrama de classes do aplicativo exemplo

7. Implementação

7.1. Java

O desenvolvimento do estudo de caso na linguagem Java foi realizado através do *framework* jCompany, tendo em vista que Groovy utiliza-se do seu *framework* para desenvolvimento web (Grails). Então, para equiparar as linguagens, o autor optou por este *framework* nacional e *open-source*, que traz uma arquitetura de software de alto nível, reutilizável e extensível (ALVIM, 2011).

Tendo os casos de uso estabelecidos e respectivamente seus diagramas de classes codificados (classes da camada de modelo), o desenvolvedor utiliza-se dos *wizards* disponibilizados pelo jCompany, para configurar e mapear os objetos relacionais, assim como para gerar a camada de visão, configurando a geração das telas de consultas e edição.

Contudo, mesmo observando-se que o jCompany aumenta a produtividade, comparado ao desenvolvimento totalmente manual, o código gerado necessita de ajustes para o pleno funcionamento do caso de uso. Outro fato são as configurações persistidas em arquivos XML, no qual posteriormente podem dificultar a manutenção dos casos de uso.

7.2. Groovy

Para desenvolver os estudos de caso em Groovy, utilizou-se o framework Grails, no qual Rudolph (2009) cita que a marcante característica do Grails em apoiar o desenvolvimento ágil, vem da filosofia de convenção sobre configuração, no qual em poucos passos é possível configurar o ambiente de desenvolvimento e criar um projeto web. Assim, efetuado o *download* do *framework*, disponível na página (grails.org), basta descompactar em uma pasta, que será a base deste, e seguir os demais passos:

a) variável de ambiente: criar a variável GRAILS_HOME, e informar o caminho da pasta onde foi descompactado o framework;

b) path: adicionar o caminho GRAILS_HOME/bin na variável de ambiente PATH do sistema operacional;

c) testar ambiente: acessar o sistema operacional via linha de comando e executar o seguinte comando: grails –version.

Com o ambiente de desenvolvimento configurado, o *framework* está pronto para criar um novo projeto web, através dos seguintes passos executados via linha de comando:

a) criar projeto: para criar um projeto execute `grails create-app nome_projeto`, o Grails irá criar um projeto com na estrutura MVC;
b) acessar projeto: `cd nome_projeto`;
c) mapear caso de uso: `grails create-domain-class br.furb.NomeClasse`;
d) definir atributos: editar a classe gerada, adicionando os seus devidos atributos;
e) gerar classe de controle: `grails create-controller br.furb.NomeClasse`;
f) scaffolding: após gerada a classe de controle, deve-se editá-la e adicionar o seguinte comando, `static scaffold = true`. Este comando irá gerar todos os eventos padrões da classe de controle, tais como: criar, listar, editar e excluir (CRUD);
g) gerar a camada de visão: `grails generate-views br.furb.NomeClasse`;
h) iniciar o aplicativo: `grails run-app`, este comando irá iniciar o servidor Jetty, que acompanha o Grails, agora basta acessar o projeto pelo *browser*, disponível em: <http://localhost:8080/nome_projeto>

Ao término destes passos, pode-se observar o quão é produtivo o *framework* Grails, devido às suas configurações por convenção, que simplificam o processo e permitem o desenvolvedor se preocupar apenas com as regras de negócio.

8. Questionário de Avaliação

Para realizar a análise comparativa das linguagens de programação, Groovy e Java, foi preenchido o questionário apresentado no quadro 3, com o intuito de evidenciar as diferenças entre as características "estáticas" das linguagens. O questionário foi respondido pelo autor, baseado nas pesquisas realizadas e em seu conhecimento e experiência com as linguagens (eventualmente outro desenvolvedor poderia alcançar resultados um pouco diferentes em função de sua experiência). As respostas 0, 5 e 10 significam respectivamente "Não", "Parcial" e "Sim".

Quadro 3. Questionário comparativo preenchido

CARACTERISTICA	PERGUNTA	GROOVY			JAVA		
		0	5	10	0	5	10
Ortogonalidade	Há uma única forma para cada construção, ou cada construção pode ser representada por diversas formas?	X				X	
	Existe um padrão na formação de expressões, ou há exceções?		X				X
Simplicidade Global	A LP possui uma grande quantidade de recursos nativos?		X			X	
	É fácil dominar todos os recursos disponibilizados pela LP?		X			X	
Legibilidade	A codificação se aproxima da linguagem natural?		X			X	
	Existe alguma representação para dados booleanos próximos da realidade?			X			X
	Há identificação na abertura e fechamento de blocos de comandos?		X			X	
	É possível utilizar o comando GOTO? Ou alguma derivação do comando?	X			X		
Tipo de dados e estrutura	Existe limite de caracteres ao definir uma variável?			X			X
	As variáveis necessitam de um tipo?	X					X
	A LP o desenvolvedor criar seus próprios tipos de dados?			X			X
Sintaxe	A LP possui uma documentação completa e de fácil acesso?			X			X
	A LP possui uma padronização para definir atributos, métodos e classes?			X			X
	Existem atualizações da linguagem, adicionados novos recursos ou é uma LP consolidada e sem alteração?			X		X	
	A LP possui palavras reservadas? É possível utilizá-las como nome de variáveis?			X			X
Capacidade de escrita	Há um conjunto de configurações iniciais pré-estabelecidas?			X		X	
	A LP é facilmente adaptável para o desenvolvimento do que se propõem?			X		X	
	A LP é totalmente orientada objetos ou a LP possui tipos primitivos?			X		X	
	A LP é totalmente orientada objetos ou a LP possui tipos primitivos?			X		X	
Abstração	A LP permite a aplicação das técnicas de abstração como Herança e polimorfismo?			X			X
	A LP possui herança múltipla?	X			X		
	A LP sugere a utilização de padrões de projetos?			X			X
Expressividade	A LP disponibilidade uma grande quantidade de bibliotecas prontas ao programador?			X			X
	A LP possui facilidades para codificação que geram grande resultado computacional?			X		X	
Confiabilidade	Existe o conceito de ponteiros na LP?	X			X		
	Existe um gerenciamento de memória na LP? É necessário informar a quantidade de memória a ser alocada e desalocá-la manualmente?			X			X

	Os programas gerados podem executar em qualquer plataforma?			X			X
Verificação de tipos	Há a verificação de tipo dinamicamente? Antes de compilar ou ao compilar?		X				X
	A LP é fortemente tipada?	X				X	
Tratamento de exceção	Existe o tratamento de exceção na LP?			X			X
	A LP permite tratar diferentes tipos de exceções de formas distintas?			X			X

As questões que obtiveram respostas distintas foram destacadas para facilitar a análise. Porém em grande parte do questionário, as linguagens comparadas tiveram respostas iguais, isto ocorre devido ao fato que Groovy é uma linguagem estendida do Java, e conseqüentemente herda muitas das suas características.

Baseado no questionário, foi possível concluir que algumas características se destacam mais em determinada linguagem (quadro 4).

Quadro 4. Características destacadas de cada linguagem.

CARACTERÍSTICA	GROOVY	JAVA
Ortogonalidade		X
Simplicidade global		X
Legibilidade		X
Tipos de dados e estrutura	X	
Sintaxe	X	
Capacidade de escrita	X	
Abstração	X	
Expressividade	X	
Confiabilidade		X
Verificação de tipos		X
Tratamento de exceção	X	

Ao responder o questionário, notou-se grande semelhança entre as linguagens, porém no quadro 4 pode-se observar que Groovy evoluiu sua linguagem aprimorando recursos em busca de produtividade. Logo, a linguagem conta com uma expressividade e abstração maior do que o Java. Em contrapartida, fica evidente que o Java é uma linguagem mais madura e estabelecida, destacando-se características como confiabilidade e ortogonalidade.

9. Resultados Obtidos

Durante o desenvolvimento dos casos de uso, mensurou-se o tempo gasto em cada linguagem (Groovy e Java). Obtendo o valor das UCPs de cada caso de uso, o quadro 5 apresenta o resultado da função de produtividade de cada linguagem.

Quadro 5. Comparativo de produtividade

CASO DE USO	GROOVY (min)	JAVA (min)	UCP	F(GROOVY)	f(JAVA)	G - J (min)	%
UC001 - Manter *sprint*	80	135	13,6	5,88	9,93	-55	40,74
UC002 - Manter fase	40	75	13,6	2,94	5,51	-35	46,67
UC003 - Manter tarefa	180	265	19,7	9,14	13,45	-85	32,08
UC004 - Manter usuário	130	210	19,7	6,60	10,66	-80	38,10
UC005 - Extrair relatório	240	45	25,9	9,27	1,74	195	-433,33
UC006 - Manter trâmite	150	210	19,7	7,61	10,66	-60	28,57

UC007 - Manter lançamento	235	360	25,9	9,07	13,90	-125	34,72
Total	**815**	**1255**	**112,20**	**41,24**	**64,11**	**-440**	**-**
Média	**137,83**	**209,17**	**18,70**	**6,87**	**10,64**	**-73,33**	**36,81**

Ao analisar os resultados obtidos (quadro 5), pode-se concluir que para desenvolver em Groovy, o mesmo caso de uso desenvolvido em Java/jCompany, leva-se aproximadamente 35% do tempo a menos. A opção por outro *framework* em Java poderia afetar os resultados do comparativo em termos de produtividade (tanto para mais quanto para menos).

Com o objetivo de comparar o desempenho dos aplicativos, desenvolvidos em cada linguagem, com uma massa de dados de aproximadamente 500 registros por tabela, foram efetuados testes em todos os casos de uso, avaliando as rotinas de inserção, listagem e exclusão. Para medir a performance dos aplicativos, foi utilizado o *plugin* para desenvolvimento web YSlow, o qual calcula o tempo que a requisição demorou para ser processada. O processamento de ambos foi realizado no mesmo computador (cliente e servidor), usando a mesma máquina virtual Java.

Quadro 6. Comparativo de desempenho.

CASO DE USO	Groovy (ms)				Java (ms)				%
	Gerar	Listar	Apagar	Média	Gerar	Listar	Apagar	Média	
UC001 – Manter *sprint*	1078	674	669	**799,67**	972	610	595	**714,33**	**10,07**
UC002 – Manter fase	809	552	521	**622,00**	744	516	479	**570,20**	**7,70**
UC003 – Manter tarefa	956	224	263	**478,67**	887	218	242	**444,59**	**6,65**
UC004 – Manter usuário	847	235	279	**451,33**	779	249	247	**420,33**	**6,32**
UC005 – Extrair relatório	1843	416	-	**1129,50**	1807	398	-	**1102,50**	**2,39**
UC006 – Manter trâmite	417	241	398	**347,67**	379	216	356	**313,01**	**9,51**
UC007 – Manter lançamento	389	215	364	**320,00**	338	198	315	**280,00**	**12,08**
Total	**6334**	**2419**	**2494**	**4202**	**5906**	**2428**	**2234**	**3894**	**-**
Média	**904,86**	**374,14**	**415,67**	**600,31**	**843,71**	**346,87**	**372,33**	**556,28**	**7,82**

Com base no quadro 6, pode-se concluir que a linguagem Java possui, aproximadamente, um processamento de dados 10% mais rápido que o Groovy. Considerando que Groovy é uma camada acima de Java, este resultado leva a crer que há rotinas de controle com baixa performance.

Este trabalho não utilizou técnicas de estudos empíricos em engenharia de software[1], as quais podem aprimorar os resultados obtidos. Fica a sugestão para futuras pesquisas.

10. Considerações Finais

Ao iniciar um projeto com o intuito de desenvolver um software voltado para web, uma das questões a serem estabelecidas é: qual linguagem de programação utilizar? Pois esta decisão irá influenciar muitos fatores, dentre eles, a produtividade no desenvolvimento.

[1] *Evidence-Based Software Engineering - http://www.dur.ac.uk/ebse/*

Ao término do desenvolvimento do estudo de caso em ambas as linguagens, foi possível extrair algumas conclusões, tais como: Groovy através de seu *framework* para web (Grails) é 35% mais produtivo que Java/jCompany; Java é aproximadamente 10% mais veloz no processamento das páginas web; Java consome menos de 50% da memória utilizada pelo Groovy, para executar os mesmos casos de uso; Groovy aloca cerca de 2500 classes a mais que Java.

Estes foram os principais resultados das características “dinâmicas”. Quanto às “estáticas” foi possível concluir que a linguagem Groovy se destaca no item produtividade, pois a mesma conta com características marcantes como: expressividade, abstração e capacidade de escrita. Percebe-se claramente que Groovy foi projetada para ser uma linguagem de desenvolvimento ágil, pois conta com inúmeras facilidades e abstrações. Em contrapartida, ao avaliar a linguagem Java notou-se uma linguagem mais madura em relação ao Groovy, onde características como ortogonalidade, legibilidade e confiabilidade destacam o quão Java é estabelecido.

Portanto, tendo os resultados da análise comparativa, resta aos arquitetos e/ou projetistas decidirem por uma linguagem mais produtiva (Groovy) ou uma linguagem com mais performance (Java).

Referências

ASSOCIACAO BRASILEIRA DE NORMAS TECNICAS. NBR ISO/IEC 9126-1: Engenharia de software – Qualidade de produto. Parte 1: Modelo de qualidade. Rio de Janeiro: ABNT, 2003.

ALVIM, Paulo. Aplicações corporativas com jCompany. Belo Horizonte, 2011. Disponível em: <http://jcompany.sourceforge.net>. Acesso em: 15 abr. 2011.

DOEDORLEIN, Osvaldo P. Aprendendo Groovy. Java Magazine, São Paulo, ano IV, n. 32, p. 30-44, jan. 2006.

DICKINSON, Jon. Grails 1.1 web application development. Olton: Packt Publishing, 2009.

JAVA. Independent tests demonstrate write once run anywhere capacibilities of java. Disponível em: <http://www.sun.com/smi/Press/sunflash/1997-09/sunflash.970918.2.xml>. Acesso em: 20 set. 2010.

JUDD, Christopher M.; NUSAIRAT, Joseph F.; SHINGLER, James. Beginning Groovy and Grails. New York: Apress, 2008.

OLIVEIRA, Eric. O Universo dos frameworks Java. [S.l.], 2009. Disponível em: <http://www.linhadecodigo.com.br/artigo/758/O-Universo-dos-Frameworks-Java.aspx>. Acesso em: 20 set. 2010.

PETERSON, Ronny. Introdução ao Grails. [S.l.], 2010. Disponível em: <http://speedydev.wordpress.com/category/desenvolvimento-agil>. Acesso em: 21 set. 2010.

REZENDE, Vandir F. Análise comparativa entre Groovy e Java, aplicado no desenvolvimento web. Trabalho de Conclusão de Curso de Ciências da Computação. Universidade Regional de Blumenau – FURB. Blumenau, 2011.

RUDOLPH, Jason. InfoQ: getting started with Grails. [S.l.], 2007. Disponível em: <http://www.infoq.com/minibooks/grails>. Acesso em: 27 mar. 2010.

SAMPAIO, Antônio B. C. Introdução à ciência da computação. Belém: [s.n.], 1999.

SEBESTA, Robert W. Conceitos de linguagens de programação. 5. ed. Porto Alegre: Bookman, 2003.

TUCKER, Allen B.; NOOMAN, Robert E. Linguagens de programação: princípios e paradigmas. 2. ed. São Paulo: McGraw-Hill, 2008.