UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

ANÁLISE COMPARATIVA ENTRE GROOVY E JAVA, APLICADO NO DESENVOLVIMENTO WEB

VANDIR FERNANDO REZENDE

BLUMENAU
2011

2011/1-36

VANDIR FERNANDO REZENDE

ANÁLISE COMPARATIVA ENTRE GROOVY E JAVA, APLICADO NO DESENVOLVIMENTO WEB

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Marcel Hugo, Mestre - Orientador

BLUMENAU
2011

2011/1-36

ANÁLISE COMPARATIVA ENTRE GROOVY E JAVA, APLICADO NO DESENVOLVIMENTO WEB

Por

VANDIR FERNANDO REZENDE

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Marcel Hugo, Mestre – Orientador, FURB
Membro:	Prof. Mauro Marcelo Mattos, Doutor – FURB
Membro:	Prof. Aurélio Faustino Hoppe, Mestre – FURB

Blumenau, 29 de junho de 2011

Dedico este trabalho a memória de meu pai, Vandir Antoninho e meu irmão Fernando Beckhauser.

AGRADECIMENTOS

À minha mãe, Albertina da Rosa Beckhauser, que apesar das dificuldades da vida, é uma guerreira que jamais deixou de lutar, um exemplo de vida para mim, que me ensinou a lutar pelos meus objetivos, porém sempre trilhando o caminho correto, o caminho do bem e sem dúvida alguma, graças a ela consegui chegar até aqui, um dos passos desta longa caminhada chamada vida.

A Deus, pois sem fé não se chega a lugar nenhum.

À minha irmã, Inara Beckhauser, que junto com o meu padrasto, Irineu Mendes, constituem a minha família, no qual convivemos o nosso dia-a-dia.

Aos meus avós paternos e maternos, onde em ambas famílias serei o primeiro neto a se formar e me orgulho muito disto.

Aos demais familiares, pois a família é a base de nossas vidas.

Aos meus amigos, pelo apoio e motivação.

Ao meu orientador, Marcel Hugo, que ao longo deste trabalho sempre se manteve calmo, disposto a ajudar a encontrar soluções aos problemas que surgiram e principalmente otimista, me entusiasmando e acreditando junto comigo na conclusão deste trabalho.

Por fim, aos entes queridos e amáveis que rezaram e torceram muito pelo sucesso deste.

Nada é permanente, exceto a mudança.

Heráclito

RESUMO

Este trabalho apresenta uma análise comparativa de produtividade, no desenvolvimento de softwares web, entre as linguagens de programação Groovy e Java. Baseando-se na norma NBR-13596 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.

Palavras-chave: Java. Groovy. Comparação. Desenvolvimento web.

ABSTRACT

This paper presents a comparative analysis for Web-based software development productivity, between Java and Groovy programming languages. Based on NBR-13596, were determined evaluation criteria. A study case application was implemented in both languages, in order to measure the productivity gap between them. To compare the development, UCP of use cases was calculated and then checked the time and performance for each language.

Key-words: Java. Groovy. Comparison. Web development.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

ABNT – Associação Brasileira de Normas Técnicas

API – *Application Programming Interface*

ASA – Avaliador de Sites Acadêmicos

ATM – *Asynchronous Transfer Mode*

CPU – *Central Processing Unit*

DSL – *Domain Specific Language*

EF – *Environmental Factor*

GORM – *Groovy Object Relational Mapping*

GSP – *Groovy Server Pages*

GUI – *Graphical User Interface*

HSQLDB – *HyperSQL Database*

HTML – *HyperText Markup Language*

IDE – *Integrated Development Environment*

IP – *Internet Protocol*

ISO – *International Organization for Standardization*

J2EE – *Java Enterprise Edition*

J2SE – *Java Standard Edition*

JDK – *Java Development Kit*

JPA – *Java Persistence API*

JSP – *Java Server Pages*

JVM – *Java Virtual Machine*

LP – Linguagem de Programação

MVC – *Model-View-Controller*

NBR – Norma da Associação Brasileira de Normas Técnicas

OO – Orientação Objeto

PHP – *Pre Hypertext Processor*

TCF – *Technical Complexity Factor*

TCP – *Transmission Control Protocol*

UCP – *Use Case Points*

UML – *Unified Modeling Language*

UUCP – *Unadjusted Use Case Point*

WORA – *Write Once Run Anywhere*

XML – *eXtensible Markup Language*

YAJWF – *Yet Another Java Web Framework*

WAR – *Web Archive*

SUMÁRIO

1 INTRODUÇÃO

A produtividade, considerada uma das mais importantes armas de competição, é a melhor determinante da competitividade e lucratividade das empresas, onde se busca produzir o máximo possível no menor intervalo de tempo (BARBARÁN; FRANCISCHINI, 2008, p. 3). Tempo, este é o principal elemento responsável pela evolução humana, pois constantemente buscam-se métodos para executar as tarefas cotidianas de forma ágil. Um grande exemplo para comprovar este fato é a revolução industrial, onde a produção artesanal foi deixada de lado e investiu-se na criação de máquinas a vapor, que permitiram o desenvolvimento de produtos em larga escala, em um curto intervalo de tempo. Com o passar dos anos e a revolução industrial estabilizada, surgiu uma nova revolução, a tecnológica, que ao invés de máquinas a vapor, utilizou o computador como principal ferramenta de desenvolvimento (VIEIRA, 2007).

Com a necessidade de criar ferramentas que facilitassem o seu trabalho diário, o homem passou a aprimorar cada vez mais os computadores, pois sua utilização não apenas poupa tempo e dinheiro, mas também permite a possibilidade de um controle cada vez melhor de, por exemplo, informações, estoques e serviços (SAMPAIO, 1999, p. 12).

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, onde programas escritos em Java podem ser executados em diferentes ambientes operacionais, dentro do conceito "escreva uma vez, execute em qualquer lugar" (*Write Once, Run Anywhere*), desde que o sistema suporte a máquina virtual java (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. Nota-se que Java contempla soluções para os mais variados desafios no desenvolvimento de aplicações. Para ter esta flexibilidade, Java precisa ser especificamente configurada 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, p. 32).

Rudolph (2007) cita que a marcante característica do Grails em apoiar o desenvolvimento ágil, vem da filosofia de convenção sobre configuração. Tratando-se de um *framework* de alta produtividade baseado em Spring e Hibernate, para desenvolvimento de aplicação web *Java Enterprise Edition* (JEE), sem a necessidade de configurar centenas de arquivos *eXtensible Markup Language* (XML), e tudo isso com a mesma agilidade do Ruby on Rails (CASTELLANI, 2009).

Neste cenário, foi desenvolvida um aplicativo de gestão de projetos, sendo ele desenvolvido em Java e em Groovy, com o intuito de comparar a produtividade entre as linguagens. Tomou-se como base para a análise comparativa a norma NBR-13596 (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 1996). Esta norma define seis características que descrevem, com um mínimo de sobreposição, qualidade de software. Essas características são: funcionalidade, confiabilidade, usabilidade, eficiência, manutenibilidade e portabilidade. No estudo de caso estas características são avaliadas focando produtividade de desenvolvimento.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é realizar uma análise comparativa de produtividade, junto ao desenvolvimento de software web, entre Java e Groovy. Para comparar a produtividade foi implementado o mesmo estudo de caso em ambas as linguagens, o qual é um aplicativo web para gestão de projetos.

Os objetivos específicos do trabalho são:

a) definir os critérios para a análise comparativa, baseados na norma NBR-13596;
b) analisar os casos de uso do aplicativo de gestão de projetos;
c) verificar as diferenças entre Java e Groovy, a partir dos critérios estabelecidos;
d) implementar os casos de uso em ambas as linguagens;
e) desenvolver relatório dos resultados obtidos na análise comparativa, conforme os critérios estipulados segundo a norma NBR-13596.

1.2 ESTRUTURA DO TRABALHO

O texto está estruturado em cinco capítulos. No segundo capítulo é conceituada a fundamentação teórica do trabalho que consiste em apresentar a norma NBR-13596, utilizada como base para a análise comparativa, junto com as características das linguagens que foram observadas para medir a produtividade. O capítulo ainda apresenta as linguagens de programação Java e Groovy, e o *framework* Grails.

No capítulo seguinte, é apresentada a definição dos critérios de avaliação, assim como o desenvolvimento do questionário para a avaliação.

No terceiro capítulo é apresentado o desenvolvimento do estudo de caso, incluído especificação de requisitos, digramas de casos de uso, as ferramentas utilizadas no desenvolvimento, a implementação e a operacionalidade do sistema.

Por fim, é exibido a aplicação do questionário de avaliação, assim como as medições de performance e consumo de memória.

No último capítulo são apresentadas as conclusões e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

No presente capítulo são apresentados alguns aspectos teóricos relacionados ao trabalho. Na seção 2.1 é apresentada a norma NBR-13596 e suas definições. Em seguida, a seção 2.2 trata das características observadas nas linguagens de programação para medir a produtividade destas. A seção 2.3 apresenta uma visão global da linguagem de programação Java. A seção 2.4 aborda a linguagem de programação Groovy e seus conceitos, enquanto a seção 2.5 apresenta o *framework* Grails e suas facilidades que contribuem na produtividade do desenvolvimento de aplicações web. Na seção 2.6 é apresentado o *framework* Scrum, metodologia para gestão de projetos utilizada como base para desenvolver o aplicativo estudo de caso. Finalizando o capítulo, a seção 2.7 descreve os trabalhos correlatos.

2.1 NBR-13596 - TECNOLOGIA DE INFORMAÇÃO: AVALIAÇÃO DE PRODUTO DE SOFTWARE

À medida que as aplicações de software têm crescido, também tem a importância da qualidade de software. Para que se possa gerenciar a qualidade de software, é muito importante a tecnologia para especificar e avaliar tanto a qualidade do produto de software como a qualidade do processo de desenvolvimento, objetiva e quantitativamente. Entretanto, é necessário um modelo que permita a avaliação da qualidade de software. Assim, a NBR-13596 foi desenvolvida como parte de um conjunto de documentos que proporcionam este modelo (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 1996).

Esta norma define seis características que descrevem, com um mínimo de sobreposição, qualidade de software. Estas características fornecem uma base para posterior refinamento e descrição de qualidade de software. As diretrizes descrevem o uso de características de qualidade para a avaliação.

A definição das características, apresentada no quadro 1, e o modelo do processo de avaliação de qualidade correspondente, nesta norma, são aplicáveis na especificação dos requisitos de produtos de software e na avaliação da sua qualidade ao longo do seu ciclo de vida.

CARACTERÍSTICAS	SUBCARACTERÍSTICA	SIGNIFICA
Funcionalidade O conjunto de funções satisfazem as necessidades explícitas e implícitas para a finalidade a que se destina o produto?	Adequação	Propõe-se a fazer o que é apropriado?
	Acurácia	Gera resultados corretos ou conforme acordados?
	Interoperabilidade	É capaz de interagir com os sistemas especificados?
	Segurança de acesso	Evita acesso não autorizado, acidental ou deliberado a programas e dados?
	Conformidade	Está de acordo com normas e convenções previstas em leis e descrições similares?
Confiabilidade O desempenho se mantém ao longo do tempo e em condições estabelecidas?	Maturidade	Com que freqüência apresenta falhas?
	Tolerância a falhas	Ocorrendo falhas como ele reage?
	Recuperabilidade	É capaz de recuperar dados após uma falha?
Usabilidade É fácil utilizar o software?	Inteligibilidade	É fácil entender os conceitos utilizados?
	Apreensibilidade	É fácil aprender a usar?
	Operacionalidade	É fácil de operar e controlar a operação?
Eficiência Os recursos e os tempos utilizados são compatíveis com o nível de desempenho requerido para o produto?	Comportamento em relação ao tempo	Qual é o tempo de resposta e de processamento?
	Comportamento em relação aos recursos	Quanto recurso utiliza?
Manutenibilidade Há facilidade para correções, atualizações e alterações?	Analisabilidade	É fácil encontrar uma falha quando ocorre?
	Modificabilidade	É fácil modificar e remover defeitos?
	Estabilidade	Há grandes riscos de *bugs* quando se faz alterações?
	Testabilidade	É fácil testar quando se faz alterações?
Portabilidade É possível utilizar o produto em diversas plataformas com pequeno esforço de adaptação?	Adaptabilidade	É fácil adaptar a outros ambientes sem aplicar outras ações ou meios além dos fornecidos para esta finalidade no software considerado?
	Capacidade para ser instalado	É fácil instalar em outros ambientes?
	Capacidade para substituir	É fácil substituir por outro software?
	Conformidade	Está de acordo com padrões ou convenções de portabilidade?

Fonte: Gomes (2006).

Quadro 1 – Características da NBR-13596

Suas características podem ser aplicáveis em qualquer tipo de software, incluindo programas e dados de computador contidos em *firmware*. A qualidade de software pode ser

avaliada de acordo com as seguintes características (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS, 1996):

a) funcionalidade: existência de um conjunto de funções e suas propriedades especificadas. As funções são as que satisfazem as necessidades explícitas ou implícitas;
b) confiabilidade: capacidade do software de manter seu nível de desempenho sob condições estabelecidas durante um período de tempo estabelecido;
c) usabilidade: esforço necessário para se poder utilizar o software, bem como o julgamento individual deste uso, por um conjunto de usuários;
d) eficiência: relacionamento entre o nível de desempenho do software e a quantidade de recursos usados, sob condições estabelecidas;
e) manutenibilidade: esforço necessário para fazer modificações especificadas no software;
f) portabilidade: capacidade do software de ser transferido de um ambiente para outro.

2.2 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, p. 519).

As seguintes 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, p. 17).

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, 2006, p. 24). 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, 2006, p. 23). 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, p. 15);
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, 2006, p. 22);
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 (2006, p. 27), 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 inicio 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, 2006, p. 28);

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, p. 17). 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, 2006, p. 29);
i) 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, p. 16). 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, 2006, p. 30);
j) 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, 2006, p. 30);
k) tratamento de exceção: capacidade de uma linguagem constatar que ocorreu algo inesperado e permitir tratar tal situação (SEBESTA, 2006, p.30).

2.3 LINGUAGEM DE PROGRAMAÇÃO JAVA

Segundo Gomes (2009, p. 55), a linguagem Java foi inicialmente desenvolvida por James Gosling na Sun Microsystems e sua primeira versão foi lançada em 1995. Gosling começou a trabalhar no projeto da linguagem em junho de 1991. Esta inicialmente foi chamada de Oak, depois de Green e finalmente Java.

Java é uma linguagem orientada a objetos de última geração que tem uma sintaxe semelhante à de C. Mantendo a linguagem simples, os projetistas também tornaram mais fácil para o programador escrever código robusto e isento de falhas. Como resultado do seu projeto elegante e de recursos de última geração, a linguagem Java tornou-se popular entre os programadores (FLANAGAN, 2006, p. 27).

A grande promessa da linguagem foi a portabilidade, chamada também de "WORA" (*Write Once, Run Anywhere* - escreva um vez e execute em qualquer lugar), isso é, a habilidade da linguagem de poder ser executada em diversas plataformas através da máquina virtual. Assim aplicativos desenvolvidos com Java são compilados e podem ser executados na máquina virtual Java em diferentes plataformas como Microsoft Windows, Mac OSX e Linux.

A portabilidade da linguagem Java pode ser alcançada porque a linguagem, diferente de C e C++, não é compilada em código de máquina, mas ao invés disso é compilada em *bytecode* Java - instruções que são interpretadas por uma máquina virtual - que é escrita especificamente para o hardware que a hospeda (GOMES, 2009, p. 56).

Segundo Cesta (2006), Java possui características herdadas de muitas outras linguagens de programação, tais como: Objective-C, Smalltalk, Eiffel e Modula-3. Java é um caso de sucesso na união de tecnologias testadas por vários centros de pesquisa e desenvolvimento de software. Como Java foi criada para ser usada em computadores pequenos, ela exige pouco espaço, pouca memória. Java é muito mais eficiente que grande parte das linguagens de *scripting* existentes.

2.4 LINGUAGEM DE PROGRAMAÇÃO GROOVY

Groovy é uma linguagem ágil e dinâmica para a plataforma Java, com muitas

ferramentas inspiradas em linguagens como Python, Ruby e Smalltalk, tornando-as disponíveis aos programadores, usando uma sintaxe próxima do Java (KONIG, 2007, p. 3) .

Segundo Konig (2007, p. 3), a linguagem combina as facilidades de uso dos recursos dinâmicos de Ruby e Python com o poder e a estabilidade de Java. Sendo compilada diretamente para *bytecode*, Groovy trabalha em perfeita harmonia com os objetos e as bibliotecas Java existentes. Ao contrário de outras linguagens, que são portadas pela Java *Virtual Machine* (JVM), Groovy foi criada com a JVM em mente, então há pouca ou nenhuma diferença de sintaxe em relação ao Java, reduzindo significativamente a curva de aprendizado (JUDD; NUSAIRAT; SHINGLER, 2008, p. 76).

Strachan (2007), um dos idealizadores do Groovy desde sua concepção, visualizou que seria uma linguagem pequena e dinâmica, compilada diretamente em classes Java, e teria toda a elegância e produtividade encontrada em Ruby e Python, porém permitiria reusar, estender e testar o código Java já existente. O principal objetivo foi criar uma linguagem que se integrasse ao que já havia sido desenvolvido em Java, sem a necessidade de reescrever códigos, e que acelerasse o processo de desenvolvimento.

"Um ponto que deve ficar claro é que Groovy não substitui a linguagem Java e sim complementa, adicionando uma linguagem de desenvolvimento ágil para escrever *scripts* e aplicações com interoperabilidade com a plataforma J2SE e J2EE." (BARROSO, 2006, p.50).

2.5 *FRAMEWORK* GRAILS

Grails é um *framework* de desenvolvimento dinâmico para aplicações web na plataforma JEE. Utiliza-se de codificação por convenção, um paradigma do desenvolvimento de software que visa diminuir o número de decisões que os desenvolvedores precisam tomar, sendo popularizada pelo *framework Ruby on Rails*. Utiliza a flexibilidade do Groovy para fornecer uma *Domain Specific Language* (DSL) para desenvolvimento web. O objetivo é ser capaz de desenvolver aplicações web com o mínimo de esforço, sem ter que repetir comandos. Grails fornece um ambiente consistente e confiável entre todos os seus projetos (DICKINSON, 2009, p. 59).

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 e SHINGLER, 2008, p. 32).

Segundo Smith e Ledbrook (2009, p. 62), Grails foi criado pensando em eliminar a necessidade de efetuar um grande número de configurações e não ser outro *Yet Another Java Web Framework* (YAJWF), através de uma abordagem baseada em convenções. Esta funcionalidade possibilitou os desenvolvedores usar o talento criativo para produzir grandes programas, ao invés de se preocupar com arquivos de configurações e ajustes.

O Grails possui grandes inovações e idéias de ponta, mas sua base é construída sobre bases sólidas de tecnologias de eficiência comprovadas pelo mercado, as quais são o Spring e o Hibernate. Estas tecnologias são muito usadas hoje em aplicações JEE, pelo importante motivo de serem confiáveis e testadas.

Judd, Nusairat e Shingler (2008, p.118) citam outros *frameworks* utilizados pelo Grails, sendo eles:

a) SiteMesh: responsável pelo mecanismo de *layout*, implementa o padrão *decorator* para a renderização de HTML, com componentes constantes nas páginas, como cabeçalho, rodapé e navegação;

b) Ajax: o Grails possui três populares *frameworks* para utilização de Ajax, que atendem as necessidades da Web 2.0, os quais são script.aculo.us, Rico, e o Prototype;

c) Jetty: é um servidor de aplicação popular e rápido, que serve para dar ao Grails um completo ambiente de desenvolvimento;

d) HSQLDB: é um banco de dados 100% Java. É possível configurá-lo para ser executado na memória ou ser mantido em disco. De forma padrão, o Grails executa o banco em memória;

e) JUnit: *framework* utilizado para testes unitários de software. Para toda nova classe de controle criada, o Grails cria uma classe de testes.

f) GORM: importante *framework* de mapeamento objeto relacional, baseado no Hibernate. Mapeia objetos para banco de dados relacional, representando o relacionamento entre as classes. Desenvolvido para utilização em linguagens dinâmicas como Groovy, possui funcionalidades como critérios e buscas dinâmicas.

2.6 METODOLOGIA DE DESENVOLVIMENTO ÁGIL SCRUM

Segundo Pereira (2005, p. 6), a indústria de software é, em sua maioria, formada por pequenas empresas de software, onde estas empresas, por começarem com uma ou duas pessoas, acabam por não adotar um processo de software adequado, tornando custosa a adoção de uma metodologia tradicional de gerência de processo. As metodologias ágeis surgiram para suprir as necessidades dessas empresas, que priorizam as pessoas e os prazos e onde a reescrita do código é uma tarefa simples e rotineira.

De acordo com Schwaber e Beedle (2002, p. 14), o Scrum é um processo interativo incremental de desenvolvimento de software, sendo um modelo empírico de desenvolvimento de software. Utiliza técnicas simples unidas ao senso comum e a uma base de experiências passadas, já testadas e aprovadas. O uso do Scrum vai ao encontro da necessidade das pequenas empresas de software pelo fato de ser simples, ágil e sem muita burocracia.

Esta metodologia trata os papéis a serem desempenhados no decorrer do projeto como sendo apenas três: *Product Owner*, *Scrum Master* e *Scrum Team*. As responsabilidades de todas as atividades do projeto são dividas entre estes papéis. As atividades do Scrum podem ser resumidas como mostrado na figura 1.

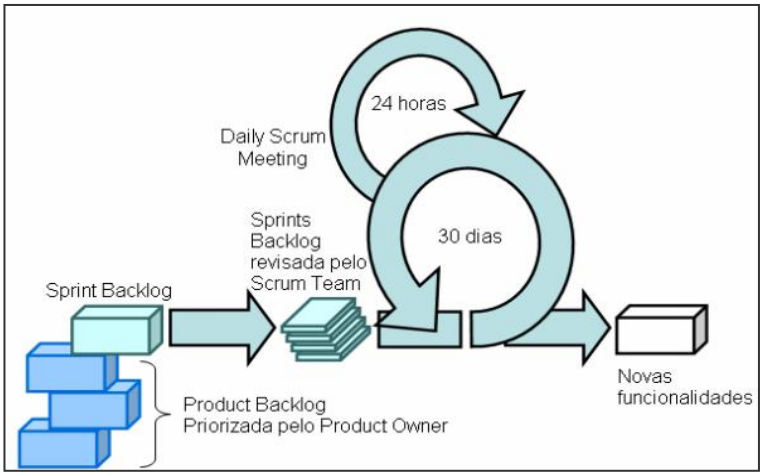

Figura 1 – Modelo de processo baseado em Scrum

Pode-se observar que o papel do *Product Owner* (representante do cliente) é definir a prioridade das tarefas a serem executadas. Isso ocorre através da ordenação do *product backlog* (listagem de tarefas a serem implementadas). A segunda etapa trata-se da definição do *sprint*, período de implementações, que normalmente variam de 2 a 4 semanas, onde o *Scrum Team* (equipe de desenvolvimento) compromete-se a concluir determinadas tarefas do *backlog*, até o término deste período.

Para maiores informações sobre Scrum, pode-se consultar o seu guia, disponível em <http://scrum.org/scrumguides>.

2.7 TRABALHOS CORRELATOS

Dentre os trabalhos encontrados na pesquisa realizada, os relacionados abaixo formam a união dos temas do trabalho proposto. O primeiro aborda o estudo da linguagem Groovy, utilizada no desenvolvimento de uma ferramenta de ensino de programação orientada objeto. Em seguida, é apresentado o trabalho de Russi (2002), o qual utilizou a norma NBR-13596 para avaliar sites acadêmicos. Quanto aos demais trabalhos citados, tratam-se de análises da metodologia Scrum, aplicados em ferramentas web.

2.7.1 RunGroovy: extensão do BlueJ para execução de linhas de código

Müller (2007) propôs uma extensão para a ferramenta BlueJ, permitindo a execução de *scripts* na linguagem Groovy, onde os códigos são interceptados para a criação de objetos e execução de métodos pela biblioteca de extensão do BlueJ, interagindo com a própria *Integrated Development Environment* (IDE). Esta extensão tem como objetivo auxiliar o aprendizado da programação orientada a objetos, facilitando o entendimento de instanciação e integração com objetos.

A extensão desenvolvida adiciona a opção *RunGroovy* ao menu *Tools* da ferramenta BlueJ. Esta opção, ao ser selecionada, abre uma nova janela de console na qual é permitida a digitação de linhas de códigos. Estes comandos podem ser salvos em arquivo no disco e recuperados posteriormente para edição na tela. Toda execução de comando nesta janela será refletida para o ambiente BlueJ com criação de objetos no *ObjectBench*. Nesta janela é possível acessar qualquer objeto que estiver no *ObjectBench* pelo seu nome. Por fim, também é possível a abertura de várias janelas *RunGroovy*, cada uma com o seu *script* para a execução. Na tela de execução de comando, o usuário pode digitar comandos para interação com as classes criadas. Os comandos devem seguir a sintaxe da linguagem Groovy. Após a

digitação destes comandos, o usuário pode executar o código.

2.7.2 Ambiente web para gerenciamento de processo de software baseado no Scrum

Pereira (2005, p. 42) apresenta a adaptação de uma ferramenta web de gerenciamento de projeto conhecida como dotProject para atender os artefatos baseados na metodologia Scrum. O ambiente foi implementado utilizando PHP e MySQL. O ambiente desenvolvido apóia as técnicas adotadas pela metodologia ágil Scrum facilitando no manejo de seus artefatos. Os principais pontos do ambiente são:

a) classificação dos usuários: o ambiente permite a classificação de usuários em *Product Owner*, *Scrum Master* e *Scrum Team*;
b) *product backlog*: o ambiente possui um módulo para gerência do *product backlog*, com uma lista de requisitos do sistema, onde o usuário pode ordenar a lista em grau de importância, assim como definir expectativas de prazos para cada uma das funcionalidades do sistema;
c) *sprint backlog*: o ambiente conta com um módulo para gerência do *sprint backlog*, que contém uma lista das tarefas a serem realizadas, bem como expectativas de tempo definidas pelo próprio *Scrum Team*;
d) *daily Scrum*: o ambiente possui um módulo para registro das *daily Scrum*, onde os integrantes dessa reunião reportam o que farão até a próxima reunião, o que foi desenvolvido desde a última reunião e quais as dificuldades encontradas;
e) avaliações: o ambiente apresenta gráficos de *burn-down* para que o *Scrum Master* possa avaliar o processo.

2.7.3 Avaliação da qualidade de sites acadêmicos baseado na norma NBR 13596

Russi (2002, p. 11) analisou e comparou a qualidade de *sites* acadêmicos, baseando-se na norma NBR-13596, que utiliza características da qualidade e diretrizes para sua aplicação. A coleta e análise dos dados efetuou-se através do ASA, um software desenvolvido para este trabalho, operando em conexão com a internet, utilizando-se de um *check-list* personalizado

de acordo com as preferências e necessidades de cada avaliador.

O software implementado, disponível em <http://campeche.inf.furb.br/tccs/2002-II>, pode ser facilmente utilizado, pois possui um processo baseado em questões que abrangem várias características previstas na norma NBR-13596.

2.7.4 PRONTO! - Software para gestão de projetos ágeis

Gomes (2009, p. 8) propôs criar um software que atenda especificamente às necessidades da empresa, tornando possível o cadastro de solicitações de seus clientes, o gerenciamento destas solicitações através de listas ordenadas por prioridade de execução, acompanhamento do estágio em que as tarefas se encontram e quem são os responsáveis por seu desenvolvimento.

Assim demonstra uma maior transparência do que será entregue nas próximas versões do software e obteve uma otimização do fluxo de trabalho, que envolve os colaboradores, time de atendimento ao cliente, desenvolvedores de software, testadores e diretores da empresa. Gomes (2009, p. 67), detalha passo-a-passo o fluxo de trabalho utilizando o software desenvolvido, seguindo as etapas abaixo:

a) cliente faz solicitação ou informa defeito (nova idéia no *backlog*);
b) *product owner* prioriza o *product backlog* (lista de tarefa prontas a serem escolhidas);
c) reunião de planejamento (equipe estima a complexidade das tarefas e escolhe quais serão entregues);
d) *sprint* acontece (durante uma semana a um mês);
e) acompanhamento através do *kanban* (equipe enxerga em qual etapa está cada tarefa);
f) acompanhamento através do *burndown chart* (equipe compara o trabalho estimado com o realizado).

3 DESENVOLVIMENTO

Este capítulo apresenta o processo de desenvolvimento da análise comparativa, descrevendo as ferramentas e técnicas utilizadas durante sua implementação, onde foram contemplados os seguintes passos:

a) análise e aplicação da norma NBR-13596 nos critérios de avaliação;
b) correlação entre as características das linguagens de programação com os itens da NBR-13596;
c) definição do meio de avaliação dos critérios;
d) especificação dos casos de uso do estudo de caso comparativo;
e) cálculo do UCP para os casos de uso;
f) identificação das diferenças estáticas entre as linguagens Java e Groovy;
g) implementação do estudo de caso em Java;
h) implementação do estudo de caso em Groovy;
i) resultado do questionário de avaliação;
j) cálculo da produtividade por UCP;
k) comparação de desempenho dos estudos de caso gerados.

3.1 NBR-13596

A NBR-13596 fornece um modelo de propósito geral, o qual define seis amplas categorias de características de qualidade de software que são, por sua vez, subdivididas em subcaracterísticas apresentadas no quadro 1.

O modelo proposto pela NBR 13596 tem por objetivo servir de referência básica na avaliação de produto de software. Assim tornou-se necessário adaptar o modelo para a análise comparativa de linguagens de programação (LPS), elencando as características e subcaracterísticas que são relevantes à este tipo comparação, pois a referida norma é o documento técnico que mais se aproxima com o objetivo proposto deste trabalho.

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 (2006) 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.

3.2 CARACTERÍSTICAS DAS LPS *VERSUS* NBR-13596

Com o intuito de quantificar os critérios determinados pela NBR-13596, pode-se correlacionar estes com as principais características das linguagens de programação citadas na obra de Sebesta (2006).

Tendo em vista que ao avaliar determinado critério estabelecido, indiretamente esta avaliando-se um conjunto de características, pois através destas é possível relacionar as linguagens de programação com a norma NBR-13596.

Por exemplo, ao analisar uma linguagem de programação, avaliando sua manutenibilidade, as características que devem ser verificadas são: ortogonalidade, legibilidade e capacidade de escrita, pois ao realizar manutenção em um software, necessita-se ler, compreender e alterar o código fonte, e através destas características será possível determinar o nível de manutenibilidade da linguagem.

Assim sugere-se as seguintes correlações apresentadas no quadro 2.

	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

Quadro 2 – Correlação dos critérios com as características

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-13596.

3.3 MEIO DE AVALIAÇÃO DOS CRITÉRIOS

Estabelecidos 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 foi denominado pelo autor 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 – Apêndice A – avaliando 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 UCP por caso de uso e mensurado o tempo de desenvolvimento destes. Assim, a produtividade está em razão da seguinte função:

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.

3.4 DIFERENÇAS ENTRE JAVA E GROOVY

Para Doederlein (2006, p. 39), 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:

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 (figura 2);
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 (figura 2);
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 (figura 2);
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 (figura 2);

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}`;

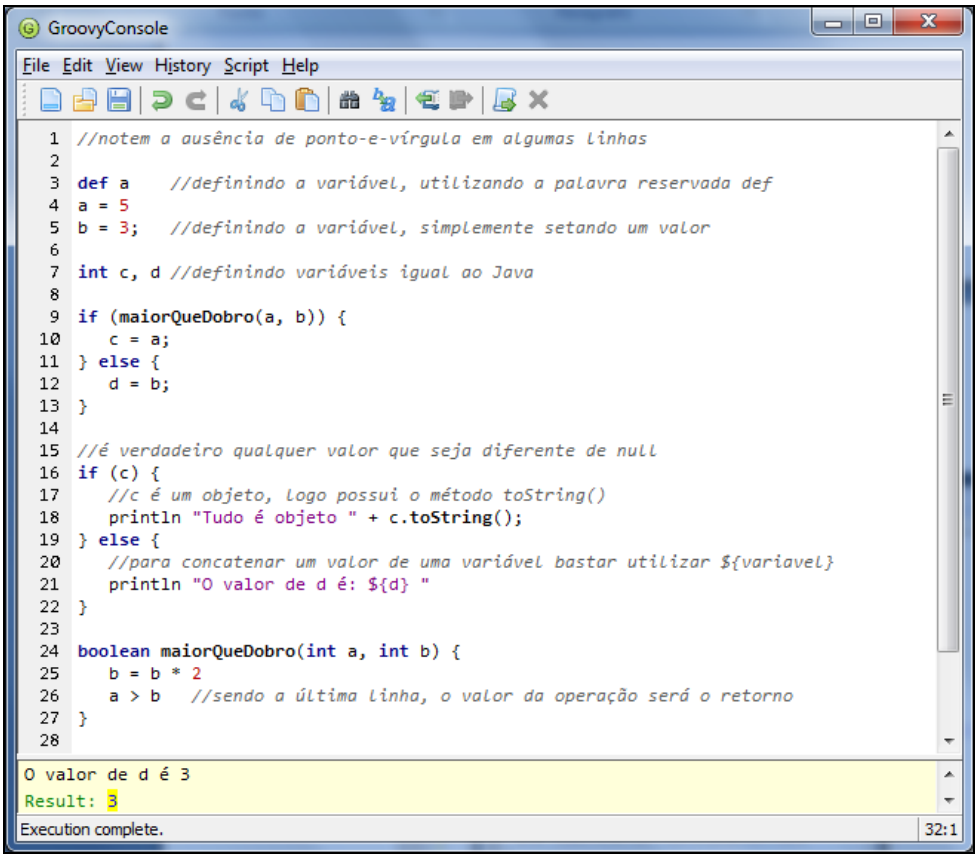

Figura 2 – Semântica Groovy

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, como no exemplo da figura 3.

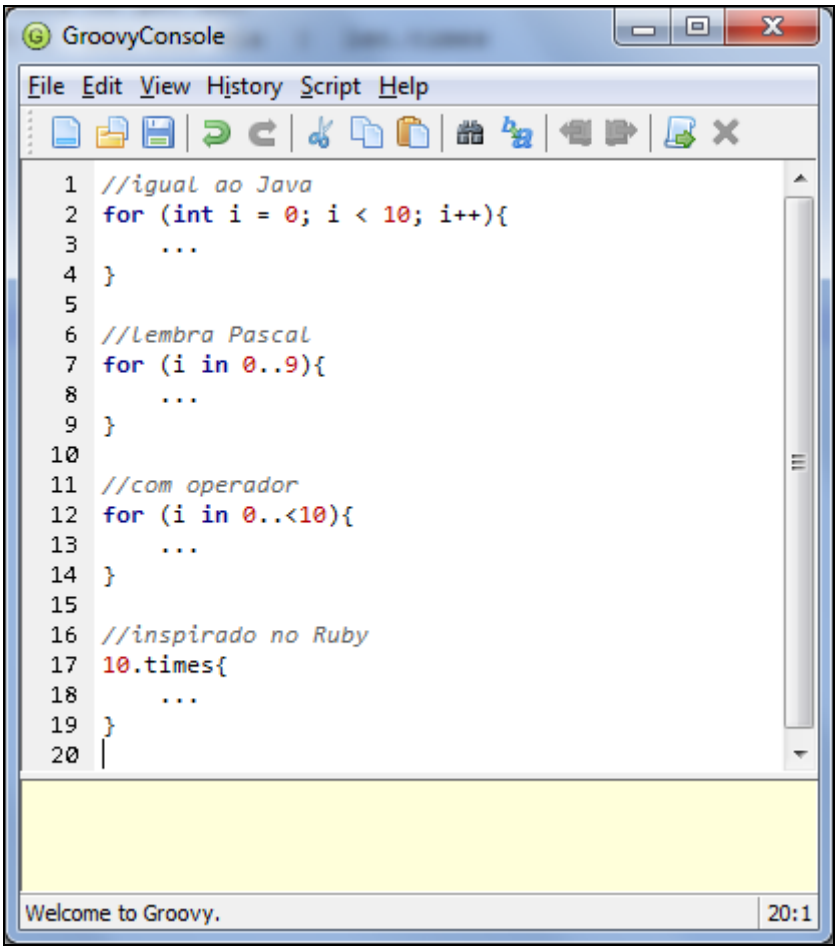

Figura 3 – Comando `for` no Groovy

i) Groovy Beans: para ganhar produtividade no desenvolvimento do código, o Groovy gera dinamicamente os métodos assessores (*getters* e *setters*). Veja a comparação de fontes na figura 4.

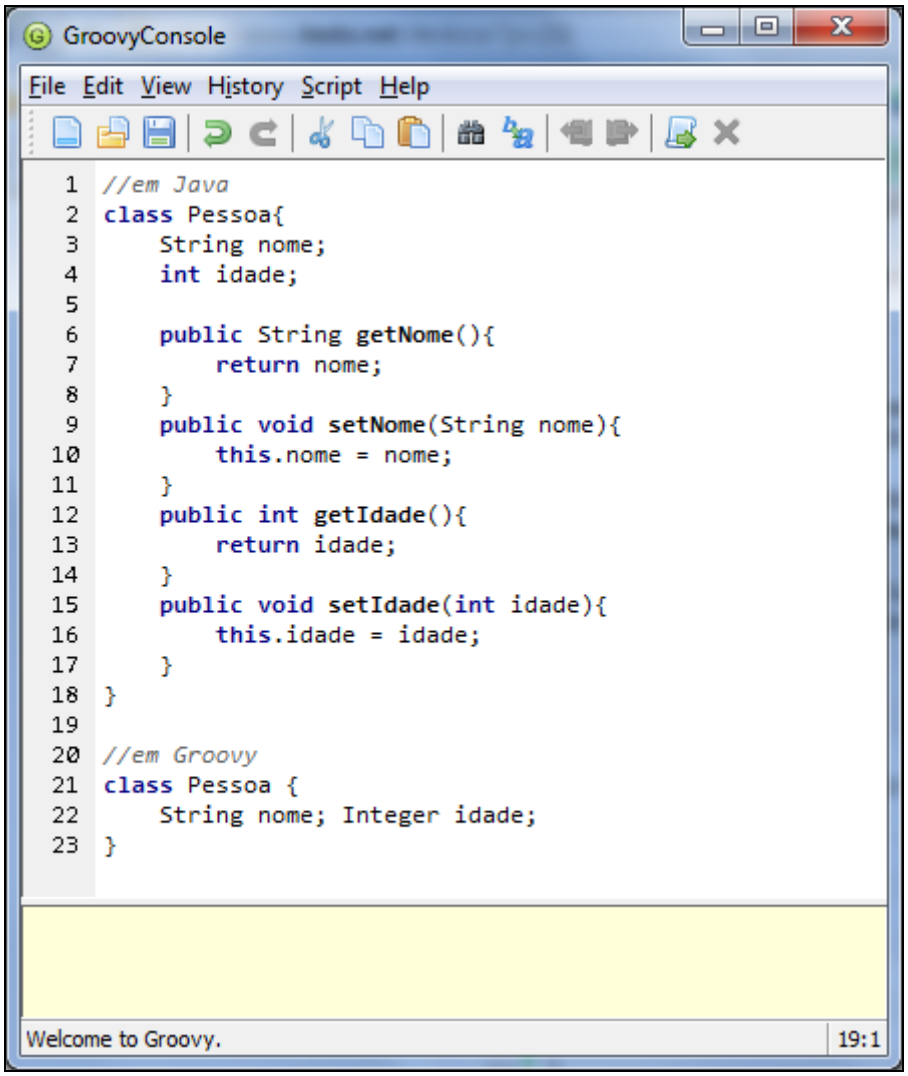

Figura 4 – Diferenças entre o Java e Groovy *beans*

j) GORM: Segundo DICKINSON (2008, p. 147), 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 (figura 5), tais como: `save`, `delele`, `update`, `get` e `findBy`.

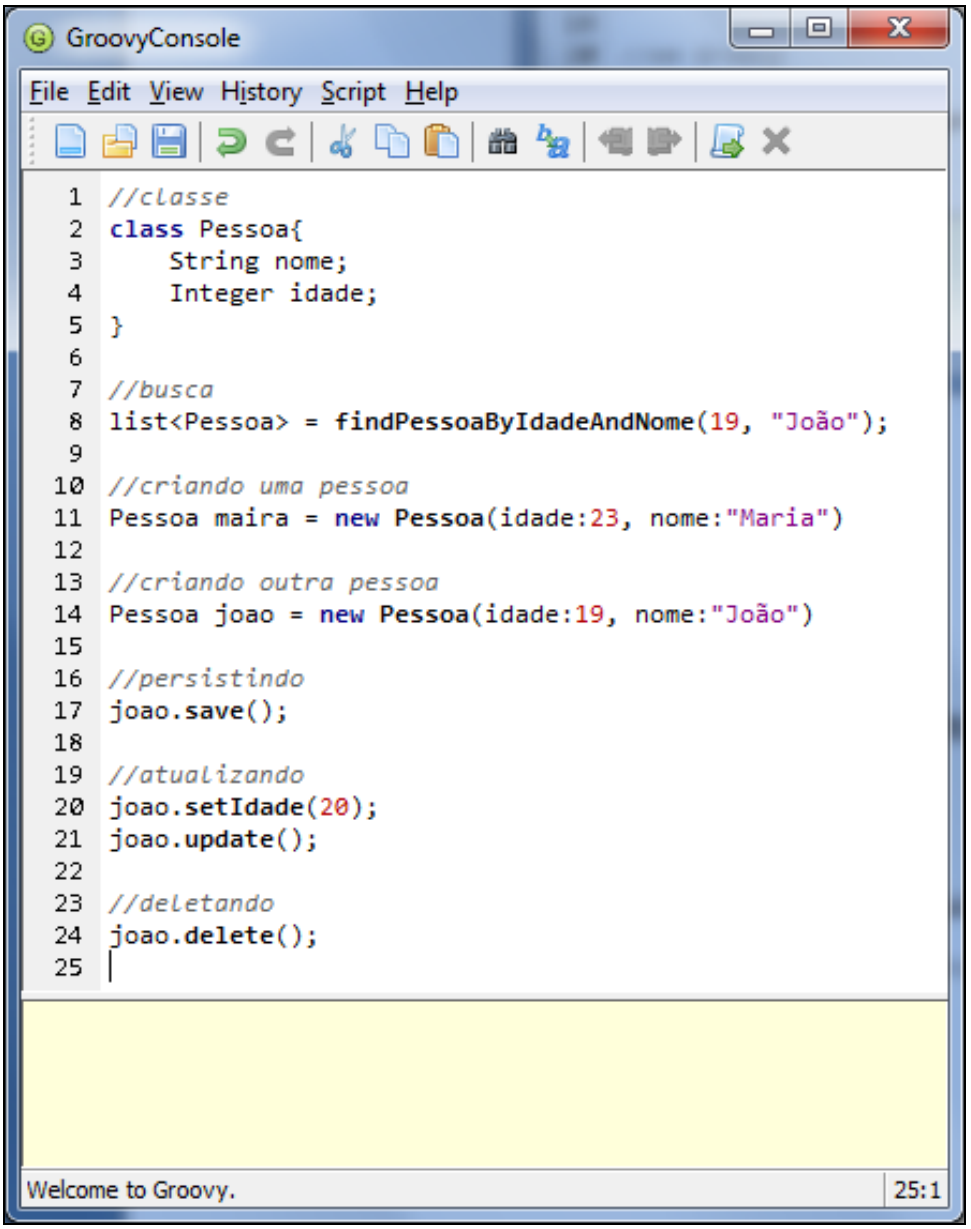

Figura 5 – Métodos GORM

3.5 REQUISITOS PRINCIPAIS DO ESTUDO DE CASO

O levantamento de requisitos consiste na etapa de compreensão do problema aplicado ao desenvolvimento do software e tem como principal objetivo que, tanto desenvolvedor quanto usuários, tenham a mesma visão do problema a ser resolvido pelo sistema (BEZZERA, 2003, p. 21). No quadro 3 são apresentados os requisitos funcionais que definem a

funcionalidade do estudo de caso.

REQUISITOS FUNCIONAIS	CASO DE USO
RF001 – Permitir manter as *sprints* necessárias para o desenvolvimento.	UC001
RF002 – Permitir informar o fator de ajuste relacionado à experiência do *scrum team*.	UC001
RF003 – Permitir estimar a data de término das tarefas relacionadas à *sprint*.	UC001
RF004 – Permitir manter os usuários que terão acesso ao aplicativo.	UC002
RF005 – Permitir distinguir os usuários por seu papel.	UC002
RF006 – Permitir manter as fases de desenvolvimento do projeto/software.	UC003
RF007 – Permitir customizar a quantidade e a nomenclatura das fases de desenvolvimento do projeto/software.	UC003
RF008 – Permitir manter as tarefas, atividades a serem realizadas ao decorrer do projeto/software.	UC004
RF009 – Permitir atribuir tarefas a um *sprint*.	UC004
RF010 – Permitir registrar as atividades executadas para a realização da tarefa.	UC005
RF011 – Permitir visualizar todas as atividades/trâmites relacionados à tarefa.	UC005
RF012 – Permitir informar o tempo gasto de cada atividade executada.	UC006
RF013 – Permitir gerenciar o andamento do *sprint* através de relatórios.	UC007

Quadro 3 – Requisitos funcionais

3.6 ESPECIFICAÇÃO

Conforme Bezerra (2002, p. 14) a UML é uma linguagem visual que serve para modelar sistemas orientados a objetos e é independente de linguagens de programação e processos de desenvolvimento de software. Utilizando a linguagem UML e a ferramenta StarUML foram especificados os diagramas de casos de uso e de classes.

3.6.1 DIAGRAMA DE CASO DE USO

A figura 6 apresenta o diagrama de casos de uso que envolve o ator *Scrum Master*.

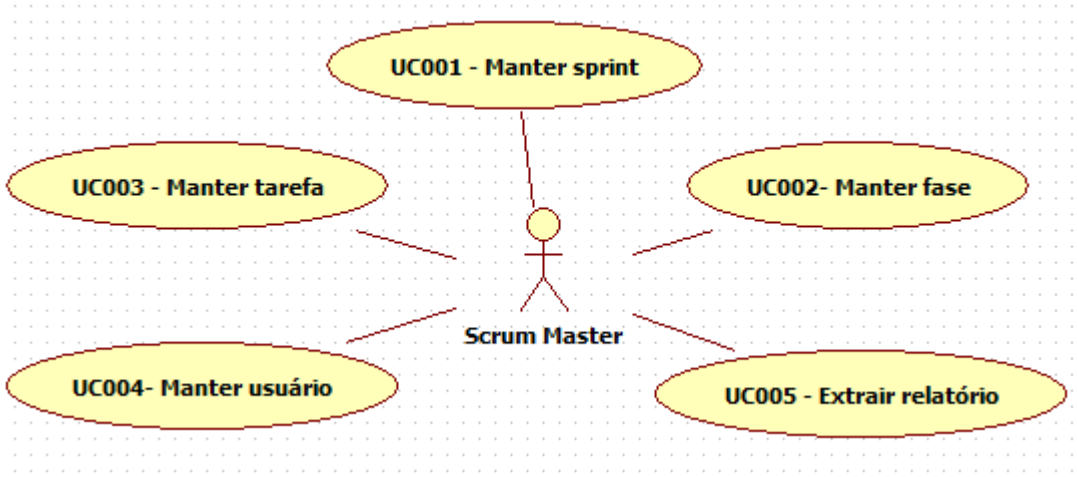

Figura 6 – Casos de uso do *Scrum Master*

A seguir são detalhados os casos de usos, apresentado a sequência de interações entre o sistema e o usuário.

UC001 – Manter *sprint*
Ator: *Scrum Master*
Fluxo principal: Criar *sprint* 1) *Scrum Master* efetua *login* no sistema; 2) *Scrum Master* seleciona a opção de menu *sprint*; 3) Sistema retorna a relação de todos os *sprints* cadastrados; 4) *Scrum Master* seleciona a opção de novo *sprint*; 5) Sistema apresenta formulário com dados a serem informados sobre o *sprint*; 6) *Scrum Master* informa a descrição, o fator de ajuste e o período do *sprint*; 7) *Scrum Master* clica em gravar; 8) Sistema retorna mensagem informado que o *sprint* foi gravado.
Fluxo alternativo: Selecionar *sprint* 3.1) *Scrum Master* seleciona um *sprint* cadastrado; 3.2) Sistema apresenta formulário com os dados do *sprint* selecionado.
Fluxo alternativo: Excluir *sprint* 3.2.1) *Scrum Master* clica em excluir; 3.2.2) Sistema retorna mensagem informando que o *sprint* foi excluído.
Fluxo de exceção: *Sprint* não pode ser excluído 3.2.2.1) Sistema retorna mensagem informando que o *sprint* não pode ser excluído, pois há tarefa relacionadas ao *sprint*.
Fluxo alternativo: Editar *sprint* 3.2.1) *Scrum Master* edita as informações do *sprint* selecionado; 3.2.2) *Scrum Master* clica em grava; 3.2.3) Sistema retorna mensagem informada que o *sprint* foi gravado.

Quadro 4 – Manter *sprint*

UC002 – Manter fase
Ator: *Scrum Master*
Fluxo principal: Criar fase 1*) Scrum Master* efetua *login* no sistema; 2) *Scrum Master* seleciona a opção de menu fases de desenvolvimento; 3) Sistema retorna a relação de todos as fases cadastradas; 4) *Scrum Master* seleciona a opção de nova fase; 5) Sistema solicita o nome da fase; 6) *Scrum Master* informa o nome da fase; 7) *Scrum Master* clica em gravar; 8) Sistema retorna mensagem informado que a fase foi gravada.
Fluxo alternativo: Selecionar fase 3.1) *Scrum Master* seleciona um usuário cadastrado; 3.2) Sistema apresenta formulário com os dados do usuário selecionado.
Fluxo alternativo: Excluir fase 3.2.1) *Scrum Master* clica em excluir; 3.2.2) Sistema retorna mensagem informando que a fase foi excluída.
Fluxo de exceção: Fase não pode ser excluída 3.2.2.1) Sistema retorna mensagem informando que a fase não pode ser excluída, pois há tarefa relacionadas a fase.
Fluxo alternativo: Editar fase 3.2.1) *Scrum Master* edita as informações da fase selecionada; 3.2.2) *Scrum Master* clica em grava; 3.2.3) Sistema retorna mensagem informada que a fase foi gravada.

Quadro 5 – Manter fase

UC003 – Manter tarefa
Ator: *Scrum Master*
Fluxo principal: Criar tarefa 1*) Scrum Master* efetua *login* no sistema; 2) *Scrum Master* seleciona a opção de menu tarefa; 3) Sistema retorna a relação de todos as tarefas cadastradas; 4) *Scrum Master* seleciona a opção de nova tarefa; 5) Sistema apresenta formulário com dados a serem informados sobre a tarefa; 6) *Scrum Master* informa a descrição da tarefa e demais informações; 7) *Scrum Master* clica em gravar; 8) Sistema retorna mensagem informado que a tarefa foi gravada.
Fluxo alternativo: Selecionar tarefa 3.1) *Scrum Master* seleciona uma tarefa cadastrada; 3.2) Sistema apresenta formulário com as informações da tarefa.
Fluxo alternativo: Excluir tarefa 3.2.1) *Scrum Master* clica em excluir; 3.2.2) Sistema retorna mensagem informando que a tarefa foi excluída.
Fluxo de exceção: Tarefa não pode ser excluída 3.2.2.1) Sistema retorna mensagem informando que a tarefa não pode ser excluída, pois há trâmites relacionados a tarefa.
Fluxo alternativo: Editar tarefa 3.2.1) *Scrum Master* edita as informações da tarefa selecionada; 3.2.2) *Scrum Master* clica em grava; 3.2.3) Sistema retorna mensagem informada que a tarefa foi gravada.

Quadro 6 – Manter tarefa

UC004 – Manter usuário
Ator: *Scrum Master*
Fluxo principal: Criar usuário 1*)* *Scrum Master* efetua *login* no sistema; 2) *Scrum Master* seleciona a opção de menu usuário; 3) Sistema retorna a relação de todos os usuários cadastrados; 4) *Scrum Master* seleciona a opção de novo usuário; 5) Sistema apresenta formulário com dados a serem informados sobre o usuário; 6) *Scrum Master* preenche os dados do usuário e informa o seu papel; 7) *Scrum Master* clica em gravar; 8) Sistema retorna mensagem informado que o usuário foi gravado.
Fluxo alternativo: Selecionar usuário 3.1) *Scrum Master* seleciona um usuário cadastrado; 3.2) Sistema apresenta formulário com os dados do usuário selecionado.
Fluxo alternativo: Excluir usuário 3.2.1) *Scrum Master* clica em excluir; 3.2.2) Sistema retorna mensagem informando que o usuário foi excluído.
Fluxo de exceção: Usuário não pode ser excluído 3.2.2.1) Sistema retorna mensagem informando que o usuário não pode ser excluído, pois há lançamentos relacionados ao usuário.
Fluxo alternativo: Editar usuário 3.2.1) *Scrum Master* edita as informações do usuário selecionado; 3.2.2) *Scrum Master* clica em grava; 3.2.3) Sistema retorna mensagem informada que o usuário foi gravado.

Quadro 7 – Manter usuário

UC005 – Extrair relatório
Ator: *Scrum Master*
Fluxo principal: Gerar relatório 1) *Scrum Master* efetua *login* no sistema; 2) *Scrum Master* seleciona a opção de menu relatórios; 3) Sistema apresenta os relatórios disponíveis; 4) *Scrum Master* seleciona o relatório desejado; 5) Sistema apresenta formulário solicitando os filtros do relatório; 6) *Scrum Master* delimita os dados que o relatório irá exibir; 7) *Scrum Master* clica em gerar; 8) Sistema gera o relatório em PDF e exibe ao *Scrum Master*.
Fluxo alternativo: Sem dados a serem listados 7.1) Sistema retorna mensagem informando que há dados a serem listados, com os filtros utilizados.

Quadro 8 – Extrair relatório

A figura 7 apresenta o diagrama de casos de uso que envolve o ator *Scrum Team*.

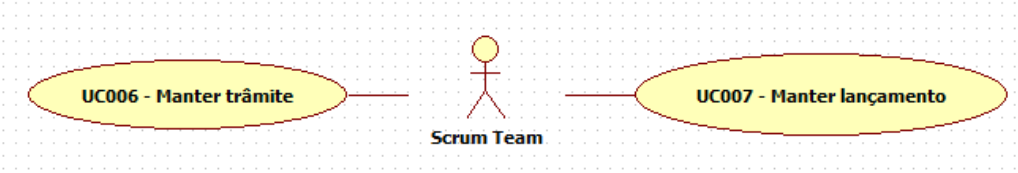

Figura 7 – Casos de uso do *Scrum Team*

UC006 – Manter trâmite
Ator: *Scrum Team*
Fluxo principal: Criar trâmite 1) *Scrum Team* efetua *login* no sistema; 2) *Scrum Team* seleciona a opção de menu tarefa; 3) Sistema retorna a relação de tarefas cadastradas; 4) *Scrum Team* seleciona uma tarefa cadastrada; 5) *Scrum Team* clica em adicionar trâmite; 6) Sistema apresenta formulário para cadastrar novo trâmite; 7) *Scrum Team* descreve todas as atividades, realizadas na tarefa durante a período; 8) *Scrum Team* clica em gravar. 9) Sistema apresenta mensagem dizendo que o trâmite foi gravado.

Quadro 9 – Manter trâmite

UC007 – Manter lançamento
Ator: *Scrum Team*
Fluxo principal: Criar lançamento 1) *Scrum Team* efetua *login* no sistema; 2) *Scrum Team* seleciona a opção de menu tarefa; 3) Sistema retorna a relação de tarefas cadastradas; 4) *Scrum Team* seleciona uma tarefa cadastrada; 5) Sistema retorna os dados da tarefa assim como os trâmites relacionados a tarefa; 6) *Scrum Team* seleciona um trâmite e clica em adicionar lançamento; 7) Sistema apresenta formulário contendo a data inicial e final do lançamento; 8) *Scrum Team* informa as datas e clica em gravar; 9) Sistema apresenta mensagem dizendo que os lançamentos foram gravados.
Fluxo alternativo: Validação de datas 8.1) Sistema retorna mensagens informando que as datas não são válidas.

Quadro 10 – Manter lançamento

3.6.2 DIAGRAMA DE CLASSE

O diagrama de classes é utilizado para descrever os aspectos estruturais estáticos de um sistema orientado a objetos, onde descreve as classes com seus métodos, atributos e os relacionamentos entre estas, assim o diagrama de classes é utilizado na construção de modelo de classes desde o nível de análises até o nível de especificação (Bezerra, 2002, p. 97).

A figura 8 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.

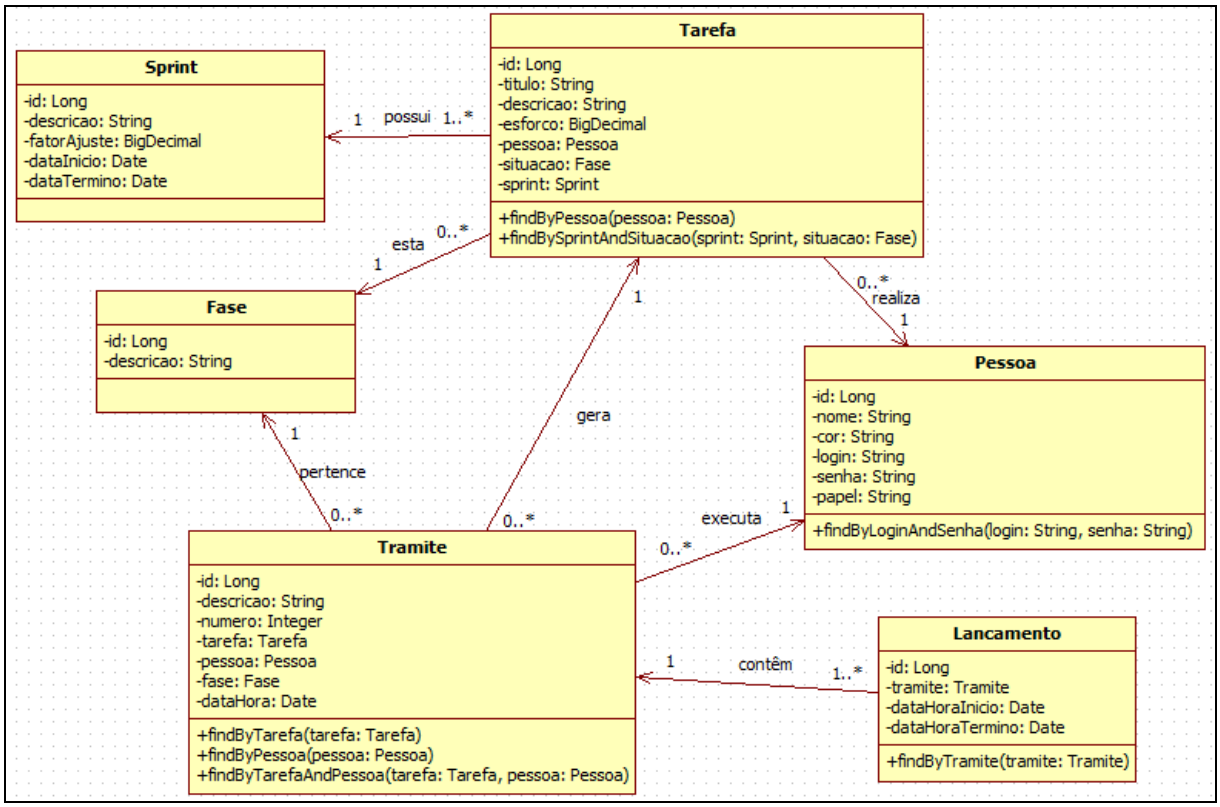

Figura 8 – Diagrama de classes do estudo de caso

3.7 CÁLCULOS DE UCP

Os *use case points* são originados a partir de uma métrica de sistemas computacionais amplamente utilizada que é a de pontos por função (VAZQUEZ; SIMÕES; ALBERT, 2003, p.157). Tal como os pontos por função, os *use case points* permitem que a complexidade do sistema seja mensurada a partir dos seus requisitos funcionais. Além disso, podem ser medidas as complexidades de partes do sistema, e em diferentes níveis de decomposição funcional e abstração.

Segundo Karner (1993 apud CELEPAR,2000), os *use case points* são calculados da seguinte maneira:

a) A complexidade de cada ator é avaliada, onde é atribuído um peso;
b) Estes pesos são somados, obtendo o peso não ajustado dos atores;
c) A complexidade de cada caso de uso é avaliada, onde é atribuído um peso;
d) Estes pesos também são somados, obtendo o peso não ajustado dos casos de uso;

e) Estes valores são somados, obtendo o valor para os *use case points* não ajustados;
f) A este valor são aplicados os fatores de ajuste técnico e ambiental, obtendo-se assim o valor para os *use case points*.

Como base para o cálculo dos *use case points* do estudo de caso, utilizou-se o *framework* Pinhão Paraná (CELEPAR, 2000), onde com os valores das respostas do questionário – Anexo A – obtêm-se o tamanho estimado dos casos de uso.

CASO DE USO	UCP
UC001 - Manter *sprint*	13,6
UC002 - Manter fase	13,6
UC003 - Manter tarefa	19,7
UC004 - Manter usuário	19,7
UC005 - Extrair relatório	25,9
UC006 - Manter trâmite	19,7
UC007 - Manter lançamento	25,9

Quadro 11 – Valor da UCP por caso de uso

3.8 IMPLEMENTAÇÃO

Nesta seção são apresentadas as técnicas utilizadas para a implementação, ferramentas que auxiliaram o desenvolvimento e o funcionamento do estudo de caso implementado.

3.8.1 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, baseada na integração de dezenas de *frameworks*, líderes em seus segmentos, tais como Struts, Tiles, Hibernate, Log4j, Apache Commons e outros, identificando padrões de alto nível e aplicando generalizações orientadas objetos em uma arquitetura MVC que resultam em um *framework* com alto nível de abstração (ALVIM,

2011).

O jCompany utiliza a IDE Eclipse para a edição dos códigos fontes Java e desenvolvimento dos seus projetos, além disto o *framework* incorpora *wizards* que auxiliam os programadores na fase de desenvolvimento do software (figura 9).

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.

As informações configuradas através dos *wizards* são gravadas em arquivos XML, para o programador posteriormente dar manutenção no código gerado, pois o *framework* possui assistente apenas para gerar e configurar as telas inicialmente, mas para realizar manutenção, as alterações nos códigos têm que ser feitas manualmente.

Para o desenvolvedor desfrutar dos assistentes disponibilizados, as classes Java da camada de modelo obrigatoriamente têm que estender de uma classe específica (figura 10) da base do *framework*. Havendo relacionamento entre as classes bases, depois de realizado a seqüência de passos do *wizard*, se faz necessário a implementação manual da inicialização destas.

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.

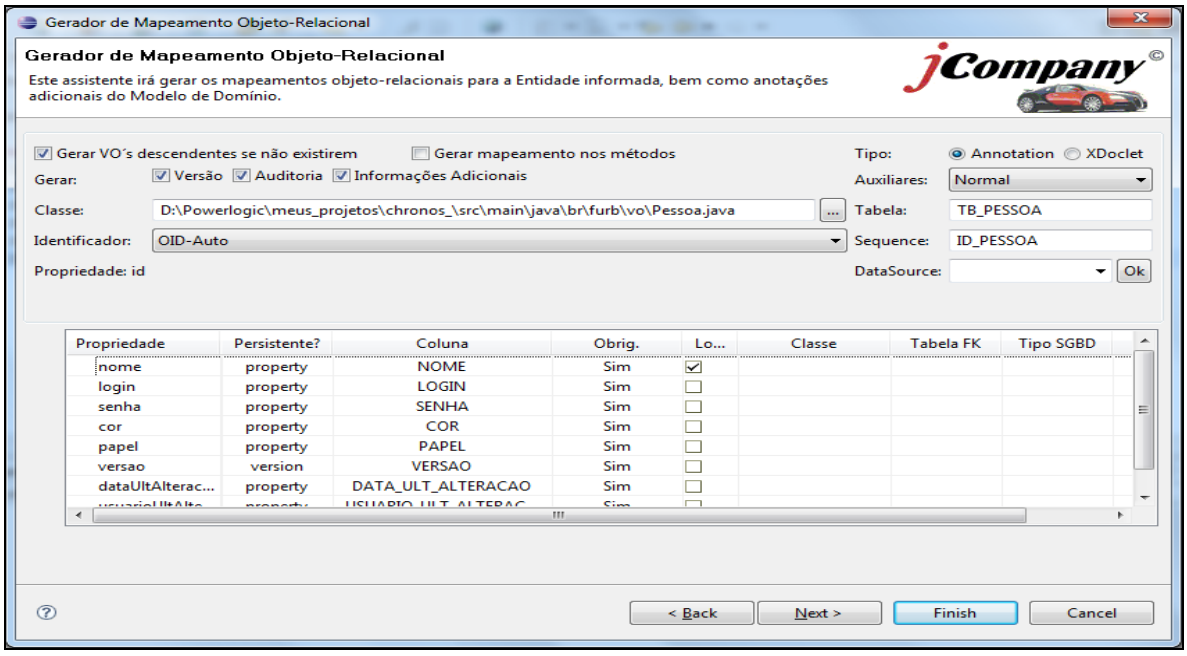

Figura 2 – *Wizard* jCompany para mapeamento de classes

```
package br.furb.vo;

import java.math.BigDecimal;

@MappedSuperclass
public abstract class Tarefa extends PlcBaseVO {

    @Id @GeneratedValue(strategy=GenerationType.AUTO, generator = "ID")
    @Column (name = "ID", nullable=false)
    private Long id;

    @Column (name = "TITULO", nullable=false)
    private String titulo;

    @Column (name = "DESCRICAO", nullable=false)
    private String descricao;

    @Column (name = "ESFORCO", nullable=false)
    private BigDecimal esforco;

    @SuppressWarnings("unchecked")
    @ManyToOne (targetEntity = PessoaVO.class, fetch = FetchType.EAGER)
    @JoinColumn (name = "ID_PESSOA", nullable=false)
    private Pessoa pessoa;

    @SuppressWarnings("unchecked")
    @ManyToOne (targetEntity = SituacaoVO.class, fetch = FetchType.EAGER)
    @JoinColumn (name = "ID_SITUACAO", nullable=false)
    private Situacao situacao;

    @SuppressWarnings("unchecked")
    @ManyToOne (targetEntity = SprintVO.class, fetch = FetchType.EAGER)
    @JoinColumn (name = "ID_SPRINT", nullable=false)
    private Sprint sprint;
```

Figura 3 – Exemplo de mapeamento de classe utilizando jCompany

3.8.2 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`. O resultado deve ser similar a figura 11.

Figura 4 – Teste da instalação do Groovy

Portanto, para configurar o ambiente de desenvolvimento Groovy, basta efetuar o *download* do Grails, descompactá-lo em uma pasta e configurar a variável de ambiente `GRAILS_HOME`.

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 a estrutura representada no quadro 12;

```
%PROJECT_HOME%
+ grails-app
   + conf              Mantém as configurações gerais da aplicação
      + hibernate         Mantém as configurações de acesso a banco de dados
      + spring            Mantém as configurações opcionais do Spring
   + controllers       Mantém as classes de controle
   + domain            Mantém as classes de domínio
   + i18n              Mantém as classes de internacionalização
   + services          Mantém as classes de serviços (equivalente a local
                       session bean)
   + taglib            Mantém as bibliotecas de tags
   + views             Mantém os templates de visão, com subdiretórios para cada
                       classe de controle.
      + layouts           Mantém os layouts de templates
   + grails-tests      Mantém as classes de testes unitários
   + lib               Mantém as bibliotecas extras necessárias para a aplicação
   + src               Mantém arquivos fontes adicionais
      + groovy            Mantém arquivos Groovy adicionais
      + java              Mantém arquivos Java Adicionais
   + web-app
      + css               Mantém as folhas de estilo da aplicação
      + images            Mantém os arquivos de imagens
      + js                Mantém as bibliotecas javascrips
      + WEB-INF           Mantém os arquivos de deploy da aplicação
```

Quadro 12 – Estrutura de um projeto Groovy

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`, conforme figura 12. Este comando irá gerar todos os eventos padrões da classe de controle, tais como: criar, listar, editar e excluir (CRUD);

```
package br.furb

class SprintController extends BaseController{
    def scaffold = true
}
```

Figura 5 – Exemplo de uma *controller* utilizando *scaffolding*

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 as suas configurações por convenção, que simplifica o processo e permite o desenvolvedor se preocupar apenas com as regras de negócio.

3.9 RESULTADOS E DISCUSSÃO

Neste capítulo serão apresentados os resultados obtidos ao término do desenvolvimento do aplicativo estudo de caso e da analise das características de cada linguagem.

3.9.1 RESULTADO DO QUESTIONÁRIO DE AVALIAÇÃO

Para realizar a análise comparativa das linguagens de programação, Groovy e Java, foi preenchido o questionário apresentado no quadro 13, 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. As respostas 0, 5 e 10 significam respectivamente “Não”, “Parcial” e “Sim”.

	GROOVY			JAVA		
	0	5	10	0	5	10
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
A LP sugere a utilização de padrões de projetos?			X			X
A LP possui uma grande quantidade de recursos nativos?		X			X	
É fácil dominar todos os recursos disponibilizados pela LP?		X			X	
Existem atualizações da linguagem, adicionados novos recursos ou é uma LP consolidada e sem alteração?			X		X	
Há uma única forma para cada construção, ou cada construção pode ser representada por diversas formas?	X				X	
A codificação se aproxima da linguagem natural?		X			X	
Existe um padrão na formação de expressões, ou há exceções?		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	
A LP possui palavras reservadas? É possível utilizá-las como nome de variáveis?			X			X
Existe limite de caracteres ao definir uma variável?			X			X
As variáveis necessitam de um tipo?	X					X
É possível utilizar o comando GOTO? Ou alguma derivação do comando?	X			X		
A LP possui facilidades para codificação que geram grande resultado computacional?			X		X	
A LP permite a aplicação das técnicas de abstração como Herança e polimorfismo?			X			X
A LP possui herança múltipla?	X			X		
A LP disponibilidade uma grande quantidade de bibliotecas prontas ao programador?			X			X
Existe o conceito de ponteiros na LP?	X			X		
A LP o desenvolvedor criar seus próprios tipos de dados?			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
A LP é totalmente orientada objetos ou a LP possui tipos primitivos?			X		X	
Há a verificação de tipo dinamicamente? Antes de compilar ou ao compilar?		X				X
A LP é fortemente tipada?	X				X	
Existe o tratamento de exceção na LP?			X			X
A LP permite tratar diferentes tipos de exceções de formas distintas?			X			X
Os programas gerados podem executar em qualquer plataforma?			X			X
Há um conjunto de configurações iniciais pré-estabelecidas?			X		X	

A LP é facilmente adaptável para o desenvolvimento do que se propõem?			X		X	

Quadro 13 – Questionário de Avaliação – Apêndice A

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, o qual auxiliou as análises das linguagens, foi possível concluir que algumas características se destacam mais em determinada linguagem (quadro 14).

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	

Quadro 14 – Comparativo de características

Ao responder o questionário, notou-se grande semelhança entre as linguagens, porém no quadro 14 pode-se observar que Groovy evoluiu sua linguagem aprimorando recursos em busca de produtividade (URUBATAN, 2010). 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.

3.9.2 CÁLCULO DA PRODUTIVIDADE POR UCP

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 15

apresenta o resultado da função de produtividade (tempo / UCP) de cada linguagem.

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**

Quadro 15 – Produtividade por UCP

Ao analisar os resultados obtidos (quadro 15), 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.

Outro fato que deve ser observado é a diferença do tempo gasto, no caso de uso UC005 – Extrair relatório, onde em Java foi possível reaproveitar o fonte escrito em Groovy, pois em ambas as linguagens a execução de relatórios é executada através de chamada de bibliotecas Jasper Reports, assim o tempo contabilizado em Java, foi o envolvido para a adaptação das devidas classes, portanto os valores deste caso de uso não foram sumarizados.

O tempo para a implementação dos relatórios não foram contabilizados, devido ao fato que cada relatório desenvolvido pode ser executado em ambas as linguagens, diferindo apenas a chamada destes.

3.9.3 COMPARATIVO DE DESEMPENHO DOS ESTUDOS DE CASO

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 os seguintes testes:

a) processamento: 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 (figura 13). Assim aplicou-se o seguinte método:
    1. para cada caso de uso, mensurar o tempo de processamento (em milissegundos) dos eventos gerar, listar e apagar;

2. executar cada evento três vezes, extrair a média aritmética e calcular o resultado no quadro 16;

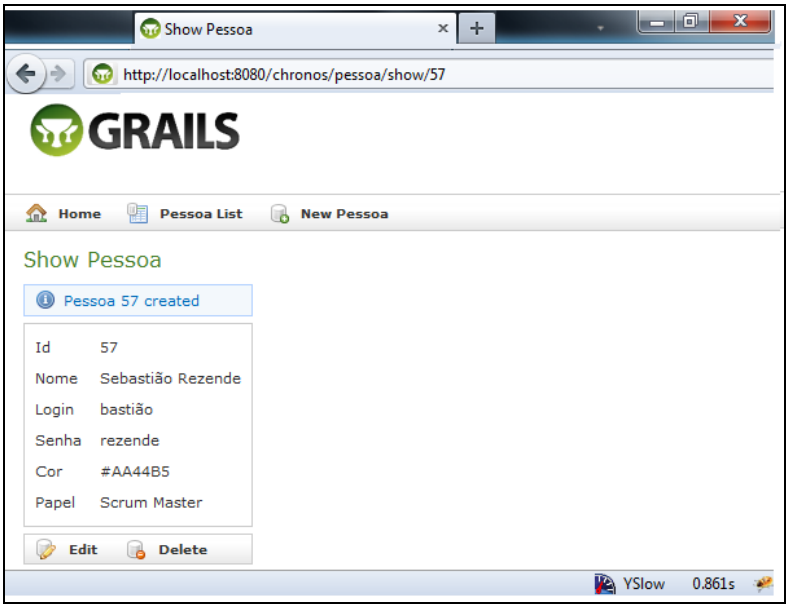

Figura 6– Tomada de tempo na criação de um usuário

3. após executados todos os eventos de cada caso de uso em ambas linguagens, foi calculada a média de cada caso de uso e calculado o porcentual de diferença entre as linguagens.

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**

Quadro 16 – Processamento dos casos de uso em milissegundos

Com base no quadro 16, 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.

b) recursos: para verificar o quanto de *hardware* cada aplicativo consumiu, utilizou-se

a ferramenta jvisualvm, que vem acompanhada na instalação da JDK (*Java Developement Kit*) e foram executados os seguintes passos:

1. servidor: foi configurado o serviço tomcat para iniciar com 128MB de memória;
2. aplicativo: em cada linguagem foi gerado o arquivo .war e inicializado no serviço tomcat individualmente;
3. gráfico: estando o serviço inicializado, foram gerados os gráficos pela ferramenta jvisualvm, conforme figura 14 (Groovy) e figura 15 (Java).

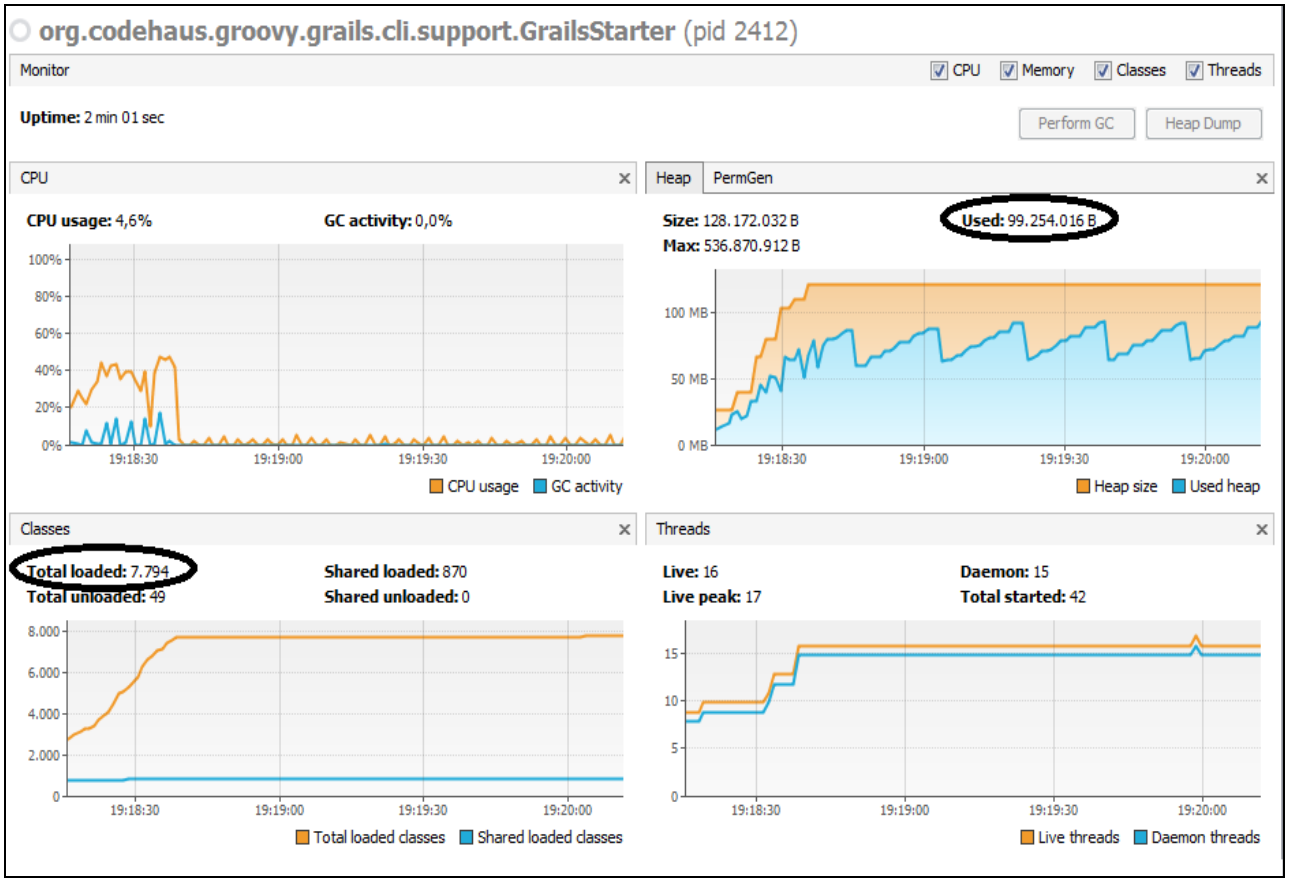

Figura 7 – Gráfico de memória consumida pelo Groovy

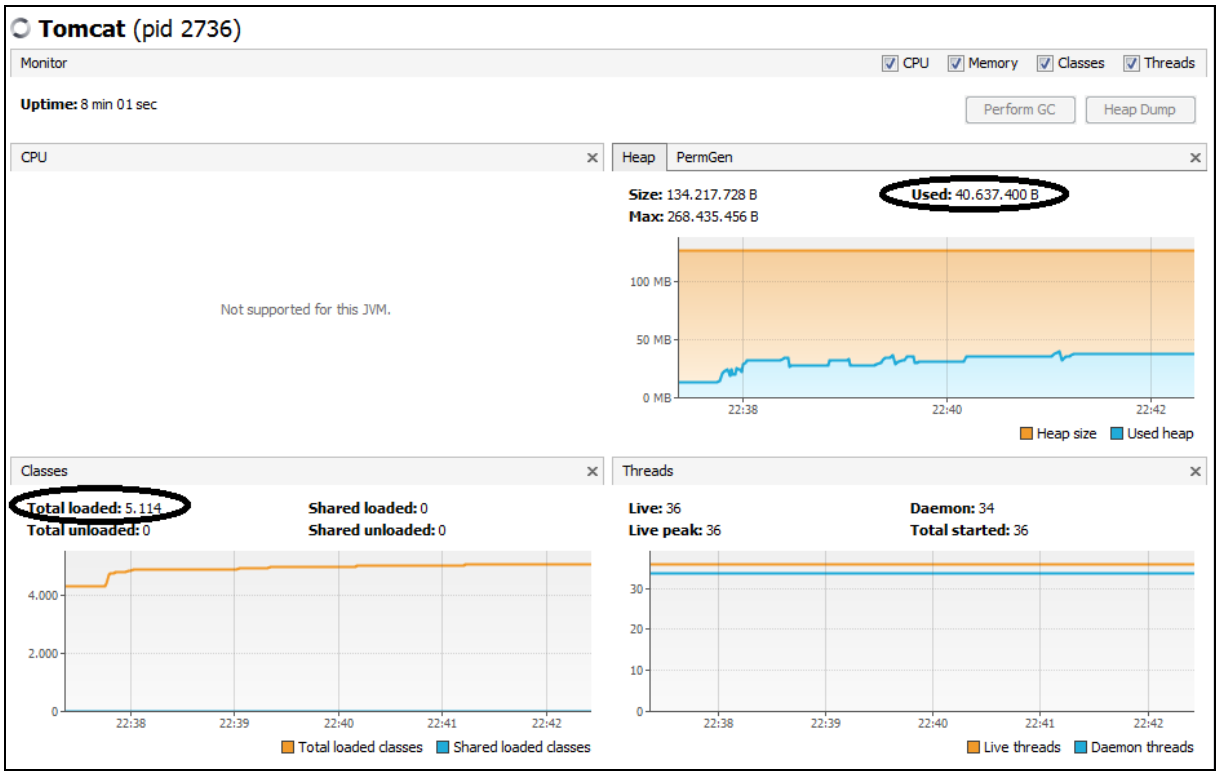

Figura 8 – Gráfico de memória consumida pelo Java

Analisando os gráficos contidos nas figuras 13 e 14, constata-se que Groovy utiliza 99MB de memória para processar seus casos de uso, contra 40MB do Java, ou seja, Groovy inicialmente consome mais que o dobro de memória para executar as mesmas rotinas do Java. Ainda analisando os gráficos, observa-se que Groovy possui aproximadamente 2500 classes carregadas a mais que Java, devido ao Groovy carregar várias APIs ao iniciar sua aplicação e também possuir mais classes de controle (KONIG, 2007, p. 265).

4 CONCLUSÕES

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.

Com o objetivo de auxiliar esta tomada de decisão, foi desenvolvida uma análise comparativa de produtividade entre Groovy e Java, iniciando-se pela definição de critérios de avaliação baseados na norma NBR-13596, que estabelece os itens a serem considerados durante o desenvolvimento de um software. Apesar desta norma não ser voltada à avaliação de linguagens de programação, ela foi importante para direcionar as características a serem observadas nas linguagens.

Estabelecidos os critérios da avaliação, estes foram correlacionados com as principais características das linguagens de programação, visando poder quantificá-las para posteriormente compará-las. Assim, notaram-se dois grupos distintos de características, denominados de “estáticos” e “dinâmicos”, onde os estáticos foram avaliados através de um questionário produzido pelo o autor e os dinâmicos pelo desenvolvimento de um aplicativo estudo de caso.

Ao término do desenvolvimento do estudo de caso em ambas as linguagens, foi possível extrair algumas conclusões, tais como:

a) Groovy através de seu *framework* para web (Grails) é 35% mais produtivo que Java/jCompany;
b) Java é aproximadamente 10% mais veloz no processamento das páginas web;
c) Java consome menos de 50% da memória utilizada pelo Groovy, para executar os mesmos casos de uso;
d) 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).

4.1 EXTENSÕES

Sugere-se as seguintes extensões deste trabalho:

a) analisar a arquitetura dos projetos Groovy, visando otimizar os algoritmos utilizados;
b) aplicar ferramentas de *profilers*, com o objetivo de encontrar os gargalos de memória;
c) comparar a linguagem Groovy com demais linguagens;
d) integrar o estudo de caso desenvolvido com a ferramenta PRONTO! (Gomes, 2009).

REFERÊNCIAS BIBLIOGRÁFICAS

ASSOCIACAO BRASILEIRA DE NORMAS TECNICAS. **NBR-13596**: tecnologia de informação - avaliação de produto de software - características de qualidade e diretrizes para o seu uso. Rio de Janeiro: ABNT, 1996. 10 p.

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

BARBARÁN, Gabriela C.; FRANCISCHINI, Paulino G. **Indicadores de produtividade na indústria de software**. São Paulo: DEP-POLI-USP, 2008.

BARROSO, Isaías C. Groovy: uma breve introdução. **Mundo Java**, São Paulo, ano III, n. 15, p. 50-57, jan. 2006.

BEZERRA, Eduardo. **Princípios de análise e projeto de sistemas com UML**. Rio de Janeiro: Campus, 2002.

CASTELLANI, Marcelo. **Um pouco de Groovy, uma linguagem de scripts 100% compatível com Java**. [S.l.], 2009. Disponível em: <http://www.devmedia.com.br/articles/viewcomp_forprint.asp?comp=12542>. Acesso em: 15 abr. 2010.

CESTA, André A. **A linguagem de programação Java**. Campinas, 2006. Disponível em: <http://www.ic.unicamp.br/~cmrubira/aacesta/java/javatut.html>. Acesso em: 21 abr. 2010.

CELEPAR. **Plataforma de desenvolvimento Pinhão Paraná**. Curitiba, 2000. Disponível em: < http://www.frameworkpinhao.pr.gov.br/>. Acesso em: 29 abr. 2011.

CONTROLCHAOS. **What is Scrum**. [S.l.], 2005. Disponível em: <http://www.controlchaos.com>. Acesso em: 25 maio 2010.

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.

FLANAGAN, David. **Java**: o guia essencial. 5. ed. Porto Alegre: Bookman, 2006.

GOMES, André Faria. **PRONTO!** Software para gestão de projetos ágeis. 2009. 75 f. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Faculdade de Informática e Administração Paulista, São Paulo. Disponível em: <http://pronto.bluesoft.com.br/historia/Monografia_pronto.pdf >. Acesso em: 26 ago. 2010.

GOMES, Nelma da S. **Qualidade de software –** Uma necessidade. São Paulo, 2006. Disponível em: <http://www.fazenda.gov.br/ucp/pnafe/cst/arquivos/Qualidade_de_Soft.pdf>. Acesso em: 21 abr. 2011.

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.

KÖNIG, Dierk. **Groovy em ação**. Rio de Janeiro: Alta Books, 2007.

MULLER, Henrique M. **RunGroovy:** extensão do bluej para execução de linhas de código. 2007. 67 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://campeche.inf.furb.br/tccs/2007-II/TCC2007-2-18-VF-HenriqueMMuller.pdf>. Acesso em: 1 jun. 2010.

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.

PEREIRA, Jhony A. **Ambiente web para gerenciamento de processo de software baseado no Scrum**. 2005. 69 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://www.bc.furb.br/docs/MO/2005/306300_1_1.pdf>. Acesso em: 1 jun. 2010.

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

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

RUSSI, Eliomar. **Avaliação da qualidade de sites acadêmicos baseado na norma NBR 13596.** 2002. 67 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://www.bc.furb.br/docs/MO/2002/266317_1_1.pdf>. Acesso em: 11 set. 2010.

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

SCHWABER, Ken; BEEDLE, Mike. **Agile software development with Scrum**. New Jersey: Prentice Hall, 2002.

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

SMITH, Glen; LEDBROOK, Peter. **Grails in action**. Greenwich: Manning Publications, 2009.

STRACHAN, James. **Groovy:** the birth of a new dynamic language for the Java platform. [S.l.], 2007. Disponível em: <http://radio.weblogs.com/0112098/2003/08/29.html>. Acesso em: 30 mar. 2010.

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

URUBATAN, Rodrigo. **Empresas que trabalham com java não gostam de produtividade?** [S.l.], 2010. Disponível em: <http://www.urubatan.com.br/empresas-que-trabalham-com-java-no-gostam-de-produtividade>. Acesso em: 16 nov. 2010.

VAZQUEZ, Carlos E.; SIMÕES, Guilherme S.; ALBERT, Renato M.. **Análise de pontos de função**: medição, estimativas e gerenciamento de projetos de software. São Paulo: Érica,2003.

VIEIRA, Rejane E. **A tecnologia moderna como herança da revolução industrial do século XVIII**. [S.l.], 2007. Disponível em: <http://www.artigosbrasil.net/art/varios/2144/tecnologia-moderna.html%22>. Acesso em: 20 set. 2010.

APÊNDICE A – QUESTIONÁRIO DE AVALIAÇÃO

O quadro 17 apresenta o questionário utilizado para avaliar as características "estáticas", das linguagens de programação.

1) A LP possui uma documentação completa e de fácil acesso? (Sintaxe)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

2) A LP possui uma padronização para definir atributos, métodos e classes? (Sintaxe)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

3) A LP sugere a utilização de padrões de projetos? (Abstração)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

4) A LP possui uma grande quantidade de recursos nativos? (Simplicidade global)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

5) É fácil dominar todos os recursos disponibilizados pela LP? (Simplicidade global)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

6) Existem atualizações da linguagem, adicionados novos recursos ou é uma LP consolidada e sem alteração? (Sintaxe)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

7) Há uma única forma para cada construção, ou cada construção pode ser representada por diversas formas? (Ortogonalidade)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

8) A codificação se aproxima da linguagem natural? (Legibilidade)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

9) Existe um padrão na formação de expressões, ou há exceções? (Ortogonalidade)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

10) Existe alguma representação para dados booleanos próximos da realidade? (Legibilidade)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

11) Há identificação na abertura e fechamento de blocos de comandos? (Legibilidade)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

12) A LP possui palavras reservadas? É possível utilizá-las como nome de variáveis? (Sintaxe)

( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

13) Existe limite de caracteres ao definir uma variável? (Tipo de dados e estrutura)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

14) As variáveis necessitam de um tipo? (Tipo de dados e estrutura)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

15) É possível utilizar o comando GOTO? Ou alguma derivação do comando? (Legibilidade)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

16) A LP possui facilidades para codificação que geram grande resultado computacional? (Expressividade)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

17) A LP permite a aplicação das técnicas de abstração como Herança e polimorfismo? (Abstração)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

18) A LP possui herança múltipla? (Abstração)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

19) A LP disponibilidade uma grande quantidade de bibliotecas prontas ao programador? (Expressividade)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

20) Existe o conceito de ponteiros na LP? (Confiabilidade)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

21) A LP o desenvolvedor criar seus próprios tipos de dados? (Tipos de dados e estrutura)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

22) Existe um gerenciamento de memória na LP? É necessário informar a quantidade de memória a ser alocada e desalocá-la manualmente? (Confiabilidade)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

23) A LP é totalmente orientada objetos? (Capacidade de escrita)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

24) Há a verificação de tipo dinamicamente? Antes de compilar ou ao compilar? (Verificação de tipos)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

25) A LP é fortemente tipada? (Verificação de tipos)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

26) Existe o tratamento de exceção na LP? (Tratamento de exceção)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

27) A LP permite tratar diferentes tipos de exceções de formas distintas? (Tratamento de exceção)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

28) Os programas gerados podem executar em qualquer plataforma? (Confiabilidade)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 - Sim

29) Há um conjunto de configurações iniciais pré-estabelecidas? (Capacidade de escrita)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 – Sim

30) A LP é facilmente adaptável para o desenvolvimento do que se propõem? (Capacidade de escrita)
( ) 0 – Não ( ) 5 – Parcial ( ) 10 – Sim

Quadro 17 – Questionário de avaliação das características “estáticas”

ANEXO A – USE CASE POINTS (UCP)

Método criado por Gustav Karner, em 1993, para medir projetos de software orientado a objetos (CELEPAR, 2000). Este método permite mensurar projetos baseado no modelo de caso de uso o que permite realizar a estimativa logo no início dos projetos, durante o levantamento de requisitos. Além da avaliação do modelo de casos de uso são considerados fatores de complexidade técnica (TCF), cuja finalidade é determinar o grau de complexidade do projeto a ser construído, e fatores ambientais (EF), cuja finalidade é determinar a eficiência do projeto e o nível de experiência dos profissionais relacionados.

Os passos para realizar esta medição são descritos no quadro 22.

a) Calcular o UUCP (*Unadjusted Use Case Point*):

Relacionar e classificar os atores envolvidos, de acordo com o nível de complexidade representado no quadro 18.

Complexidade	Definição	Peso
Simples	Quando o ator representa um sistema externo que é acessado através de API(Application Programming Interface)	1
Médio	Quando o ator representa um sistema externo, acessado através de um protocolo de comunicação(por exemplo: TCP/IP)	2
Complexo	Quando o ator interage com o sistema através de uma interface gráfica (GUI)	3

Fonte: CELEPAR (2000).

Quadro 18 – Complexidade dos atores

Relacionar e classificar os casos de uso, de acordo com o nível de complexidade representado no quadro 19.

Complexidade	Definição	Peso
Simples	Quando o caso de uso possui 3 ou menos transações, incluindo cenários alternativos, e sua realização deve acontecer menos de 5 objetos(classes de análise)	5
Médio	Quando o caso de uso possui de 4 a 7 transações, incluindo cenários alternativos, e sua realização deve acontecer com 5 a 10 objetos(classes de análise)	10

Complexo	Quando o caso de uso possui mais de 7 transações, incluindo cenários alternativos, e sua realização deve acontecer com mais de 10 objetos(classes de análise)	15

Fonte: CELEPAR (2000).

Quadro 19 – Complexidade dos casos de uso

Calcular efetivamente o UUCP, como segue:

UUCP = Total de pesos dos atores relacionados + Total de pesos dos casos de uso relacionados.

b) Calcular o TCF (*Technical Complexity Factor*):

Atribuir um valor a cada fator em uma escala de 0 a 5, onde 0 significa que o fator é irrelevante e 5 significa que é essencial. Se o fator não é importante e nem irrelevante deve-se atribuir o valor 3.

Código Fator	**Descrição do fator**	**Peso**	**Valor (Fator)**	**Resultado**
F1	Sistema distribuído	2		0
F2	Performance	1		0
F3	Eficiência para o usuário final (*online*)	1		0
F4	Processamento interno complexo	1		0
F5	Código deve ser reusável	1		0
F6	Fácil para instalar	0,5		0
F7	Fácil para usar	0,5		0
F8	Portável	2		0
F9	Fácil para mudar	1		0
F10	Concorrente	1		0
F11	Seguro	1		0
F12	Fornece acesso direto para *third parties* (sistemas/componentes externos)	1		0
F13	É requerido treinamento do usuário para usar o *software*	1		0

Fonte: CELEPAR (2000).

Quadro 20 – Calculo do fator de complexidade técnica

- Multiplicar o valor (fator) atribuído pelo respectivo peso (coluna resultado).
- Totalizar o resultado da multiplicação (TFator).
- Calcular o fator de complexidade de acordo com a seguinte formula:

$$TCF = 0.6 + (0.01 * fator)$$

Segue abaixo uma lista de características para alguns dos fatores de complexidade técnica, com o objetivo de auxiliar a definição de seu respectivo valor:

F1 – Sistema distribuído: dados ou processamento distribuído entre várias unidades de processamento – CPU.

0 ( ) A aplicação não auxilia na transferência de dados ou processamento entre as CPUs da instalação.

1 ( ) A aplicação prepara dados para o usuário final processar em outra CPU da instalação. Por exemplo, planilhas eletrônicas ou gerenciadores de banco de dados de PC.

2 ( ) Os dados são preparados para transferência, transferidos e processados em uma outra CPU da instalação (mas NÃO para processamento pelo usuário final como visto no item 1).

3 ( ) Processamento distribuído e transferência de dados on-line apenas em uma direção.

4 ( ) Processamento distribuído e transferência de dados on-line em ambas direções.

5 ( ) As funções de processamento são executadas dinamicamente na CPU mais apropriada.

F2 – Performance: Identifica os objetivos de performance da aplicação estabelecidos e aprovados pelo usuário.

0 ( ) Nenhuma exigência especial de performance foi fixada pelo usuário.

1 ( ) Requisitos de performance foram estabelecidos e revisados, mas nenhuma ação especial foi necessária.

2 ( ) O tempo de resposta é crítico durante as horas de pico. O intervalo de tempo limite (*deadline*) do processamento é sempre para o próximo dia útil. Nenhuma consideração especial para utilização de CPU foi requerida.

3 ( ) O tempo de resposta é crítico durante todo o horário de utilização. Os requisitos de prazo de processamento com outros sistemas são limitantes. Não foi necessário nenhum procedimento especial para utilização de CPU.

4 ( ) Os requisitos de performance estabelecidos pelo usuário são rigorosos o bastante para requerer tarefas de análise de performance na fase de análise e projeto da aplicação.

5 ( ) Além do descrito no item 4, ferramentas de análise de performance foram usadas nas fases de projeto, desenvolvimento e/ou implementação a fim de proporcionar a performance estabelecida pelo usuário.

F3 - Eficiência para o usuário final (*online*): as funções online fornecidas enfatizam um projeto da aplicação voltado para a eficiência do usuário final.

- Menus
- Documentação/Help online
- Movimento automático do cursor
- Movimento de Tela (*scrolling*) vertical e horizontal
- Impressão remota (via transações online)
- Teclas de Função pré-definidas
- Execução de *jobs batch* a partir de transações online
- Seleção de dados da tela via movimentação do cursor
- Uso intenso de vídeo reverso, brilho intensificado, sublinhado, cores e outros recursos de vídeo
- Documentação de transações online via hardcopy
- Interface para mouse
- *PopUp Windows*
- O mínimo possível de telas para executar as funções do negócio
- Fácil navegação entre telas (por exemplo, através de teclas de função)
- Suporte bilíngüe (suporta dois idiomas, contar como quatro itens)
- Suporte multilíngüe (suporta mais de dois idiomas, contar como seis itens)

0 ( ) A aplicação não apresenta nenhum dos itens acima.

1 ( ) Apresenta de 1 a 3 dos itens acima.

2 ( ) Apresenta de 4 a 5 dos itens acima.

3 ( ) Apresenta 6 ou mais dos itens acima, mas não há nenhum requisito do usuário relacionado à eficiência.

4 ( ) Apresenta 6 ou mais dos itens acima, e os requisitos estabelecidos para eficiência do usuário são rigorosos o suficiente para que a fase de projeto da aplicação inclua fatores para: minimizar a digitação, maximizar os *defaults*, utilizar *templates* etc.

5 ( ) Apresenta 6 ou mais dos itens acima, e os requisitos estabelecidos para eficiência do usuário são rigorosos o suficiente para que seja necessário o uso de ferramentas e processos especiais para demonstrar que os objetivos de eficiência foram alcançados.

F4 – Processamento interno complexo - a complexidade de processamento influência

no dimensionamento do sistema, e, portanto, deve ser quantificado o seu grau de influência com, base nas seguintes categorias:

- Processamento especial de auditoria e/ou processamento especial de segurança
- Processamento lógico extensivo.
- Processamento matemático extensivo.
- Grande quantidade de processamento de exceções, resultante de transações incompletas que necessitam de reprocessamento. Por exemplo: transações incompletas de ATMs causadas por interrupções de comunicação, valores de dados ausentes ou validações de erros.
- Processamento complexo para manipular múltiplas possibilidades de entrada/saída. Por exemplo: múltiplos meios e independência de equipamentos.

0 ( ) Não apresenta nenhum dos itens acima.
1 ( ) Apresenta um dos itens acima.
2 ( ) Apresenta dois dos itens acima.
3 ( ) Apresenta três dos itens acima.
4 ( ) Apresenta quatro dos itens acima.
5 ( ) Apresenta todos os itens acima.

F5 – Código deve ser reusável - a aplicação e o seu código foram especificamente projetados, desenvolvidos e suportados para serem reutilizados em outras aplicações.
0 ( ) Não apresenta código reutilizável.
1 ( ) O código reutilizável é usado somente dentro da aplicação.
2 ( ) Menos de 10% da aplicação foi feita, levando-se em conta a sua utilização por outras aplicações.
3 ( ) 10% ou mais da aplicação foi feita, levando-se em conta a sua utilização por outras aplicações.
4. ( ) A aplicação foi projetada e documentada para facilitar a reutilização de código e a aplicação é customizada pelo usuário a nível do código fonte.
5 ( ) A aplicação foi projetada e documentada para facilitar a reutilização de código

F6 – Fácil para instalar - indica o nível de preparação de procedimentos e ferramentas para instalação do sistema.

0 ( ) Nenhuma consideração especial foi feita pelo usuário e nenhum procedimento

especial foi requerido instalação.

1 ( ) Nenhuma consideração especial foi feita pelo usuário, mas um procedimento especial foi requerido para instalação.

2 ( ) Requisitos de instalação foram fixados pelo usuário.

3 ( ) Requisitos de instalação foram fixados pelo usuário e roteiros de instalação foram preparados e testados.

4 ( ) Além do descrito no item 2, ferramentas automatizadas de instalação foram preparadas e testadas.

5 ( ) Além do descrito no item 3, ferramentas automatizadas de instalação foram preparadas e testadas.

F7 – Fácil para usar (facilidade operacional) - Procedimentos efetivos de inicialização, *backup* e recuperação foram desenvolvidos e testados. A aplicação minimiza a necessidade de atividades manuais, tais como montagem de fitas magnéticas, manuseio de formulários e intervenção manual do operador.

0 ( ) Nenhuma consideração especial sobre facilidade operacional, além dos procedimentos normais de *backup*, foi feita pelo usuário.

1 ( ) Procedimentos eficientes de inicialização, *backup* e recuperação foram preparados, mas a intervenção do operador é necessária.

2 ( ) Procedimentos eficientes de inicialização, *backup* e recuperação foram preparados, mas nenhuma intervenção do operador é necessária (contar como dois itens).

3 ( ) A aplicação minimiza a operação de montagem de fitas magnéticas.

4 ( ) A aplicação minimiza a necessidade de manuseio de formulários.

5 ( ) A aplicação foi projetada para não precisar de intervenção do operador no seu funcionamento normal. Apenas a inicialização e parada do sistema ficam a cargo do operador. A recuperação automática de erros é uma característica da aplicação.

F8 – Portável: A aplicação foi especificamente projetada, desenvolvida e suportada para ser instalada em múltiplas plataformas (Windows, Unix, Linux).

0 ( ) Nenhuma solicitação do usuário para considerar a necessidade de instalar a aplicação em mais de uma plataforma.

1 ( ) Necessidade de instalação em múltiplas plataformas foi levada em consideração

no projeto do sistema e a aplicação foi projetada para operar somente em ambientes idênticos de hardware e software.

2 ( ) Necessidade de instalação em múltiplas plataformas foi levada em consideração no projeto do sistema e a aplicação foi projetada para operar somente em ambientes similares de hardware e software.

3 ( ) Necessidade de instalação em múltiplas plataformas foi levada em consideração no projeto do sistema e a aplicação foi projetada para operar inclusive em plataformas diferentes.

4. ( ) Um plano de documentação e manutenção foi elaborado e testado para suportar a aplicação em múltiplas plataformas e a aplicação atende aos itens 1 e 2.

5 ( ) Um plano de documentação e manutenção foi elaborado e testado para suportar a aplicação em múltiplas plataformas e a aplicação atende ao item 3.

F9 - Fácil para mudar: a aplicação foi especificamente projetada, desenvolvida para suportar manutenção, visando à facilidade de mudanças. Por exemplo:

· capacidade de consultas/relatórios flexíveis está disponível

· dados de controle do negócio são agrupados em tabelas passíveis de manutenção pelo usuário.

0 ( ) Nenhum requisito especial do usuário para projetar a aplicação, visando minimizar ou facilitar mudanças.

1 ( ) É fornecido recurso da consulta/relatórios flexíveis capaz de manipular solicitações simples de consulta (*quer/requests*). Por exemplo: lógica de *and*/*or* aplicada a somente um Arquivo Lógico Interno (contar como um item).

2 ( ) É fornecido recurso de consulta/relatórios flexíveis capaz de manipular solicitações de consulta (*query\requests*) de média complexidade. Por exemplo: lógica de *and/or* aplicada a mais de um arquivo lógico interno (contar como dois itens).

3 ( ) É fornecido recurso de consulta/relatórios flexíveis capaz de manipular solicita*/or* aplicadas a um ou mais arquivos lógicos internos (contar como três itens).

4 ( ) Dados de controle são mantidos em tabelas que são atualizadas pelo usuário através de processos on-linee interativos, mas as alterações só são efetivadas no próximo dia útil.

5 ( ) Dados de controle são mantidos em tabelas que podem ser atualizadas pelo usuário através de processos on-line e interativos e as alterações são efetivadas

imediatamente (contar como dois itens).

F10 – Concorrente: indica o volume de acesso simultâneo a aplicação.

0 ( ) Não é esperado acesso simultâneo

1 ( ) São esperados acessos simultâneos esporadicamente

2 ( ) Acessos simultâneos são esperados.

3 ( ) Acessos simultâneos são esperados diariamente

4 ( ) Muitos acessos simultâneos foram fixados pelo usuário para a aplicação, o que força a execução de tarefas de análise de performance na fase de projeto da aplicação.

5 ( ) Requer o uso de ferramentas controle de acesso nas fases de projeto desenvolvimento e/ou implantação, além das considerações acima.

F11 – Seguro (características especiais de segurança): indica o nível de segurança exigido pela aplicação.

0 ( ) Nenhuma solicitação do usuário para considerar a necessidade de controle de segurança da aplicação.

1 ( ) Necessidade de controle de segurança foi levada em consideração no projeto do sistema.

2 ( ) Necessidade de controle de segurança foi levada em consideração no projeto do sistema e a aplicação foi projetada para ser acessada somente por usuários autorizados.

3 ( ) Necessidade de controle de segurança foi levada em consideração no projeto do sistema e a aplicação foi projetada para ser acessada somente por usuários autorizados. O acesso será controlado e auditado.

4. ( ) Um plano de segurança foi elaborado e testado para suportar o controle de acesso a aplicação.

5. ( ) Um plano de segurança foi elaborado e testado para suportar o controle de acesso a aplicação e a auditoria.

F12 - É requerido treinamento do usuário para usar o software: indica o nível de facilidade de treinamento de usuários.

0 ( ) Nenhuma solicitação do usuário para considerar a necessidade de treinamento especial.

1 ( ) Necessidade de treinamento especial foi levada em consideração no projeto do

sistema.

2 ( ) Necessidade de treinamento foi levada em consideração no projeto do sistema e a aplicação foi projetada para ser acessada com facilidade pelos usuários.

3 ( ) Necessidade de treinamento especial foi levado em consideração no projeto do sistema e a aplicação foi projetada para ser utilizada por usuários de diversos níveis.

4 - 5 ( ) Um plano de treinamento foi elaborado e testado para facilitar o uso da aplicação.

c) Calcular o EF (*Environmental Factor*):

Atribuir um valor a cada fator numa escala de 0 a 5, onde 0 significa que o fator é irrelevante e 5 significa que é essencial. Se o fator não é importante e não é irrelevante deve-se atribuir valor 3.

Código Fator	Descrição do fator	Peso	Valor (Fator)	Resultado
F1	Familiaridade com o processo de desenvolvimento orientado a objetos adotado.	1,5		0
F2	Colaboradores de meio período	-1		0
F3	Capacidade de análise	0,5		0
F4	Experiência em desenvolvimento de aplicações deste gênero	0,5		0
F5	Experiência em Orientação a Objetos	1		0
F6	Motivação	1		0
F7	Dificuldade na linguagem de programação	-1		0
F8	Requisitos estáveis	2		0

Fonte: CELEPAR (2000).

Quadro 21 – Calculo dos fatores de ambiente

- Multiplicar o valor (Fator) atribuído pelo respectivo peso (coluna resultado);
- Totalizar o resultado da multiplicação (EFator);
- Calcular o fator de complexidade de acordo com a seguinte formula:

$$EF = 1.4 + (-0.03 * EFator)$$

Segue abaixo uma lista de características para os fatores ambientais que visam auxiliar a definição de seu respectivo valor:

F1 - Familiaridade com o processo de desenvolvimento orientado a objetos adotado:

indica a experiência da equipe com o processo/método utilizado para desenvolvimento do projeto.

0 ( ) A equipe não é familiar com o processo de desenvolvimento de software.

1 ( ) A equipe tem conhecimento teórico do processo de desenvolvimento de software.

2 - 3 ( ) Um ou mais membros utilizou o processo uma ou poucas vezes.

3 - 4 ( ) Pelo menos a metade dos membros da equipe tem experiência no uso do processo em diferentes projetos.

5 ( ) Toda a equipe tem experiência no uso do processo em vários projetos diferentes.

F2 – Colaboradores de meio período: mede a estabilidade da equipe e a influencia do trabalho parcial na produtividade.

0 ( ) Não tem membro com dedicação parcial.

1 - 2 ( ) Poucos membros (20%) trabalham em período parcial.

3 - 4 ( ) A metade dos membros da equipe trabalham em período parcial.

5 ( ) Toda os membros da equipe trabalham em período parcial

F3 – Capacidade de análise: indica a experiência do líder com análise de requisitos e modelagem.

0 ( ) O líder do projeto é novato.

1 - 2 ( ) Possui experiência de poucos projetos.

3 - 4 ( ) Pelo menos 2 anos de experiência com vários projetos.

5 ( ) Pelo menos 3 anos de experiência com projetos variados.

F4 – Experiência em desenvolvimento de aplicações deste gênero: indica a experiência com diferentes tipos de aplicação ou com o tipo de aplicação que está sendo desenvolvida.

0 ( ) Todos os membros da equipe são novatos

1 - 2 ( ) Poucos membros da equipe possuem alguma experiência (de 1 a 1 ½ ano). Os outros são novatos.

3 ( ) Todos os membros tem mais de 1 ½ ano de experiência.

4 ( ) A maioria da equipe tem 2 anos de experiência.

5 ( ) Todos os membros são experientes.

F5 – Experiência em orientação a objeto: mede a experiência da equipe com análise e projeto OO, modelagem de casos de uso, classes e componentes.
0 ( ) A equipe não é familiar com análise e projeto OO.
1 ( ) Todos os membros tem menos de 1 ano de experiência.
2 – 3 ( ) Todos os membros tem de 1 a 1 ½ ano de experiência.
4 ( ) A maioria da equipe tem mais de 2 anos de experiência.
5 ( ) Todos os membros são experientes (mais de 2 anos).

F6 – Motivação: descreve a motivação total da equipe.
0 ( ) Não motivada
1 - 2 ( ) Pouca motivada
3 - 4 ( ) A equipe está motivada para fazer um bom trabalho
5 ( ) A equipe está muito motivada e inspirada

F7 – Dificuldade na linguagem de programação: indica a experiência com ferramentas primárias de desenvolvimento e com a linguagem de programação escolhida.
0 ( ) Todos os membros da equipe são programadores experientes
1 ( ) A maioria dos membros da equipe possuem mais de 2 anos de experiência
2 ( ) Todos os membros tem mais de 1 ½ ano de experiência
3 ( ) A maioria da equipe tem mais de 1 ano de experiência
4 ( ) poucos membros da equipe tem alguma experiência (1 ano). Os outros são novatos.
5 ( ) Todos os membros da equipe são novatos.

F8 – Requisitos estáveis: mede o grau de mudança de requisitos e inseguranças sobre o significado dos requerimentos.
0 ( ) Requisitos muito instáveis com mudanças freqüentes.
1 - 2 ( ) Requisitos instáveis. Clientes demandam algumas mudanças realizadas em diversos intervalos.
3 - 4 ( ) Estabilidade global. Pequenas mudanças são necessárias.
5 ( ) Requisitos estáveis ao longo do desenvolvimento.

d) Calcular o UCP (*Use Case Points*):

Por fim, calcular o UCP utilizando os cálculos obtidos anteriormente:

UCP = UUCP * TCF * EF

e) Estimar o projeto em horas:

Karner (1993 apud CELEPAR, 2000), sugere a aplicação de 20 horas/homem por ponto de UCP.

Estimativa (horas) = UCP * 20

Fonte: CELEPAR (2000).

Quadro 22 – Calculo dos *use case points*