Estudo e avaliação de ferramentas para desenvolvimento ágil

Pablo Ribeiro Suárez[1], Gernan Soares de Andrade[2], Allyson Jerônimo Dantas[3], Karlos Thadeu Matias Vital de Oliveira[4], Geam Carlos de Araújo Filgueira[5]

[1] pablo@ffm.com.br. [2] gernan@ffm.com.br. [3] allyson@ffm.com.br. [4] karlos.oliveira@ifrn.edu.br. [5] geam.filgueira@ifrn.edu.br. Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte – Câmpus Caicó - RN 288, s/n, Nova Caicó. Caicó – RN. CEP: 59300-000

RESUMO

O desenvolvimento de software é ainda considerado uma tarefa árdua. Equipes de desenvolvimento despendem muito tempo realizando tarefas triviais, desde a estruturação do projeto à criação dos componentes mais básicos e indispensáveis para sua implementação, deixando assim de se concentrar nas particularidades do projeto. Buscando maior eficiência, foram criados os *frameworks* de desenvolvimento ágil, ferramentas que permitem o desenvolvimento de software de maneira rápida e eficiente, utilizando-se de recursos como geração automática de código e convenções sobre configurações, entre outros. Este trabalho tem como objetivo fazer uma análise comparativa dessas ferramentas. Para alcançar esse objetivo, primeiramente foram realizadas pesquisas para definir quais ferramentas seriam utilizadas. Logo após, foram definidos os critérios de avaliação que serviriam como métricas para os testes. Para alcançar os resultados, um cenário foi simulado e a partir desse cenário um projeto idêntico foi implementado em cada uma das ferramentas. Durante a implementação desse projeto, foi feito o estudo das ferramentas, posteriormente explicado nos resultados.

Palavras-chave: Frameworks. Desenvolvimento ágil. Geração automática de código.

ABSTRACT

Software development is still considered a hard task. Development teams expend a lot of time doing trivial tasks, since structuring the project to creating the most basic and essential components for its implementation, then not focusing on the particulars of the project. Searching by greater efficiency, came into being some agile development frameworks, tools that allow software developing in a quick and efficient way, using features like automatic code generation and setting conventions, among others. This paper aims to make a comparative analysis of some of these tools. To achieve this object, initially a research was conducted to determine which tools would be used. After, the evaluation criteria were defined. To achieve the results, a scenario was simulated and from that scenario an identical project was implemented in each one of the tools. During the implementation of this project, a tool study was made and then further explained at the results.

Keywords: *Frameworks. Agile development. Automatic code generation.*

1 Introdução

O *software* está cada vez mais presente no cotidiano das empresas e instituições, com o objetivo de aumentar a capacidade de concorrência, automatizar as tarefas que antes mobilizavam toda uma equipe e também resolver problemas encontrados pela mesma equipe na sua área de atuação.

De acordo com Talib e Malkawi (2011), a demanda de *software* cresceu 108% entre 2000 e 2007.

Devido a toda essa busca por *software*, exige-se cada vez mais qualidade. Para que as empresas desenvolvedoras possam garantir sua sobrevivência no competitivo mercado de *software*, elas devem oferecer aos seus clientes o maior número de vantagens e a maior qualidade possível do produto, no menor custo e prazo.

Com o objetivo de cumprir esses requisitos do mercado, as empresas de desenvolvimento de *software* adotam boas práticas, técnicas e processos regidos pela Engenharia de Software. Segundo Sommerville (2003), a Engenharia de Software é uma disciplina que se envolve em todos os aspectos da produção, desde os estágios iniciais de especificação do sistema até a manutenção desse sistema, depois que ele entrou em operação.

Durante o desenvolvimento de um *software*, existem atividades rotineiras, ou seja, atividades que para todo novo projeto os programadores e/ou a equipe desenvolvedora têm que executar, como a criação das entidades principais, as classes de persistência dessas entidades (*DAOs – Data Access Objects* – Objetos de Acesso a Dados) contendo todo o CRUD (*Create, Read, Update, Delete* – Criar, Ler, Atualizar, Deletar) e as classes de interfaces gráficas e fachadas dessas entidades, além de todas as configurações (muitas vezes genéricas) do *framework* de persistência e conexões externas com o SGBD (Sistema Gerenciador de Banco de Dados). Essas tarefas consomem muito tempo, tempo esse que a equipe de desenvolvimento poderia estar usando com as atividades mais complexas e particulares do projeto em questão.

Para agilizar essas tarefas, foram criadas várias ferramentas e *frameworks* que geram código automaticamente – a esses mecanismos dá-se o nome de ferramentas de desenvolvimento ágil ou RAD (*Rapid Application Development* – Desenvolvimento Rápido de Aplicação). A proposta deste artigo é elencar as tecnologias mais utilizadas pela comunidade de desenvolvimento, para o propósito citado: fazer uma análise individual e um comparativo entre elas, levando em consideração critérios que serão definidos adiante.

2 Material e Métodos

Para a escolha das ferramentas, foi realizada uma pesquisa das principais linguagens de programação utilizadas no mercado. O *ranking* pode ser visto na Tabela 1.

Tabela 1 – *Ranking* das linguagens de programação mais utilizadas no mercado.

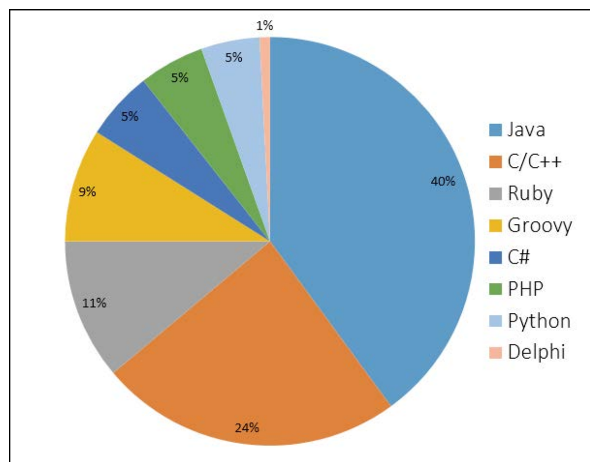
Position May 2013	*Position May 2012*	*Delta in Position*	*Programming Language*	*Ratings May 2013*	*Delta May 2012*	*Status*
1	1	=	C	18729%	+1.38%	A
2	2	=	Java	16914%	+0.31%	A
3	4	↑	Objective-C	10428%	+2.12%	A
4	3	↓	C++	9198%	-0.63%	A
...	...	...	...	...	...	...
10	11	↑	Ruby	1670%	+0.22%	A
...	...	...	...	...	...	...

Fonte: Tiobe, 2013.

Com o intuito de abordar os *frameworks* mais utilizados e/ou populares atualmente, também foi feita uma pesquisa em algumas comunidades de desenvolvimento, cujos resultados podem ser visualizados na Figura 1.

Baseando-se no *ranking* da Tabela 1 e nos resultados da pesquisa exibidos na Figura 1, as seguintes linguagens foram selecionadas: *Java*, *Ruby* e *Groovy*. As linguagens C/C++ não possuem nenhum *framework* de desenvolvimento ágil para web, o que as desclassifica da seleção.

Figura 1 – Resultado da pesquisa sobre linguagens de programação mais utilizadas.



A linguagem *Groovy* não aparece no *ranking*, porém optou-se por essa linguagem devido à influência que ela sofre do *Java* – seu popular *framework* de desenvolvimento ágil, que será apresentado mais adiante – e ao resultado da pesquisa feita nas comunidades de desenvolvedores. Cada uma das linguagens escolhidas para serem abordadas neste artigo possui um ou vários *frameworks* para elevar a produtividade. Dentre todas as ferramentas, algumas foram escolhidas porque demonstraram uma comunidade mais ativa e um maior número de material disponível. São elas: *Spring-Roo* (*Java*), *Grails* (*Groovy*) e *Ruby On Rails* (*Ruby*).

Todos esses *frameworks* fazem uso de CoC (*Convention-over-Configuration*) e obedecem ao princípio DRY (*Don't Repeat Yourself*), que serão explicados adiante, e logo após será feita uma breve introdução a cada uma das tecnologias.

2.1 *Don't Repeat Youself* (DRY)

Um dos grandes problemas enfrentados durante o desenvolvimento de uma aplicação web é a repetição de trabalho. O desenvolvedor precisa sempre redefinir várias vezes a mesma configuração. *Frameworks* ágeis, como o *Spring-Roo*, *Grails* e *Ruby On Rails*, eliminam esse problema através da geração automática de grande parte desses artefatos, e como resultado, a repetição dessas atividades é diminuída significativamente.

2.2 *Convention-over-Configuration* (CoC)

A convenção sobre configuração, também conhecida como programação por convenção, consiste em diminuir a complexidade e o número de decisões que o desenvolvedor tem de tomar, tornando o desenvolvimento mais fácil e produtivo e reforçando o princípio DRY, conforme Albuquerque (2011).

Isso quer dizer que o desenvolvedor irá se preocupar apenas com aspectos não convencionais da aplicação, pois, ao invés de vários arquivos de configuração XML (*eXtensible Markup Language* – Linguagem de Marcação Extensível), o desenvolvedor baseia-se em convenções para definir inúmeras propriedades.

Por exemplo, uma convenção de nomes pode ser adotada, na qual o nome da tabela seja sempre o plural do nome da classe persistente, sendo preciso mudar essa convenção apenas quando for realmente necessário desviar do modelo.

2.3 *Spring-Roo*

Spring-Roo é um *framework* para geração rápida de aplicações Java EE (*Enterprise Edition* – Edição Empresarial) que foca na produtividade, permitindo que, com apenas um comando, o desenvolvedor crie e gerencie aplicações *Spring*, podendo adicionar ou alterar componentes em qualquer uma das camadas da arquitetura, segundo Rimple e Penchikala (2012).

Vale a pena citar que, como o *Spring-Roo* é 100% *Java*, isso implica dizer que todos os recursos da linguagem podem ser aproveitados.

2.4 *Grails*

Grails, assim como o *Spring-Roo*, é um *framework* para o desenvolvimento ágil, mas não na linguagem *Java*, e sim em *Groovy*, que é uma linguagem dinâmica para a plataforma *Java*.

Primeiramente chamada de *Groovy On Rails*, depois renomeada para *Grails*, possui código aberto e tem como principal objetivo aumentar a produtividade e permitir o desenvolvimento rápido de aplicações utilizando as tecnologias recentes do mundo *Java*.

Como o *Groovy* é uma linguagem dinâmica para a plataforma *Java*, também é possível utilizar os recursos do *Java* de forma transparente.

2.5 *Ruby On Rails*

O RoR (*Ruby On Rails*), assim como o *Grails*, é um *framework* de código aberto que tem o objetivo de aumentar a produtividade, faz uso da linguagem *Ruby* e foi o pioneiro nesse tipo de ferramenta, conforme Bigg e Katz (2012).

O *Ruby* não consegue fazer uso das bibliotecas *Java*; contudo, possui seu próprio tipo de pacote para instalação e divulgação de bibliotecas, chamadas de *Gem* (Gema) – as gemas são gerenciadas pelo *RubyGems*, que é um gerenciador de pacotes que cuida não só da gema como também de suas dependências.

2.6 Levantamento dos Critérios de Avaliação

A avaliação dos *frameworks* utilizou como base as vantagens e desvantagens elencadas a partir do levantamento dos critérios. A seguir, são listados os critérios com uma breve explicação sobre como eles serão avaliados nas ferramentas:

Curva de aprendizado: a ideia de curva de aprendizado é que, quanto mais uma pessoa repete uma atividade, menos tempo leva para executar essa atividade. Esse critério, de acordo com Balbinotto (2008), diz respeito a aprender fazendo (*learning-by-doing*). No caso deste trabalho, será avaliada a dificuldade de aprender a utilizar cada uma das ferramentas.

Mapeamento objeto-relacional: dá-se o nome de impedância à incompatibilidade entre o paradigma orientado a objetos e o paradigma relacional, afirma Cordeiro (2012). Esse critério diz respeito a como os *frameworks* fazem a tradução de um paradigma para o outro e vice-versa.

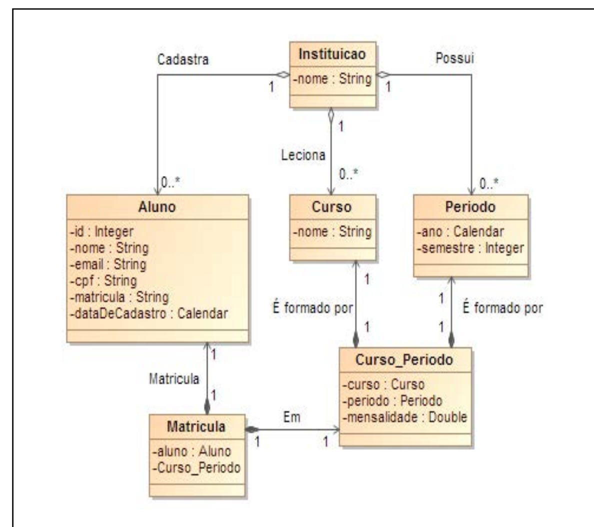
Levando em consideração que o código gerado por essas ferramentas não irá sempre atender totalmente a necessidade dos desenvolvedores e de seus clientes, é necessário fazer certas modificações manualmente. Isso implica em dois critérios: legibilidade – que diz respeito a quão legível e de fácil entendimento é o código – e escritabilidade – que diz respeito a quão modificável é o código que foi gerado.

2.7 Cenário de Avaliação

Para realizar a avaliação das ferramentas, foi escolhido um cenário baseado em situações que são encontradas no dia a dia do desenvolvedor. Os mais típicos dos cenários consistem em CRUDs de entidades. De acordo com Cockburn (2001), um CRUD representa as operações básicas de uma entidade no banco de dados – são elas: criar, editar, remover e consultar. Sendo assim, o cenário escolhido consiste em um conjunto de CRUDs, simulando um sistema de gerenciamento de cadastros de alunos, cursos e períodos em uma instituição, além da criação de semestres para cada curso e da matrícula dos alunos nesses semestres.

Esse sistema será de agora em diante chamado de SistemaArtigo. Na Figura 2, pode ser visto um diagrama de classes que descreve as entidades do sistema e seus atributos.

Figura 2 – Diagrama de classes do SistemaArtigo.



O SistemaArtigo será implementado de forma exatamente igual para os 3 *frameworks*; a partir dessas implementações serão feitas as avaliações.

3 Resultados e Discussão

3.1 Implementação no *Spring-Roo*

"Quanto à curva de aprendizado, esse *framework* se utiliza do *shell* para fazer/gerar quase tudo: adicionar *frameworks*, criar entidades, definir atributos e configurações (se é único, tamanho, etc.) desses atributos – além de relações com outros objetos – gerar classes controladores e *views*, configurar *plug-ins*, entre outros.

O *shell* conta com um incrível e funcional recurso de autocompletar, que se mostra bem intuitivo e faz boa parte do trabalho, inclusive com um único comando (por exemplo, gerar todos os controladores),

e todo o código gerado é 100% *Java*, restando ao desenvolvedor implementar as particularidades.

O mapeamento objeto-relacional é feito de maneira quase transparente. Quando se referencia um objeto, dependendo do tipo do objeto (por exemplo, o próprio objeto referenciado ou uma lista do objeto referenciado), a ferramenta já cuida de todo o resto, restando apenas dizer quem é o dono do relacionamento nos casos mais específicos. As tabelas são geradas no banco de dados automaticamente.

Quanto à legibilidade, as entidades e as classes controladoras dessas entidades são muito fáceis de serem compreendidas e até editadas diretamente(fora do *shell*). Já o código das *views* se mostrou mais complexo, e foi necessário um estudo mais profundo para entender esse código e conseguir aplicar modificações, mesmo que pequenas.

Na questão de escritabilidade, por mais que quase tudo seja feito ou gerado a partir de um *shell*, todo o código gerado está amplamente aberto a modificações; devido à facilidade de leitura do código ser alta, torna-se muito fácil fazer essas alterações sempre que necessário.

3.2 Implementação no *Grails*

Quanto à curva de aprendizado, há certa dificuldade inicial para entender como tudo funciona no *Grails*. A definição de classes e atributos não é tão simples quanto no *Spring-Roo*, porém isso não o torna difícil de ser utilizado. O *Grails* é bastante similar ao esquema de programação POO (Programação Orientada a Objetos) tradicional em *Java*, no qual primeiramente são definidos os atributos das classes, depois os valores iniciais de cada atributo. No entanto, o *Grails* também fornece *constraints*, que são configurações para cada atributo, nada disso feito em *shell*. Os controladores têm que ser gerados manualmente para cada entidade, e precisa ser definida uma configuração (*scaffold*) para que o *framework* gere o CRUD da entidade em questão. As *views* são geradas automaticamente e existem arquivos para a configuração de erros, mensagens e internacionalização, o que torna bem fácil o gerenciamento do *framework*.

O mapeamento objeto-relacional é intuitivo: existem palavras reservadas só para definir o tipo de associação (*hasOne* de 1 para 1, *hasMany* de 1 para muitos), e depois de definido qual o tipo de associação, basta informar qual a entidade alvo e quem é o dono da relação, nos casos não bidirecionais, através de *belongsTo*; todo o resto é gerenciado pelo *framework*. As tabelas são geradas automaticamente; contudo, por padrão, ele não usa nenhum banco de dados para armazenar os dados, então esse recurso deve ser configurado posteriormente.

O código gerado é fácil de ser entendido, principalmente nos controladores e nas classes tanto dos controladores quanto das *views*, apesar de que, em alguns casos, alguns códigos "estranhos" são gerados nos arquivos da visão e geram certa complexidade.

Como depende de uma linguagem dinâmica, no *Grails* é um pouco mais confuso de se aprender o que é opcional e o que não é. Depois de dominar essa parte da sintaxe, vê-se que a linguagem trabalha da maneira mais prática possível; todo o código gerado é altamente modificável.

3.3 Implementação no *RubyOnRails*

Quanto à curva de aprendizado, o *RubyOnRails* é o único entre as ferramentas que estão sendo avaliadas que não se parece com o *Java*, e sim com C/C++. Assim como no *Spring-Roo*, quase tudo pode ser feito via *shell*. Não há recursos de autocompletar, contudo, os comandos são intuitivos e ficam na memória depois do primeiro uso.

Para definir as propriedades dos atributos dos objetos, usam-se palavras reservadas da linguagem (por exemplo, *validates* :*presence*, :*length* – lembram as *constraints* do *Grails*), e uma grande vantagem é que erros e mensagens podem ser configurados diretamente em um arquivo de propriedades, simplesmente apontando para o campo a ser validado, informando qual o tipo de validação e a mensagem de erro.

Em relação ao mapeamento objeto-relacional, o *RubyOnRails* trabalha de maneira parecida com o *Grails*, possui as palavras reservadas *has_one* e *has_many* para definir os tipos de associações e *belongs_to* para definir o dono da relação, sendo o resto gerenciado pelo *framework*, de forma simples e eficiente.

Entretanto, diferentemente do *Spring-Roo* e do *Grails*, tabelas não são geradas automaticamente na execução do sistema: usa-se o comando *rakedb:create* para criar as tabelas e o comando *rakedb:migration*, que é um grande diferencial. Uma migração é basicamente qualquer modificação que for feita no banco de dados. O *rails* guarda todas as *migrations* e permite que o desenvolvedor retorne às

versões mais antigas do banco a qualquer momento que desejar.

A maior dificuldade é entender a sintaxe do *Ruby*, que é um tanto quanto complexa; após isso, percebe-se que o código gerado é limpo e direto. Uma das dificuldades foi encontrar onde estavam alguns dos códigos gerados.

3.4 Comparação dos *frameworks*

Com o SistemaArtigo implementado em todas as ferramentas elencadas, foi feita uma comparação levando em consideração a experiência do autor em cada uma das implementações.

A seguir, será explicada a qualificação de cada uma das ferramentas à luz dos critérios, utilizando uma das qualificações a seguir:

Excelente: Caso o *framework* ofereça suporte a todo o requisito de maneira simples e transparente ou não demonstre complexidade.

Muito bom: Caso o *framework* atenda ao requisito de maneira simples, porém não cumpra toda a tarefa ou demonstre alguma complexidade.

Satisfatório: Caso o *framework* não atenda completamente ao requisito ou demonstre um nível médio de complexidade.

Ruim: Caso o *framework* não ofereça suporte a alguma função ou demonstre alto nível de complexidade. O resultado da comparação pode ser visto adiante, na Tabela 2.

Tabela 2 – Qualificação dos *Frameworks*

Ferramentas	Critérios			
	Curva de aprendizado	**Mapeamento objeto-relacional**	**Legibilidade**	**Escritabilidade**
Spring-Roo	Excelente	Excelente	Muito bom	Excelente
Grails	Muito bom	Muito bom	Muito bom	Excelente
RubyOnRails	Satisfatório	Excelente	Satisfatório	Excelente

No quesito Curva de aprendizado, o *Spring-Roo* ficou classificado como excelente, pois não houve nenhum tipo de dificuldade no seu aprendizado, a comunidade ajuda bastante, tirando dúvidas, e existe muito material na internet, desde o básico ao avançado. O *Grails* ficou qualificado como muito bom, pois também é muito fácil de se aprender, e apesar de não possuir tanto material disponível como o *Spring-Roo*, também possui material mais do que suficiente e comunidade ativa. O *RubyOnRails* foi satisfatório nesse quesito: foi o mais difícil de aprender, o que não o qualifica como ruim, apenas requer um estudo mais profundo. No entanto, os materiais encontrados são um tanto quanto básicos demais. Outro ponto negativo que vale ser levado em consideração é que a IDE (*Integrated Development Environment*) *Netbeans* deixou de dar suporte a essa ferramenta, o que talvez demonstre desinteresse por parte dos programadores.

Em relação ao Mapeamento objeto-relacional, o *Spring-Roo* fica classificado como excelente, pois é muito simples criar seu banco de dados; depois de este criado, é tarefa do *framework* criar as tabelas, e tudo é gerenciado automaticamente. O *Grails*, assim como o *RubyOnRails*, também desempenha muito bem esse papel; entretanto, o *Grails* fica classificado como muito bom por exigir algumas configurações a mais, enquanto o *RubyOnRails* recebe a qualificação "excelente", por ter a mesma complexidade do *Grails*, porém adicionar recursos muito interessantes como o *db:migration*.

No que diz respeito à Legibilidade, o *Spring-Roo* recebe a classificação "muito bom", pois o código gerado é bem intuitivo, as anotações fazem boa parte do trabalho, sendo ele um pouco confuso apenas na camada de visão. O *Grails* também recebe a mesma classificação, pois, assim como o *Spring-Roo*, gera códigos bem intuitivos, e faz uso do que chama de *constraints* para realizar trabalhos que no *Spring-Roo* são feitos por anotações, porém com a mesma competência. O *Grails* também é um pouco confuso na camada da visão. Compreender o *RubyOnRails* demonstrou ser um pouco complicado: não foi tão simples encontrar o que se procurava quanto foi no *Spring-Roo* e *Grails*; todavia, com um pouco mais de

estudo, tudo foi resolvido, por isso a sua qualificação fica como satisfatório.

Todo o código gerado por todas as ferramentas está amplamente passível de edição em qualquer camada da aplicação. Como todas as ferramentas fazem uso de arquitetura MVC (*Model-View-Controller* – Modelo-Visão-Controlador), já se deduz onde ou em que camada vai ocorrer a alteração. Em momento algum qualquer tipo de código gerado por parte dos *frameworks* estava bloqueado a alterações manuais (como é o caso em alguns geradores automáticos de código), por isso todas as ferramentas recebem a qualificação "excelente". Nenhuma das ferramentas foi classificada como ruim em nenhum dos critérios, pois todas conseguiram atingir uma qualificação no mínimo satisfatória, provando assim que são de qualidade e podem ser opções extremamente viáveis.

4 Conclusões

Este artigo teve como principal foco a programação ágil para web, que busca permitir aos desenvolvedores a criação de aplicativos de forma rápida, na qual módulos funcionais são entregues rapidamente ao cliente e colocados em produção.

Utilizando-se de princípios como o DRY e o CoC, já existem vários *frameworks* para as mais diversas linguagens que se encontram disponíveis e amadurecidos. Tendo em vista as várias ferramentas existentes, foram realizadas pesquisas em comunidades de desenvolvedores para elencar as ferramentas que seriam testadas durante o desenvolvimento deste trabalho; tais pesquisas resultaram no *Spring-Roo*, *Grails* e *RubyOnRails*.

Com as ferramentas elencadas, foram levantados critérios de avaliação que são considerados de extrema importância quando se trata da adaptação a uma nova tecnologia. A partir desses critérios, as ferramentas foram analisadas e testadas em um cenário escolhido.

Através dos testes foi possível elencar pontos fortes e pontos fracos das ferramentas, que se mostraram muito eficientes em um cenário no qual o desenvolvimento ainda é considerado lento, e erros e falhas ainda são difíceis de encontrar, conforme afirmam Stepp, Miller e Kirst (2009).

Para a implementação do cenário proposto, as ferramentas disponibilizaram uma gama de recursos que permitiram gerar desde toda a estrutura do projeto até interfaces gráficas e persistência de entidades do sistema. Vale informar que esses *frameworks* são projetados e implementados utilizando normas e padrões de projeto, acrescentando assim ainda mais qualidade e segurança ao projeto em si e aos seus utilizadores.

Foi possível notar também, de fato, a enorme facilidade provida pela CoC, que permitiu abstrair muitas configurações que eram anteriormente realizadas uma por uma e se concentrar nas particularidades do projeto, resultando em muito mais produtividade.

REFERÊNCIAS

ALBUQUERQUE, RENAN R. B. **Desenvolvendo Aplicações Ágeis e Dinâmicas com Groovy e Grails.** 2011. 129 p. Trabalho de Conclusão de Curso (Bacharelado em Sistemas de Informação) – Centro de Ensino Superior e Desenvolvimento, Campina Grande, 2011.

BALBINOTTO, Giácomo. **Curvas de Aprendizagem – Análise Microecconômica.** 2008. Disponível em: <http://www.ppge.ufrgs.br/giacomo/arquivos/ats/learning-curve-laparoscopia.pdf>. Acesso em: 30 maio 2013.

BIGG, Ryan; KATZ, Yehuda. **Rails 3 in Action**. 1. ed. San Francisco: Manning Publications, 2012.

COCKBURN, Alistair. **Escrevendo Casos de Uso Eficazes**: um guia prático para desenvolvedores de software. 1. ed. São Paulo: Artmed, 2001.

CORDEIRO, Gilliard. **Aplicações Java para a Web com JSF e JPA**. 1. ed. São Paulo: Casa do Código, 2012.

RIMPLE, Ken; PENCHIKALA, Srini. **Spring Roo in Action**. 1. ed. San Francisco: Manning Publications, 2012.

STEPP, Marty; MILLER, Jessica; KIRST, Victoria. **A "CS 1.5" introduction to web programming.** Seattle: University of Washington, 2009.

SOMMERVILLE, Ian. **Engenharia de Software**. 6. ed. São Paulo: Addison-Wesley, 2003.

TALIB, Ayman; MALKAWI, Mohammed. **Inward Strategy**: an optimal solution to build a software industry in Saudi Arabia. 2011. Disponível em: <http://www.ibimapublishing.com/journals/IBIMABR/2011/126226/126226.html>. Acesso em: 04 jun. 2012.

TIOBE SOFTWARE. **TIOBE Programming Language Community Index**. 2013. Disponível em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Acesso em: 22 mai. 2013.