DISSERTAÇÃO DE MESTRADO

UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

"MDWA: Uma abordagem guiada por modelos para desenvolvimento de software Web"

ALUNO: Marcelo Brandão Theodoro Júnior
ORIENTADOR: Profª Dra. Rosângela A. D. Penteado

São Carlos
Novembro/2012

CAIXA POSTAL 676
FONE/FAX: (16) 3351-8233
13565-905 - SÃO CARLOS – SP
BRASIL

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia Programa de Pós-Graduação em Ciência da Computação

MDWA: Uma abordagem guiada por modelos para desenvolvimento de software Web

MARCELO BRANDÃO THEODORO JÚNIOR

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software.

Orientadora: Profª. Dra. Rosângela A. D. Penteado

UFSCar – Universidade Federal de São Carlos

Novembro de 2012

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

T388ma

Theodoro Júnior, Marcelo Brandão.
MDWA : Uma abordagem guiada por modelos para desenvolvimento de software Web / Marcelo Brandão Theodoro Júnior. -- São Carlos : UFSCar, 2013.
125 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2012.

1. Engenharia de software. 2. Desenvolvimento orientado por modelos. 3. Desenvolvimento de software Web. 4. Framework Ruby on Rails (Programa de computador). 5. Linguagens específicas de domínio. I. Título.

CDD: 005.1 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"MDWA: Uma abordagem guiada por modelos para desenvolvimento de software Web"

Marcelo Brandão Theodoro Júnior

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Profa. Dra. Rosângela Ap. Dellosso Penteado
(Orientadora - DC/UFSCar)

Prof. Dr. Daniel Lucrédio
(DC/UFSCar)

Profa. Dra. Maria Istela Cagnin Machado
(UFMS)

São Carlos
Novembro/2012

AGRADECIMENTOS

Agradeço à professora Drª. Rosângela, minha orientadora, que acreditou em mim durante todos esses anos de convívio. Sua participação, orientação e companheirismo foram fundamentais para este trabalho.

Aos meus pais, Marcelo e Elaine, que me deram condições de realizar um curso de graduação e, posteriormente, este trabalho de mestrado. Ao meu irmão Matheus, agradeço pela amizade.

Aos grandes amigos, Oscar e Deivid, muito obrigado pelo apoio e pela cobertura. Neste trabalho tem alguns dos cabelos brancos de vocês. Espero poder ajudá-los da mesma forma quando precisarem.

À minha amada esposa Beth e meu filho Eduardo. Vocês são a inspiração para tudo que eu faço. O apoio de vocês foi importantíssimo durante este trabalho e será nos próximos.

Aos companheiros de laboratório, Matheus, André, Paulo e Rafael, obrigado pelos conselhos pontuais que foram de grande valia para este trabalho.

A todos que contribuíram direta ou indiretamente para a realização deste trabalho.

Agradeço ao apoio das instituições que contribuíram com este trabalho, à UFSCar pela estrutura e ensino e ao CNPQ pelo apoio financeiro.

“Everything should be made as simple as possible, but not simpler.”

Albert Einstein

RESUMO

As técnicas de desenvolvimento de software evoluem continuamente com a finalidade de melhorar processos de construção e manutenção de software, além de obter ganhos em tempo, custo e qualidade. O objetivo do MDD é reduzir a distância semântica entre um problema e a especificação de sua solução. Para isso, MDD tem enfoque na modelagem de alto nível de abstração e em sucessivos refinamentos dos modelos construídos em artefatos mais detalhados, até enfim, gerar código. Há afirmações de que o desenvolvimento orientado a modelos pode ser significativamente mais eficiente que o desenvolvimento tradicional guiado por código fonte, além de reduzir a possibilidade de ocorrência de uma série de problemas durante o ciclo de vida do software. Da mesma forma, a engenharia de aplicações Web também pode ser beneficiada pela adoção de MDD, em especial com o apoio de abordagens que facilitem sua utilização. Como o desenvolvimento de aplicações Web comumente é ágil e com publicações freqüentes, essas abordagens devem ser flexíveis para que se adaptem a esse contexto. Entretanto, em geral, as abordagens propostas pela comunidade acadêmica apresentam processos complexos que envolvem diversos modelos, linguagens de programação, *plug-ins* e ambientes de programação. Essas características contrariam as práticas aprovadas pelos desenvolvedores Web. Esta dissertação apresenta a abordagem MDWA (*Model-Driven Web Applications*) que fornece um processo simples para desenvolvimento de software Web com apoio de MDD. A abordagem não depende de ferramentas, tecnologias ou *plug-ins* e estimula a combinação com outras formas de reuso e processos de desenvolvimento. Além disso, foi construída uma ferramenta, denominada Ruby-MDWA, baseada na linguagem Ruby e no framework Ruby on Rails destinada à criação de aplicações Web com auxílio da abordagem MDWA. Essa ferramenta fornece um conjunto de quatro modelos textuais e define transformadores M2M e M2C, que mantém a rastreabilidade de um requisito desde sua especificação até sua construção e posterior manutenção. Para mostrar o uso da abordagem e da ferramenta, foi realizado um estudo de caso real em conjunto com uma empresa de software de São Carlos – SP, onde um sistema de gerenciamento de projetos foi desenvolvido. De forma paralela, foram conduzidos dois experimentos com alunos de graduação em Bacharelado em Ciência da Computação e Engenharia de Computação e mestrado em computação da UFSCar, visando avaliar os ganhos e as limitações da ferramenta Ruby-MDWA.

Palavras-chave: desenvolvimento orientado a modelos, desenvolvimento de software web, linguagens específicas de domínio, geração de código, frameworks, linguagens de padrões, ruby on rails, abordagens MDD.

ABSTRACT

Software development techniques continually evolve in order to improve development and maintenance processes in addition to lower costs and higher quality. The goal of MDD is to reduce the semantic distance between a problem and its solution specification. Therefore MDD focuses on high-level abstraction modeling and successive model transformations, until finally, generate code. Studies assert that model-driven development can be significantly more efficient than traditional source code-driven software development and still reduce the possibility of occurrence of several problems during the software life-cycle. Likewise, Web engineering can also be benefited by MDD adoption, especially when supported by approaches that facilitate MDD use. Web development is usually agile with frequent releases, these approaches must be flexible to adapt to this context. However, generally, the approaches proposed by the academic community have complex processes which involve many different model definitions, programming languages, plug-ins and IDEs. These features contradict the practices adopted by Web developers. This paper presents the MWDA (Model-Driven Web Applications) approach that provides a simple process to support model-driven web development. This approach does not depend on tools, technologies or plug-ins and encourage combination with other forms of reuse and development processes. Furthermore, the Ruby-MDWA was developed with Ruby language and Ruby on Rails framework support, in order to create Web applications with MDWA assistance. This tool provides a set of textual models and defines M2M and M2C transformation tools, maintaining the requirements traceability since its specification to its construction and further maintenance. In order to show the use of the approach and tool, it was performed a real study case with a software company, from São Carlos – SP, where a project management system was developed. In parallel, two experiments were conducted with undergraduate students in Computer Science and Computer Engineering and a Masters in Computer Science, to evaluate the gains and limitations of the Ruby-MDWA tool.

Keywords: model-driven development, web software development, domain-specific languages, code generation tools, frameworks, pattern languages, ruby on rails, MDD approaches.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 Introdução

1.1 Contextualização

A popularização de serviços Web essenciais, como aplicativos de e-mail, compras via *e-commerce* e *internet banking*, aliados aos conceitos de *cloud computing* (Armbrust *et al*, 2010) e a diminuição dos custos de hospedagem de serviços *on-line*, motivou a evolução do desenvolvimento de software Web e a criação de novas tecnologias para suprir a demanda por softwares Web mais complexos.

O aumento da complexidade de um software normalmente afeta a sua evolução e a sua manutenibilidade. Esse problema é recorrente por que o desenvolvimento de software tradicional é guiado por um projeto de baixo nível de abstração e de código-fonte (Kleppe *et al*, 2003). Normalmente, isso faz com que a documentação de software fique rapidamente desatualizada, uma vez que as constantes transformações do código fonte não são aplicadas à documentação (Fu *et al*, 2011). Dessa forma, o conhecimento sobre a aplicação diminui com o passar do tempo, uma vez que para compreender todas as funções do software é necessário ter acompanhado sua execução ou analisar seu código fonte. As manutenções se tornam complexas e demoradas, pois os mantenedores devem entender todo o código fonte antes de efetuar qualquer alteração no software.

Várias abordagens foram propostas com intuito de amenizar os problemas anteriormente citados. *Model-driven Development* (MDD), em inglês, ou Desenvolvimento Orientado a Modelos, em português (Stahl *et al*, 2006) foi proposta como uma alternativa viável para a melhoria dos processos de desenvolvimento e manutenção de software. MDD propõe a construção de software a partir de modelos que representam níveis mais altos de abstração, expressando o domínio da aplicação de forma mais efetiva (Schmidt, 2006). Assim, os modelos produzidos são considerados como os principais artefatos do processo de desenvolvimento de software em vez do código fonte como ocorria anteriormente (Mellor; Clark; Futagami, 2003).

Schmidt (2006) afirma que o desenvolvimento orientado a modelos é baseado em linguagens de modelagem, que podem ser de propósito gerais ou específicas de domínio (DSML, em inglês, *Domain-Specific Modeling Languages*), e mecanismos de transformação

entre modelos. As transformações refinam os modelos e proporcionam mais detalhamento da aplicação até a geração de código fonte.

MDD é uma das técnicas que promovem a reutilização de software, não só do código fonte, mas principalmente da modelagem realizada. A especificação do software é feita por meio de modelos abstratos, que podem ser reutilizados em outras aplicações do mesmo domínio. A combinação de MDD com outras técnicas de reuso, como por exemplo, linguagens de padrões, frameworks e linhas de produtos de software (LPS), é uma forma de maximizar a eficiência e qualidade do software construído.

Para reduzir o esforço de construção de um software, é possível combinar diferentes técnicas de reuso, como por exemplo, padrões de projeto, frameworks e linhas de produtos de software, em um ambiente de desenvolvimento orientado a modelos. Este trabalho se aplica neste contexto, com a criação de uma abordagem para aplicação de MDD em um processo de desenvolvimento de software e a apresentação de diretivas para que essas técnicas de reuso sejam integradas à abordagem.

1.2 Motivação

A adoção de MDD para a engenharia de aplicações Web pode ser benéfica, assim como ocorre no desenvolvimento de aplicativos móveis e sistemas embarcados. Por exemplo, a empresa Nokia tem produzido aparelhos celulares dez vezes mais rápido com o apoio de MDD do que usando seu processo tradicional. A empresa de consultoria em *outsourcing* Lucent reporta ganhos de produtividade de três a dez vezes em suas atividades (Tolvanen, 2004 *apud* Lucrédio, 2009), também com o uso de MDD. Dessa maneira, o uso de MDD pode possibilitar a minimização de esforços e custos para o desenvolvimento de aplicações Web.

Apesar desses benefícios, ainda existe resistência quanto à adoção de MDD. Os desenvolvedores são seduzidos a escrever código, por ser a forma mais rápida de se implementar uma funcionalidade (Selic, 2003), comprometendo sua compreensão futura, ainda que a especificação de modelos seja a melhor forma para representar uma aplicação ou um domínio (Mellor *et al*, 2003) (France; Rumpe, 2007).

Além do fardo que a modelagem representa (Lucrédio, 2009), a utilização de MDD acrescenta novas etapas e tecnologias no desenvolvimento (Hailpern; Tarr, 2006) (Stahl *et al*,

2006) (Heijstek; Chaudron, 2010), pois os engenheiros de software devem fazer uso intensivo de modelos e ferramentas de transformação que envolvem várias linguagens de modelagem e tecnologias (Beydeda, 2005), aumentando a curva de aprendizado.

MDD também acrescenta complexidade nas tarefas de implementação da lógica de negócio de uma aplicação e das particularidades da tecnologia utilizada (Staron, 2006). A lógica de negócio, antes descrita somente diretamente no código fonte, passa a ser especificada em: i) ferramentas de transformação entre modelos; ii) *templates* de código, ou; iii) no próprio código fonte gerado, passível de ser sobrescrita em outras transformações (Völter; Bettin, 2004). As particularidades da tecnologia utilizada normalmente são implementadas diretamente no código fonte, sem apoio dos modelos especificados. No caso de aplicações Web, a implementação *client-side* é feita de forma *ad-hoc* (Kroiss *et al*, 2009).

A implantação de MDD em um processo de desenvolvimento de software Web não é trivial (Fernandes; Melnikoff, 2012). Os estudos recentes com MDD dão enfoque em transformações entre modelos e fornecem abordagens para facilitar a sua utilização para desenvolver software. Existem ferramentas adequadas para transformações de modelos para código, porém com objetivo específico em geração de código estrutural e não se preocupam com a geração e reutilização do código de lógica de negócio (Fu *et al*, 2011).

As abordagens de implantação MDD, em sua grande maioria, necessitam automatizar a transformação entre modelos de alto nível de abstração e modelos geradores de código. A utilização de modelos gráficos pode ser útil para a visualização de alguns domínios e especificação de aplicações, porém a realização de transformações entre eles não é trivial, uma vez que:

- Modelos gráficos normalmente utilizam diagramas UML 2.0 (OMG, 2012c), diferenciando-se por meio de estereótipos e variações;

- Ferramentas prontas de transformação entre modelos normalmente dependem de *IDEs* (ambientes de desenvolvimento integrado), como por exemplo Eclipse, WebRatio ou Fujaba, e para transformações automáticas entre modelos, por exemplo, é necessário conhecimento em linguagens como ATL (Obeo, 2012b) ou QVT (OMG, 2012a);

- Essas ferramentas de transformação geram código fonte a partir de *templates*, por meio de *plug-ins* da plataforma Eclipse (Eclipse, 2012), como por exemplo, JET (Java Emitter Pages, 2012) ou Acceleo (Obeo, 2012a);

- Os desenvolvedores de software necessitam conhecer várias linguagens, como as citadas anteriormente, para utilizar MDD.

Como o desenvolvimento de aplicações Web comumente é ágil e com publicações freqüentes, as abordagens MDD devem ser flexíveis para que se adaptem a esse contexto.

1.3 Objetivos

A partir do contexto e da motivação, neste projeto é desenvolvida uma abordagem evidenciando que o desenvolvimento orientado a modelos, para softwares Web, pode ter métodos simples de aplicação e ser independente de plataformas de programação. Para isso, podem ser utilizados modelos expressos em linguagens descritivas, ou mesmo linguagens de programação, a fim de reduzir o esforço de implementação de transformações entre modelos. Dentre os objetivos propostos para este trabalho pode-se citar:

- Desenvolver uma abordagem que facilite a utilização de MDD por desenvolvedores de software Web, por meio de modelos genéricos e atividades consolidadas em modelos de processos tradicionais;

- Construir metamodelos para cada modelo proposto pela abordagem desenvolvida, de modo a auxiliar os desenvolvedores sem experiência em MDD;

- Implementar uma ferramenta que utiliza modelos textuais, baseados nos metamodelos descritos, que forneça transformadores automáticos entre esses modelos, de forma a apoiar o uso de MDD para construção de aplicações Web desenvolvidas na linguagem Ruby e com apoio do framework Ruby on Rails;

- Desenvolver aplicações exemplificando o uso da ferramenta desenvolvida;

- Fornecer diretivas que apóiem o reuso de software, por meio de linguagens de padrões, frameworks e linhas de produtos de software, em conjunto com a abordagem desenvolvida.

Com isso, dentre os benefícios alcançados, pode-se citar:

1. Aumento de produtividade do engenheiro de software no desenvolvimento de aplicações Web em comparação com o desenvolvimento tradicional;
2. Maior facilidade na manutenção de softwares Web, pois as técnicas utilizadas fornecem reuso e conhecimento detalhado do domínio e da aplicação desenvolvida;
3. Aumento da qualidade das aplicações Web desenvolvidas, pois as ferramentas construídas são previamente testadas e apóiam o reuso e testes automatizados de software;
4. Maior facilidade na utilização de MDD para construir aplicações Web, de acordo com as diretivas fornecidas;
5. Construção de uma interface de usuário agradável que pode ser usada em aplicações reais;
6. Geração rápida de protótipos de aplicações Web;
7. Redução de esforços, tempo e custos no desenvolvimento de aplicações Web em comparação com o desenvolvimento tradicional de software Web.

1.4 Organização do trabalho

Esta dissertação está organizada em nove capítulos.

No Capítulo 1 foram mostrados o cenário atual envolvendo MDD e desenvolvimento Web, a motivação e os objetivos deste trabalho.

No Capítulo 2 são apresentadas as tecnologias *client-side* utilizadas no desenvolvimento de aplicações Web. Além disso, são apresentados conceitos de reutilização de software, como por exemplo, linguagens de padrões, frameworks e linhas de produtos de

software. Uma explicação detalhada sobre a linguagem Ruby e o framework Ruby on Rails, utilizados na construção das ferramentas propostas. Por fim, são apresentados os trabalhos relacionados, comparando com este trabalho.

No Capítulo 3 são apresentados os conceitos de desenvolvimento guiado por modelos e suas técnicas de utilização mais conhecidas: MDA e Eclipse EMF/Ecore.

No Capítulo 4 é apresentada a abordagem criada para o desenvolvimento de software Web guiado por modelos, denominada MDWA. O fluxo de etapas e os artefatos gerados durante o desenvolvimento com a utilização da abordagem e como efetuar combinações com outras formas de reuso, também são discutidos.

No Capítulo 5 é apresentada uma ferramenta, denominada Ruby-MDWA, que apóia a implementação dos modelos propostos pela abordagem MDWA e seus transformadores, com base na linguagem Ruby e no framework Ruby on Rails.

No Capítulo 6 são apresentados dois experimentos conduzidos com alunos de graduação dos cursos de Bacharelado em Ciência da Computação e Engenharia da Computação e de mestrado em computação da Universidade Federal de São Carlos, com a finalidade de verificar os ganhos e as limitações da ferramenta Ruby-MDWA comparando o desenvolvimento de sistemas Web sem o apoio dessa ferramenta.

No Capítulo 7 é descrito um estudo de caso real realizado em conjunto com uma empresa de software da cidade de São Carlos. Nesse estudo, três desenvolvedores com experiência em desenvolvimento Web, framework Ruby on Rails e linguagem Ruby construíram um sistema de gerenciamento de projetos com apoio da ferramenta Ruby-MDWA. O objetivo do estudo foi avaliar a aplicabilidade da ferramenta em ambiente comercial, bem como utilizar o conhecimento obtido para efetuar melhorias na ferramenta.

No Capítulo 8 são apresentadas as conclusões a respeito do trabalho desenvolvido e discutidas as contribuições e limitações do mesmo. Também são apresentadas sugestões de trabalhos futuros que podem melhorar e evoluir a abordagem aqui desenvolvida.

2 Revisão Bibliográfica

2.1 Considerações iniciais

Neste capítulo, são apresentados os conceitos e tecnologias utilizadas como embasamento para este trabalho juntamente com alguns trabalhos relacionados. Os conceitos usados nesta dissertação envolvem: padrões, linguagens de padrões, frameworks e as tecnologias envolvidas no desenvolvimento Web. Além desses conceitos, as ferramentas desenvolvidas, durante este projeto, usaram a linguagem de padrões Ruby e os frameworks Ruby on Rails, para implementação de arquitetura e *middleware*. No Capítulo 3, os conceitos de MDD e duas de suas abordagens mais conhecidas, MDA (OMG, 2012b) e Eclipse Ecore/EMF (Steinberg *et al*, 2008) são apresentados com maior detalhamento.

Além desta introdução, este capítulo está organizado em cinco seções. Na Seção 2.2 são abordadas as tecnologias *client-side* utilizadas no desenvolvimento de software Web. Na Seção 2.3 são apresentadas a linguagem Ruby, o framework Ruby on Rails e a utilização de *templates* ERB. Na Seção 2.4 são tratados dos conceitos de reutilização de software e são apresentados padrões, linguagens de padrões, linhas de produtos de software e frameworks. Na Seção 2.5 são comentados alguns dos trabalhos relacionados e apresentado um comparativo com a proposta desta dissertação. Na Seção 2.6 são expostas as considerações finais.

2.2 Desenvolvimento de software Web

O desenvolvimento de aplicações Web pode seguir a mesma metodologia usada para o desenvolvimento em outras plataformas. Porém, por serem baseadas na arquitetura cliente-servidor, as aplicações Web são implementadas com tecnologias *client-side* (executadas no navegador) e *server-side* (executadas no servidor) (W3C, 2012), e as interações com o usuário são feitas por intermédio de requisições HTTP, via navegador Web.

As tecnologias *server-side* são linguagens de programação usadas também em outras plataformas, como Java (Java, 2012), Ruby (Ruby, 2012), Python (Python, 2012), entre outras. Quando o cliente faz uma requisição HTTP a um sistema Web, um software servidor coleta informações da requisição e designa à aplicação, que faz o processamento com as linguagens *server-side* e envia a resposta em linguagem *client-side*.

As linguagens *client-side* são interpretadas pelo navegador e compõem a parte visual de um software Web. Nas subseções seguintes são apresentados os conceitos referentes às tecnologias *client-side* HTML, Javascript, CSS e Ajax.

2.2.1 HTML – *Hyper Text Markup Language*

HTML é a linguagem que descreve a parte estrutural de páginas Web (W3C, 2012). Páginas HTML são interpretadas e executadas diretamente pelo navegador e consistem em um conjunto de elementos organizados em estrutura de árvore, com a sintaxe muita próxima ao XML (W3C, 2012). Inclusive, XHTML é uma iniciativa recente na qual os documentos HTML devem seguir a sintaxe XML (W3C, 2012), como forma de facilitar a interpretação por parte dos navegadores.

Com HTML, o desenvolvedor pode criar páginas que apresentam recursos multimídia de forma estática. Quando aliada a linguagens de script *server-side*, páginas HTML podem apresentar recursos que dependem de processamento, como resultados de uma busca a uma base de dados, por exemplo. Na Figura 2.1 é ilustrado um exemplo de código HTML e na Figura 2.2 é exibida a execução desse código no navegador.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
      <title>HTML CSS - W3C</title>
      <!-- Comentário HTML -->
      <link rel="stylesheet" href="style.css" type="text/css" />
      <script src="script.js" type="text/javascript"></script>
    </head>
    <body>

        <div class="html_formatting">
            <h1>HTML</h1>
            <p>
                <a href="http://w3c.org/html">HTML</a> is the language for describing the structure of Web pages. <br />
                HTML gives authors the means to:
            </p>
            <ul>
                <li>Publish online documents with headings, text, tables, lists, photos, etc.</li>
                <li>Retrieve online information via hypertext links, at the click of a button.</li>
                <li>Design forms for conducting transactions with remote services, for use in searching for information,
                    making reservations, ordering products, etc.</li>
                <li>Include spread-sheets, video clips, sound clips, and other applications directly in their documents.</li>
            </ul>
        </div>

    </body>
</htlm>
```

Figura 2.1: Exemplo de código HTML.

HTML

HTML is the language for describing the structure of Web pages.
HTML gives authors the means to:

- Publish online documents with headings, text, tables, lists, photos, etc.
- Retrieve online information via hypertext links, at the click of a button.
- Design forms for conducting transactions with remote services, for use in searching for information, making reservations, ordering products, etc.
- Include spread-sheets, video clips, sound clips, and other applications directly in their documents.

Figura 2.2: Exemplo de código HTML no navegador.

HTML 5 (W3C, 2012) recebeu mudanças significativas para auxiliar a criação de código semântico, com novos elementos e apresentação de conteúdo multimídia com desenhos, vídeos e áudio. Anteriormente, essas funções eram providas somente por softwares proprietários. Essa nova iniciativa tem a finalidade de melhorar a navegação do usuário, padronizar a distribuição de conteúdo multimídia e facilitar a indexação de páginas por mecanismos de busca.

2.2.2 CSS – *Cascading Style Sheets*

CSS é a linguagem para descrição da forma como os elementos HTML são apresentados em uma página, possibilitando a escolha de cores, posicionamento no layout, fontes, bordas, entre outros (W3C, 2012). Os arquivos CSS, ou arquivos de estilo, são incluídos em um documento HTML usando o elemento *link* e é possível descrever código CSS diretamente dentro de um documento HTML utilizando o elemento *style*.

A linguagem CSS independe do HTML e pode ser usada para formatar documentados baseados em XML (W3C, 2012). O documento HTML, por exemplo, apresenta uma tabela de dados e o CSS define a borda, disposição das células, cor e fonte dos textos.

A sintaxe do CSS é bastante simples e se baseia em três pontos: seletor, regra e valor. O seletor seleciona quais elementos serão afetados pela definição dos valores das regras. Uma regra é um atributo específico de cada tipo de elemento e possui um conjunto de valores válidos. Cada navegador tem seu conjunto de regras e valores específicos, gerando problemas de compatibilidade de código entre navegadores.

Na Figura 2.3 é mostrado um exemplo de código CSS, enquanto que na Figura 2.4 é ilustrado o código HTML da Figura 2.1, utilizando o estilo definido na Figura 2.3.

```
/*
* Exemplo de código CSS
*/
seletor {
    regra: valor;
    regra: valor;
}

body {
    font-family: 'Helvetica Neue', Helvetica, Arial, sa
    color: #222;
}

h1 {
    color: #036;
    margin: 10px 0;
}

a {
    color: #036;
    text-decoration:none;
}

p {
    padding-left: 20px;
}

ul li {
    margin-left: 20px;
}
```

Figura 2.3: Exemplo de código CSS.

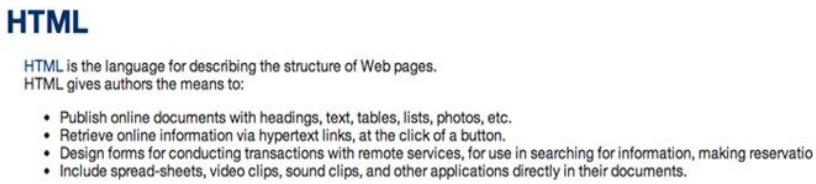


Figura 2.4: Código HTML da Figura 2.1 formatado por CSS.

As regras que definem os elementos são herdadas de seus elementos-pai. Na Figura 2.3, por exemplo, o seletor *body* formata o elemento HTML *body* e todos os outros elementos da página, pois estão contidos dentro do *body*. Essa característica da linguagem que se refere ao *cascading* (cascata em português) que o CSS leva em seu nome.

2.2.3 Javascript

Javascript (W3C, 2012) é uma linguagem para criação de scripts *client-side* que interagem com páginas Web e com o comportamento do navegador. Tem a capacidade de manipular o documento HTML em uma estrutura de árvore, denominada DOM (*Document Object Model*), tratar eventos de mudanças na página, carregar conteúdo remoto em chamadas AJAX e alterar regras CSS. Sua sintaxe é parecida com a linguagem Java, porém é interpretada e dinamicamente tipada.

Assim como o CSS, a linguagem Javascript também independe do documento HTML. Essa separação reflete a preocupação do W3C (2012) em especificar linguagens diferentes para estruturação de dados (HTML), apresentação (CSS) e dinâmica da página (Javascript).

2.2.4 AJAX – *Asynchronous Javascript And XML*

AJAX (W3C, 2012) é uma técnica para o envio de requisições HTTP de forma assíncrona, utilizando Javascript. Seu objetivo é diminuir o tempo de carregamento de recursos e melhorar a navegação do usuário com a aplicação Web, pois não é necessário

recarregar o conteúdo HTML completamente, tornando o tempo de resposta menor (Riordan, 2008).

Requisições assíncronas podem acontecer a qualquer momento da execução de uma página HTML. AJAX envolve a utilização de todas as tecnologias *client-side*, uma vez que a chamada é feita via Javascript e a resposta do servidor pode ser uma representação de dados, com XML (W3C, 2012) ou JSON (JSON, 2012), ou mesmo uma parte de uma página HTML. Também é possível criar mecanismos para que a resposta de uma chamada AJAX execute código Javascript processado no servidor.

2.3 Desenvolvimento Web usando Ruby e Ruby on Rails

Ruby on Rails (Rails, 2012), ou simplesmente Rails (Geer, 2006), é um framework *open-source*, desenvolvido na linguagem Ruby (Ruby, 2012), que tem a finalidade de aumentar a produtividade dos desenvolvedores e facilitar a construção de aplicações Web de alta qualidade (Bachle; Kirchberg, 2007). Para isso, Rails integra um conjunto de componentes da linguagem Ruby, criados pela comunidade desenvolvedora *open-source* da linguagem, e utiliza convenções para a ligação entre esses componentes (Geer, 2006), de forma que o esforço em configurar o comportamento das aplicações é reduzido.

Na Subseção 2.4.1 são apresentados conceitos básicos da linguagem Ruby; na Subseção 2.4.2 são descritas as funções fornecidas pelo framework Ruby on Rails; na Subseção 2.4.3 é apresentada a sintaxe de uso dos *templates* Ruby ERB, que são utilizados pela ferramenta Ruby-MDWA.

2.3.1 A linguagem Ruby

Ruby é uma linguagem de programação com enfoque em simplicidade e produtividade, tornando sua leitura de fácil compreensão (Ruby, 2011). Trata-se de uma linguagem interpretada, ou seja, que não é compilada antes de executar, e possui as seguintes características:

i) **Totalmente orientada a objetos:** todas as variáveis e referências são objetos, cujos tipos são classes, não existindo tipos primitivos. Inclusive, ponteiro nulo, representado por *nil*, é um objeto da classe *NilClass*;

ii) **Tipagem dinâmica:** os tipos de métodos são definidos de acordo com seu retorno e os tipos de variáveis são definidos de acordo com seus objetos;

iii) **Fortemente tipada:** todas as variáveis têm tipos bem definidos, identificados pelo interpretador em tempo de execução. Por isso, não existe modificação do tipo das variáveis, ocasionando erros, por exemplo, ao tentar somar um inteiro com uma *string*.

iv) **Multiparadigma:** todos os tipos são orientados a objetos, a construção de programas pode ser feita com *scripts*, caracterizando Ruby como uma linguagem imperativa, e oferece suporte programação funcional por meio de blocos.

A sintaxe da linguagem Ruby é similar à das linguagens SmallTalk (SmallTalk, 2012) e Perl (Perl, 2012). Na Figura 2.5 são ilustrados exemplos de utilização de algumas estruturas de controle da linguagem Ruby. Nas Figuras Figura 2.6 e Figura 2.7 são apresentados dois exemplos da sintaxe de Ruby para a codificação de classes, respectivamente.

```ruby
def print?
  return true
end

i = 0
while i < 10
  if print?
    puts 'Esse é um parâmetro de um método'
  end
  i = i + 1
end
```

```ruby
10.times do |index|
  if index%2 == 0
    puts index
  end
end
```

```ruby
for i in 0..10 do
  case i
  when 1 then
    puts 'um'
  when 2 then
    puts 'dois'
  else
    puts 'não é 1 nem 2'
  end
end
```

Figura 2.5: Exemplos de estruturas de controle em Ruby.

```ruby
class Pessoa
  attr_accessor :nome, :endereco

  def initialize(nome = nil, endereco = nil)
    self.nome = nome
    self.endereco = endereco
  end

  def Pessoa.pessoa_fisica(nome = nil, endereco = nil, cpf = nil)
    PessoaFisica.new nome, endereco, cpf
  end

  def Pessoa.pessoa_juridica(nome = nil, endereco = nil, cnpj = nil)
    PessoaJuridica.new nome, endereco, cpnj
  end

end
```

Figura 2.6: Exemplo de código Ruby que representa uma classe.

```ruby
class PessoaFisica < Pessoa
  attr_accessor :cpf

  def initialize(nome = nil, endereco = nil, cpf = nil)
    super nome, endereco
    self.cpf = cpf
  end
end

class PessoaJuridica < Pessoa
  attr_accessor :cnpj

  def initialize(nome = nil, endereco = nil, cnpj = nil)
    super nome, endereco
    self.cnpj = cnpj
  end

end
```

Figura 2.7: Exemplo de código Ruby que representa duas classes com herança.

Duas características que Ruby apresenta são blocos e módulos. Blocos (*blocks*, em inglês) são trechos de código, similares aos blocos de código aplicados a comandos *if* e *while* em outras linguagens, que podem manipulados de maneira independente, sendo passados como parâmetros para métodos ou setados em variáveis. Essa característica possibilita a organização de informações de forma aninhada. A ferramenta Ruby-MDWA, descrita no Capítulo 5, utiliza blocos Ruby para definição de uma DSML textual que representa modelos de software. Na Figura 2.8 é apresentado um exemplo de utilização de blocos Ruby, por meio da utilização dos métodos *times* e *each*, respectivamente. Os blocos são definidos dentro dos elementos *do* e *end* e seus parâmetros são identificados após o *do*, dentro de *pipes* (“|”).

```ruby
10.times do |i|
  puts "Times: #{(i + 100)}"
end

search_engines = ['Google', 'Yahoo', 'Bing', 'UOL']
search_engines.each do |engine|
  puts "Buscadores: http://" + engine.downcase + ".com"
end
```

Figura 2.8: Exemplo de utilização de blocos em Ruby.

Módulos (*modules*, em inglês) são conjuntos de métodos e variáveis, e possibilitam encapsular comportamentos e funções para reutilização. Além disso, um módulo pode incluir seus métodos dentro de classes, estendendo suas funções. Esse comportamento é denominado *mixin*. Por exemplo, é possível encapsular um requisito de persistência em um módulo e

acrescentar o comportamento de persistência diretamente em classes, conforme ilustrado na Figura 2.9. Nesse exemplo, é apresentado o módulo "Persistencia" que possui os métodos "salvar" e "recuperar" e sua utilização pela classe "Livro" através da chamada *include*, bem como as chamadas dos métodos de persistência a partir da classe "Livro".

```ruby
module Persistencia

  # métodos de objetos
  def salvar
    # ...
  end

  # métodos da classe
  module ClassMethods

    def recuperar(id)
      # ...
    end

  end

  # Chamado quando o módulo é incluído em alguma classe.
  def self.included(base)
    # Inclui os métodos da classe.
    base.send :extend, ClassMethods
  end

end
```

```ruby
class Livro

  include Persistencia
  # ...
end

livro = Livro.new
livro.salvar
livro2 = Livro.recuperar(333)
```

Figura 2.9: Exemplo de módulo em Ruby, encapsulando o comportamento de persistência.

2.3.2 O Framework Ruby on Rails

Ruby on Rails, ou simplesmente Rails, como é chamado por sua comunidade de desenvolvimento, é um framework *open-source* que apóia o desenvolvimento ágil de aplicações Web. Foi desenvolvido em Ruby e liberado em 2004, sob a licença de software MIT (MIT, 2012). Trata-se de um framework do tipo caixa-branca, por se basear nos mecanismos de herança e ligação dinâmica para instanciação de aplicações (Fayad, 1997).

Com intuito de aumentar a produtividade dos desenvolvedores, Rails se baseia em dois princípios: DRY (*Don´t Repeat Yourself*, em inglês) e convenção sobre configuração.

O conceito de DRY é evitar que os desenvolvedores realizem retrabalho. Para isso o framework provê geradores de código básico, em um mecanismo denominado *scaffold*, ferramentas de linha de comando e tem mecanismos para evitar a redundância de qualquer trecho de código. Por exemplo, os atributos de uma classe que são persistidos em banco de dados ficam disponíveis de forma implícita para sua classe e não precisam ser declarados.

A premissa de convenção sobre configuração prevê que os componentes e os comportamentos de uma aplicação devem se comunicar sem a necessidade de configuração (Barchle; Kirchberg, 2007). Todas as convenções que o framework assume podem ser

customizadas pelo desenvolvedor, possibilitando que implemente novas funções. Além disso, Rails fornece, por padrão, a criação de três ambientes: desenvolvimento, testes e produção. Dessa forma, o desenvolvedor pode efetuar configurações especificas para cada ambiente, como por exemplo, separar as bases de dados, evitando perda de dados do ambiente de produção ou criando rotinas automatizadas para o ambiente de testes.

Sua arquitetura é baseada no padrão arquitetural MVC (*Model-View-Controller*, em inglês) (Gamma *et al*, 1995) em conjunto com *helpers*, com a finalidade de fazer a ligação entre as ações do usuário, a lógica da aplicação e a apresentação (Viwanathan, 2008), separando a camada de negócio da apresentação e controle. *Helpers* são representados por módulos Ruby e tem a finalidade de encapsular métodos acessores, evitando o espalhamento e repetição de código pelas *views*.

Quando uma requisição é feita a uma aplicação Rails, ela é tratada pelo controle de rotas (Rails *Router*), que interpreta a requisição e seleciona o *controller* MVC e ação solicitados. Então, o *controller* MVC faz a decodificação dos parâmetros da requisição e o processamento dos dados de apresentação e repassa o controle para a *view* MVC apropriada, podendo redirecionar para outra página ou apresentar uma *view* MVC com o mesmo nome da ação, seguindo a convenção estabelecida pelo framework. Um *controller* MVC consiste de uma classe e um conjunto de métodos, que representam os nomes das ações. A camada de apresentação, por sua vez, é gerenciada pelo componente *Action View* (Geer, 2006), que faz a geração das páginas da interface gráfica da aplicação. Essas páginas consistem basicamente de código HTML, Javascript, estilos CSS e *scripts* Ruby.

A camada de negócio é gerenciada pelo componente *Active Record*, um pacote que efetua o mapeamento objeto-relacional (ORM, *Object-Relational Mapping*, em inglês) entre as classes do modelo (*models* MVC) e suas tabelas no banco de dados (Bachle; Kirchberg, 2007). ORM é uma abordagem para tentar diminuir as diferenças entre as entidades da orientação a objetos e os bancos de dados relacionais, e facilitar a persistência de objetos (Van Zul *et al*, 2006). Outra característica do mapeamento objeto-relacional é o mecanismo de *migrations*. Uma *migration* é um trecho de código Ruby que pode criar ou alterar tabelas, colunas ou inserir diretamente código SQL na base de dados. Esse mecanismo possibilita o controle de versões das alterações de banco de dados, pois o framework mantém um histórico das *migrations* executadas e das que estão pendentes.

A conexão entre os componentes que implementam a arquitetura MVC e os *helpers* é efetuada de acordo com o princípio de convenção sobre configuração, conforme é exibido na Tabela 2.1.

Tabela 2.1: Exemplo do princípio de convenção sobre configuração do framework Ruby on Rails para um cadastro de produtos.

Componente do framework	**Convenção de utilização**
Classe e arquivo do *model* MVC	Product e app/models/product.rb
Tabela do banco de dados	Products
Classe e arquivo do *controller* MVC	ProductsController e app/controllers/ products_controller.rb
Pasta que encapsula *views* MVC	app/views/products
Nome do arquivo e módulo *Helper*	app/helpers/products_helper.rb e ProductsHelper
Associações entre *models* N para 1	belongs_to :product
Chave estrangeira no banco de dados	product_id
Associações entre *models* 1 para N	has_many :products
Roteador de URLs	resources :products

Rails oferece suporte para a execução de testes automatizados, por meio da construção de testes unitários, funcionais e de integração. Teste unitário é um método para testar módulos de do comportamento funcional da aplicação. No contexto do framework, testes unitários são aplicados nas classes da camada de negócio, os *models* MVC. Teste funcional é identificado como a análise de dados de entrada e saída de formulários funcionais. No Rails, testes funcionais são usados para avaliar o comportamento dos *controllers* MVC. O objetivo de testes de integração é garantir que todos os componentes se relacionem corretamente com a interface, isto é, garantir que não existam falhas na interface gráfica. No Ruby on Rails, os testes de integração garantem que as páginas HTML sejam apresentadas conforme o esperado.

Por ser independente de IDE (ambiente de desenvolvimento integrado), Rails possui ferramentas de linha de comando que executam diferentes funções de auxílio ao desenvolvedor. Na Tabela 2.2 são apresentados alguns desses comandos.

Tabela 2.2: Lista de comandos de terminal do framework Ruby on Rails.

Comando	**Execução**
rails new <nome da aplicação>	Instancia uma nova aplicação.
rails Server	Inicializa o servidor de desenvolvimento.
rails generate <nome do gerador>	Executa tarefas de geração de código.
rails console	Inicia um console para teste de código e funcionalidade.
rake <nome da tarefa>	Executa tarefas de sistemas, como, por exemplo, atualização do esquema de banco de dados por meio de *migrations* (rake db:migrate), ou execução de testes automatizados (rake test).
rails dbconsole	Abre o console do banco de dados da aplicação.
Bundle	Executa ações que atualizam a lista de dependências da aplicação.

2.3.3 *Templates* ERB

Templates são gabaritos que aceitam linguagens de *scripts*, como Ruby, por exemplo, com a intenção de gerar documentos. A biblioteca ERB da linguagem Ruby fornece um mecanismo para a construção de *templates*, possibilitando que qualquer código Ruby seja incluído em documentos de texto com o propósito de gerar informações ou controlar o fluxo de execução (Ruby, 2012). *Templates* podem ser usados para gerar qualquer formato de informações em texto.

No framework Ruby on Rails, *templates* são utilizados para gerar código HTML dinamicamente nas *views* MVC. Outra aplicação é a geração de arquivos Javascript dinamicamente como resposta de requisições AJAX. A ferramenta Ruby-MDWA utiliza *templates* ERB para criar *templates* de código, responsáveis pela geração de código e transformações entre modelos, conforme é descrito no Capítulo 5.

Nos *templates* ERB, o código Ruby é especificado por meio das *tags* **<%**, **<%= e %>**, e as demais informações do documento são lidas como informações estáticas, que não serão processadas. Nas Figuras Figura 2.10 e Figura 2.11 são apresentados *templates* ERB que renderizam código Javascript e HTML, respectivamente.

```
$.fancybox.close(true);
$("#avisos-index").html("<%= escape_javascript( render :partial => 'avisos' )%>");
$("#notice").html("<%= escape_javascript( render "/template/mdwa/notice", :notice => @mensagem )%>");
```

Figura 2.10: *Template* ERB que renderiza código Javascript.

```
<p id="notice"><%= notice %></p>

<p>
  <b>Descricao:</b>
  <%= @aviso.descricao %>
</p>

<p>
  <b>Remetente:</b>
  <%= @aviso.remetente_id %>
</p>

<p>
  <b>Lido:</b>
  <%= @aviso.lido %>
</p>


<%= link_to 'Edit', edit_aviso_path(@aviso) %> |
<%= link_to 'Back', avisos_path %>
```

Figura 2.11: *Template* ERB que renderiza código HTML.

2.4 Reutilização de software

Uma das maneiras bem sucedidas de se desenvolver software com qualidade e de forma rápida é reutilizando experiências anteriores. Dessa maneira, o esforço de desenvolvedores em construir software é amenizado (Mili *et al*, 1995). Para isso, pode-se reusar conhecimentos adquiridos em praticamente todos os níveis de abstração, etapas e artefatos de um software (Viana, 2009).

Nesta Seção são apresentadas três formas que podem ser utilizadas no desenvolvimento com base em reutilização de software: o conceito de Padrões e Linguagens de Padrões é apresentado na Subseção 2.3.1; na Subseção 2.3.2 é descrita a reutilização por meio de Linhas de Produtos de Software; e na Subseção 2.3.3 são apresentados conceitos a reuso por meio de Frameworks.

2.4.1 Padrões e Linguagens de Padrões

Um padrão de software é uma solução de sucesso para um determinado problema em um contexto particular, descreve os esforços despendidos e os benefícios obtidos ao aplicá-lo (Gamma et al, 1995), fornecendo reuso de experiência (Fowler, 1997). Com o conhecimento obtido na solução de problemas que ocorrem no desenvolvimento de software, os desenvolvedores tendem a reutilizar soluções de sucesso do passado para resolver problemas de projeto atuais (Braga, 2003). Dessa forma, projetistas de software podem aplicar padrões para a solução de problemas atuais sem ter que reinventá-los.

Um conjunto de padrões de um domínio específico que tem dependências entre si constitui uma linguagem de padrões (Coplien, 1996). Uma linguagem de padrões representa o conhecimento de um domínio particular a partir da experiência dos desenvolvedores em resolver problemas similares desse domínio (Brugali; Sycara, 2000).

A diferença básica entre uma linguagem de padrões e um conjunto de padrões relacionados é que as linguagens de padrões são compostas por padrões fortemente relacionados a ponto de não existirem de maneira independente (Viana, 2009), enquanto padrões compostos se propõem a resolver problemas isolados.

As linguagens de padrões podem ser empregadas como guias durante o desenvolvimento de aplicações de um mesmo domínio (Braga, 2003). O desenvolvimento de software com apoio de linguagens de padrões facilita o planejamento e implementação de requisitos já previstos na especificação da linguagem, reduzindo o tempo de desenvolvimento.

2.4.2 Linhas de Produtos de Software

O termo Linha de Produto de Software (LPS), em inglês, *Software Product Lines*, remete às linhas de produção da indústria que introduziram a idéia de produção seqüencial de artefatos, baseada em tarefas repetitivas (Durski *et al*, 2004). Na Engenharia de Software, uma linha de produtos de software é um conjunto de sistemas que compartilham um conjunto de características comuns e gerenciadas que satisfazem as necessidades de um domínio específico (Clements; Northrop, 2002). Dessa forma, uma LPS é constituída de um núcleo, composto das características comuns a todos os produtos da linha, e de um conjunto de variabilidades específicas de cada produto.

O desenvolvimento de uma Linha de Produtos de Software é dividido em duas etapas: engenharia de domínio e engenharia de aplicação (Gomaa, 2005). Na etapa de engenharia de domínio é efetuada a construção do núcleo de características da linha de produtos. A engenharia de domínio é dividida em três atividades básicas (Ziadi *et al*, 2003): análise, projeto e implantação. Na atividade de análise deve-se definir os pontos comuns e os variáveis da linha. Na atividade de projeto, especifica-se a maneira como os pontos comuns e variáveis serão implementados. Na atividade de implementação, são construídos os artefatos da linha de produtos. De forma complementar ao processo de engenharia de domínio, também pode-se construir ferramentas para auxiliar a engenharia de aplicações (Weiss; Lai, 1999).

O processo de engenharia de aplicações consiste no agrupamento dos artefatos construídos na etapa de engenharia de domínio objetivando o desenvolvimento de produtos de acordo com as necessidades de cada cliente. Nesse processo deve-se identificar os artefatos que correspondem às características variáveis e quais correspondem às características do núcleo da linha.

2.4.3 Frameworks

Um framework é um projeto reutilizável, por completo ou em parte no desenvolvimento de sistemas, que é representado por um conjunto de classes abstratas e por um modelo que representa os comportamentos de suas instâncias (Johnson, 1997a). Todas as aplicações instanciadas de um framework apresentam características similares, diferenciando-se em seu comportamento, que varia conforme a necessidade da aplicação.

Os frameworks permitem a adaptação do código específico da aplicação, o que facilita sua aplicabilidade (Johnson, 1997a).

Frameworks fornecem reuso de código e de projeto (Viana, 2009) em um domínio específico. Para isso, as classes de um framework são dependentes entre si e não podem ser utilizadas separadamente (Johnson, 1997b), diferentemente de uma biblioteca de classes. Isso permite que o desenvolvedor fique livre de cuidar dos aspectos comuns de um domínio, aumentando sua produtividade. Outra característica que diferencia um framework de uma biblioteca de classes é que os frameworks ditam o fluxo de execução, pois é o framework quem invoca o código da aplicação (Braga, 2003). Esse conceito é denominado inversão de controle (Johnson, 1997b) ou Princípio de Hollywood (Larman, 2004). Essa inversão de controle torna o framework extensível, pois os desenvolvedores de aplicações podem especializar os algoritmos genéricos de um framework de acordo com os requisitos da aplicação.

Os frameworks possuem pontos fixos (em inglês, *frozen spots*) e pontos variáveis (em inglês, *hot spots*) (Brugali; Sycara, 2000). Os pontos fixos são formados por classes concretas e representam a funcionalidade do software já desenvolvida, testada e pronta para ser reutilizada. Esses pontos definem os comportamentos imutáveis de um framework. Os pontos variáveis representam a parte utilizada pelo desenvolvedor para instanciação de acordo com os requisitos da aplicação.

O acesso aos pontos variáveis indica se um framework é caixa branca, preta ou cinza. Quando o acesso é feito via especialização, o framework é do tipo caixa branca. Caso o acesso seja feito por meio de associação, agregação ou composição, o framework é do tipo caixa preta. Quando esse acesso pode ser feito das duas maneiras, o framework é caixa cinza (Abi-Antoun, 2007). Os mais fáceis de construir e mais flexíveis são os frameworks do tipo caixa branca. Porém, esses são os mais difíceis de usar, pois exigem maior conhecimento por parte do desenvolvedor sobre suas classes. Já com os frameworks caixa preta ocorre exatamente o inverso (Johnson, 1997b).

O escopo de um framework também pode ser classificado como (Fayad e Schmidt, 1997): i) *System Infrastructure Frameworks* (SIF), ii) *Middleware Integration Frameworks* (MIF) e iii) *Enterprise Application Frameworks* (EAF). Frameworks SIF simplificam o desenvolvimento de infra-estruturas de sistemas como, por exemplo, sistemas operacionais, interfaces com o usuário e compiladores (Braga, 2003). Frameworks MIF são usados para integrar aplicações e componentes de um mesmo domínio. O Ruby on Rails (Rails, 2012) e o Apache Struts (Struts, 2012) são exemplos de frameworks desse tipo. Frameworks EAF tratam da implementação das regras de negócio de um domínio específico, como sistemas de informação, por exemplo. O GRENJ (Durelli, 2008) é um exemplo de framework EAF.

2.5 Trabalhos relacionados

Neste trabalho foi desenvolvida uma abordagem genérica que apóia a utilização de MDD para a construção de aplicações Web. Também foi desenvolvida uma ferramenta, denominada Ruby-MDWA, que a partir de modelos especificados em uma DSML interna textual, fornece mecanismos de transformação, favorecendo o desenvolvimento guiado por modelos de aplicações Web baseadas na linguagem Ruby e no framework Ruby on Rails.

Alguns trabalhos existentes na literatura são semelhantes à forma de desenvolvimento da ferramenta Ruby-MDWA, porém utilizam outras tecnologias para geração de código e se baseiam, especialmente, em MDA (OMG, 2012b). Não foram encontradas ferramentas de apoio à utilização MDD para desenvolvimento com base em Ruby e Ruby on Rails. As ferramentas relacionadas à Ruby-MDWA, apresentadas nesta Seção, utilizam a nomenclatura de abordagem MDD, porém esses trabalhos dão enfoque em modelos e ferramentas de transformação, ao invés da realização de atividades referentes ao desenvolvimento Web

guiado por modelos. A abordagem MDWA tem como ponto central a realização sistemática de atividades para desenvolvimento Web com apoio de MDD, de forma que ao final de cada atividade sejam especificados todos os artefatos necessários.

As abordagens genéricas para apoio ao uso de MDD de outros autores apresentam processos com a finalidade de: i) melhorar a reutilização de software, como, por exemplo, Lucrédio (2009), Czarnecki *et al* (2005) e Knodel *et al* (2005); ii) facilitar a utilização de MDD em um domínio específico, como, por exemplo, Souza e Olivera (2012) e Medeiros (2008); iii) aumentar a automação de processos com MDD, como, por exemplo, Pellegrini *et al* (2010) e Maciel *et al* (2006).

De acordo com a motivação e objetivos deste trabalho, descritos nas Seções 1.2 e 1.3, respectivamente, a utilização de MDD não é trivial, pois requer o conhecimento de várias tecnologias, acrescenta novas etapas no desenvolvimento e aumenta a complexidade de especificação da lógica de negócio. Para que essas dificuldades sejam minimizadas, foram propostas ferramentas que apóiam a utilização de MDD, da mesma forma que é realizado neste trabalho.

Nesta seção são apresentadas as ferramentas de apoio ao uso de MDD para desenvolvimento Web relacionadas à ferramenta Ruby-MDWA. Uma avaliação dessas ferramentas é realizada em relação à independência de plataforma, extensibilidade e simplicidade, com intuito de identificar se esses trabalhos podem satisfazer as motivações deste trabalho.

Castrejón *et al.* (2011) apresentam uma abordagem, denominada *Model2Roo*, que apóia o desenvolvimento Web a partir modelos do framework EMF (Gronback, 2009) e perfis UML 2.0 (OMG, 2012c). Esses modelos descrevem a aplicação e transformações ATL (Obeo, 2012b) geram scripts de linha de comando que invocam ferramentas do framework *SpringRoo*, que, por sua vez, geram código Java (Java, 2012). *Model2Roo* é um *plug-in* da IDE Eclipse (Eclipse, 2012) e visa a automatizar a geração de código estrutural e de código base para a lógica de negócio da aplicação, assim como este trabalho. A diferença é que *Model2Roo* não oferece mecanismos para evitar a sobrescrita de arquivos modificados pelo desenvolvedor, enquanto a ferramenta Ruby-MDWA fornece *templates* para customização e posterior geração de código.

Bernardi *et al.* (2011) apresentam uma abordagem para geração de protótipos rápidos orientados a modelos. Seu trabalho define um metamodelo, baseado no framework Eclipse EMF, para construção de modelos que representam aplicações Web com arquitetura MVC (Gamma *et al*, 1995). O desenvolvimento de protótipos é realizado com a modelagem gráfica da aplicação, a partir de modelos MVC, e transformações usando Xpand (Eclipse, 2012), gerando código para Java e apoio do framework JSF. O trabalho de Bernardi é mencionado como parte de um conjunto de iniciativas para automatização de desenvolvimento usando MDD e é classificado como uma ferramenta para prototipação. Não permite customização de *templates*, como Ruby-MDWA, porém define um metamodelo para a criação de elementos visuais para as páginas Web geradas.

Distante *et al.* (2007) propõem uma abordagem MDA (OMG, 2012b) baseada em modelos UWA (UWA, 2002), para a etapa de levantamento de requisitos na camada CIM, e especifica um metamodelo, denominado UML-MVC, para especificação da camada PIM. Os autores definem regras de transformação dos modelos UWA em UML-MVC e de modelos UML-MVC para PSM. A abordagem proposta é extensa, englobando todas as etapas de desenvolvimento, desde o levantamento de requisitos com UWA até a geração de código, podendo ser utilizado comercialmente, assim como Ruby-MDWA. Ao contrário deste trabalho, a abordagem de Distante *et al* (2007) não permite a customização do código gerado, o que dificulta a inclusão da lógica de negócio e customização das páginas Web.

Rhamouni e Mbarki (2011) apresentam uma abordagem MDA com apoio do framework Eclipse EMF (Gronback, 2009) e da linguagem ATL (Obeo, 2012b). São definidos dois metamodelos, um para UML e outro para a arquitetura MVC2 para modelagem de aplicações. Os modelos UML e MVC2 são transformados em código XMI por meio de transformações ATL. O código XMI é especificado com os detalhes da camada PSM, possibilitando a geração de código para várias tecnologias. Essa abordagem possui enfoque em modelos PIM e PSM e não auxilia todo o processo de desenvolvimento guiado por modelos, de maneira diferente do que Ruby-MDWA realiza.

Fu *et al.* (2011) definem uma metodologia baseada em planejamento usando conceitos inteligência artificial para desenvolvimento de software guiado por modelos. A figura central desse processo é o planejador (*AI-planning*), o especialista no domínio, que cria um plano de ações para cada requisito de um domínio. Esse plano de ações resulta em um modelo de sequência ou atividades que será construído por desenvolvedores e será armazenado em um

repositório de componentes. Além dos requisitos de domínio, são criados modelos funcionais (caso de usos) e estáticos (estrutural, diagrama de classes). O código estrutural é gerado pela ferramenta Rose a partir dos modelos definidos. O código de negócio é gerado a partir dos requisitos de domínio e das implementações construídas pelos desenvolvedores que estão armazenadas como componentes em um repositório. O *AI-planning* é o responsável por selecionar os requisitos de domínio e estruturas para geração de código final da aplicação. A abordagem de Fu *et al.* (2011) engloba todas as etapas de desenvolvimento e define papéis para os envolvidos no desenvolvimento. O especialista de domínio é o condutor do processo de desenvolvimento e o código é gerado de acordo com os modelos de planos de ações. A abordagem favorece o reuso de conhecimento por usar um repositório de requisitos implementados. Apesar desses pontos positivos, essa abordagem tem enfoque nos artefatos gerados e em papéis realizados pelos membros da equipe e não nas atividades a serem desempenhadas, como é proposto na abordagem MDWA. Assim como a ferramenta Ruby-MDWA, essa abordagem pode auxiliar em todo o processo de desenvolvimento e manutenção de software Web, pois a ferramenta Rose permite a customização de seus *templates* de código e pode gerar código em outras tecnologias, ao contrário de Ruby-MDWA.

Eickhof *et al.* (2011) apresentam um processo para desenvolvimento de aplicações, com interface rica e que fazem uso intensivo de AJAX (W3C, 2012), orientado a modelos e guiado por estórias de usuário. O processo de geração de código de interface é apoiado pelo editor de interface GWT-Designer e pela biblioteca Google GWT. A geração de código de negócio é automatizada a partir de modelos de classe e de atividade estereotipados e storyboard exportados pela ferramenta Fujaba. Diferentemente da Ruby-MDWA essa abordagem não apóia todo o processo de desenvolvimento e não permite a customização do código gerado.

Kroiss *et al.* (2009) definem uma abordagem, que utiliza plug-in para a IDE Eclipse UWE4JSF, para o desenvolvimento Web com MDD, baseada na linguagem de modelagem UWE (Koch, 2006). Inicia com a definição de diagramas UML estereotipados, que são posteriormente transformados em modelos UWE, utilizando ATL. Em seguida, os modelos UWE são transformados em modelos JSF, que são interpretados por *templates* para a geração de código final. A abordagem favorece o reuso de modelos e de conhecimento, pois efetua a modelagem específica para Web com UWE e engloba todas as etapas de desenvolvimento, assim como a ferramenta Ruby-MDWA. Essa abordagem conta com apoio ferramental para

criação de modelos gráficos, pois tem como base os modelos UWE, ao contrário da Ruby-MDWA.

UWE4JSF faz parte de uma classe de abordagens MDD que se baseia em modelos de uma linguagem de modelagem Web. Krauss *et al* (2007) especifica regras ATL para transformação dos modelos de conteúdo e apresentação UWE e serve como base para o processo usado por Kroiss (2009).

Trabalhos semelhantes baseados na linguagem de modelagem WebML (WebML, 2012a) também foram propostos. A ferramenta proprietária WebRatio (WebRatio, 2012) apóia o desenvolvimento Web orientado a modelos utilizando diagramas BPMN (OMG, 2012d), diagramas WebML para especificação de detalhes e *templates* XSL (W3C, 2012) para a geração de código. Zhuang e Du (2009) apresentam o desenvolvimento de um software de e-commerce em conjunto com a ferramenta WebRatio (WebRatio, 2012). *Acerbis et al.* (2007) apresenta o desenvolvimento da ferramenta Web Acer-Euro com apoio da ferramenta WebRatio e modelos WebML.

WebML (2012b) apresenta mais trabalhos utilizando WebML e WebRatio. A ferramenta WebRatio tem por base modelos gráficos WebML e BPMN para fornecer uma solução para desenvolvimento guiado por modelos. Por outro lado, seu formato proprietário e sua forma engessada interferem na agilidade no desenvolvimento. Ao contrário de Ruby-MDWA, WebRatio não oferece facilidades para a geração de testes automatizados, podendo causar a redução da qualidade das aplicações geradas.

As abordagens e ferramentas descritas nesta seção são comparadas de acordo com os quesitos: independência de plataforma, extensibilidade e simplicidade, nas Tabelas

Tabela 2.3, Tabela 2.4 e Tabela 2.5, respectivamente.

Na

Tabela 2.3 as abordagens e ferramentas são apresentadas na coluna 1, na coluna 2 tem-se a informação se a abordagem/ferramenta é ou não dependente de IDE; na coluna 3 se os modelos gráficos são com base em GMF ou os diagramas em UML e na última coluna se ocorre a utilização ATL, QVT ou alguma linguagem de transformação em modelos.

Tabela 2.3: Análise de independência de plataforma de desenvolvimento dos trabalhos relacionados.

Ferramenta	**Depende de IDE?**	**Baseado em modelos gráficos GMF ou diagramas UML**	**Utiliza ATL, QVT ou outra linguagem de transformação entre modelos?**
Ruby-MDWA	Não	Não	Não
Castrejon *et al.* (2011)	Sim	Sim	Sim
WebRatio (2012)	Sim	Sim	Não
Kroiss *et al.* (2009)	Sim	Sim	Sim
Rhamouni *et al.* (2011)	Sim	Sim	Sim
Fu *et al.* (2011)	Não	Sim	Não
Eickhof *et al.* (2011)	Sim	Sim	Nao
Bernardi *et al.* (2011)	Sim	Sim	Não

Na Tabela 2.4 as abordagens e ferramentas são apresentadas na coluna 1 e comparadas em relação a sua extensibilidade. Na coluna 2 consta se a abordagem/ferramenta permite outras formas de reuso, como por exemplo, padrões, linguagens de padrões, frameworks ou LPS; na coluna 3 se o código é ou não sobre-escrito durante a geração automática; na coluna 4 se existem *templates* para customização do código gerado e na coluna 5 se essa ferramenta/abordagem pode ser usada com outras tecnologias.

Tabela 2.4: Análise da possibilidade de extensão dos trabalhos relacionados.

Ferramenta	**Permite outras formas de reuso**	**Sobre-escreve o código que é gerado uma vez?**	***Templates* para customização de código?**	**Pode ser usado com outras tecnologias?**
Ruby-MDWA	Sim	Não	Sim	A ferramenta não, mas a abordagem MDWA, sim
Castrejon *et al.* (2011)	Não menciona	Sim	Não	Não
WebRatio (2012)	Sim, de análise	Sim, com templates	Sim	Não
Kroiss *et al.* (2009)	Sim, de análise e padrões UWE / UML	Sim, com templates	Sim	Não
Rhamouni *et al.* (2011)	Não	Sim	Não	Sim
Fu *et al.* (2011)	Sim	Não	Sim	Sim
Eickhof *et al.* (2011)	Sim	Sim	Não	Não
Bernardi *et al.* (2011)	Não	Sim	Não são extensíveis	Não

Na Tabela 2.5 as abordagens e ferramentas são apresentadas na coluna 1 e comparadas em relação a sua simplicidade. Na coluna 2 consta se os modelos da abordagem/ferramenta são baseados em perfis UML; na coluna 3 se o desenvolvedor pode

estender a abordagem/ferramenta com metamodelos customizados. Na coluna 4, o número de etapas que devem ser realizadas até a geração de código fonte; e na coluna 5 se a abordagem/ferramenta é construída com código aberto.

Tabela 2.5: Análise de simplicidade dos trabalhos relacionados.

Ferramenta	**Baseado em perfis UML**	**Desenvolvedor pode desenvolver seu próprio metamodelos?**	**Etapas até o código ser gerado**	**Baseada em código aberto?**
Ruby-MDWA	Não	Sim	Mais rápido: 4 Completo: 7	Sim
Castrejon *et al.* (2011)	Sim	Não	3	Sim
WebRatio (2012)	Não	Somente via Eclise EMF	4	Não
Kroiss *et al.* (2009)	Sim	Somente via Eclipse EMF	4	Sim
Rhamouni *et al.* (2011)	Sim	Não	3	Sim
Fu *et al.* (2011)	Sim	Sim	6	Sim
Eickhof *et al.* (2011)	Sim	Sim	4	Sim
Bernardi *et al.* (2011)	Não	Não	3	Sim

2.6 Considerações finais

O reuso é uma prática que ajuda a obter eficiência e qualidade no desenvolvimento de software. Porém, toda forma de reuso depende fundamentalmente da qualidade dos artefatos reutilizados. Neste capítulo, foram apresentadas técnicas de reuso em níveis mais altos de abstração, como Padrões e Linguagens de Padrões, Linhas de Produtos de Software e Frameworks.

A utilização de frameworks do tipo MIF, como o Ruby on Rails, por exemplo, fornecem um conjunto de classes que implementam comportamentos comuns à plataforma usada. Dessa forma, o desenvolvedor pode direcionar esforços para o domínio da aplicação. O framework Ruby on Rails introduz um conjunto de práticas com o objetivo de melhorar a produtividade dos desenvolvedores e auxiliar a construção de aplicações Web de alta qualidade.

A combinação de desenvolvimento guiado por modelos com as práticas de reuso descritas neste capítulo pode maximizar a qualidade das aplicações construídas e a produtividade de engenheiros de software, por reutilizar experiência e conhecimento em várias etapas de um projeto. No próximo capítulo são apresentados os conceitos de MDD e suas abordagens mais conhecidas: MDA e Eclipse EMF/Ecore.

3 Desenvolvimento de software guiado por modelos

3.1 Considerações iniciais

As técnicas de desenvolvimento de software evoluem continuamente com a finalidade de melhorar processos de construção e manutenção de software e obter aplicações de alta qualidade e baixo custo. O objetivo do MDD é aproximar a distância semântica entre um problema e a especificação de sua solução. Para isso, MDD tem o enfoque na modelagem e na transformação de modelos em artefatos de implementação (Cirilo, 2010).

O desenvolvimento de aplicações Web pode ser facilitado com o uso de desenvolvimento guiado por modelos, em particular quanto à evolução e adaptação dessas aplicações às plataformas e às mudanças tecnológicas. Dessa maneira, os modelos tornam-se mais específicos e completos, e recursos como frameworks, padrões de projeto e linguagens de padrões são incluídos na modelagem com o objetivo de gerar maior quantidade de código e com melhor qualidade (Greenfield *et al*, 2004).

Este capítulo está organizado da seguinte maneira: na Seção 3.2 são apresentados os conceitos, vantagens e desvantagens do desenvolvimento guiado por modelos; na Seção 3.3 são mostradas as duas principais técnicas para utilização de MDD: MDA e Eclipse EMF/Ecore; na Seção 3.4 são apresentadas as considerações finais.

3.2 MDD

Apesar das evoluções alcançadas pela Engenharia de Software, ainda existem muitos problemas a serem resolvidos (Kleppe, 2003), pela forma como o software é construído atualmente: baseado em modelagem de baixo nível de abstração e enfoque em código fonte.

À medida que a complexidade de uma aplicação aumenta, o conhecimento da equipe de desenvolvimento em relação ao seu funcionamento diminui. Isso acontece porque os sistemas sofrem constantes modificações, e em algum momento, os desenvolvedores deixam de atualizar a documentação por ser um esforço a mais, um fardo (Lucredio, 2009). Dessa forma, a modelagem deixa de representar o domínio ou a aplicação desenvolvida. Assim, novos desenvolvedores precisam entender o código fonte da aplicação antes de efetuar manutenções, aumentando o custo de desenvolvimento.

Como o desenvolvimento de software tradicional é um processo semi-artesanal, o esforço de construir e manter uma aplicação é alto, pois desenvolvedores executam tarefas repetitivas manualmente, reduzindo sua produtividade. Dessa forma, a qualidade do software e a velocidade do desenvolvimento podem variar.

Uma alternativa para amenizar esses problemas é a utilização de MDD como método de desenvolvimento de software. MDD (Stahl *et al*, 2006) é uma abordagem que compreende a definição de modelos e a transformação desses modelos em outros artefatos de software (Medeiros, 2008) com a finalidade de prover automação e aumentar a produtividade de desenvolvimento (Cirilo, 2010). Na visão do MDD, modelos são os principais artefatos do software (France; Rumpe, 2007) e suas transformações guiam o desenvolvimento (Beydeda *et al*, 2005), conforme ilustrado na Figura 3.1. Uma transformação é um refinamento de um modelo, resultando em um artefato mais detalhado. No MDD, sucessivas transformações são efetuadas até a geração do código fonte. Transformações podem ser classificadas segundo seu resultado: quando o resultado da transformação é outro modelo, ela é denominada M2M (*Model to Model*) e quando o artefato gerado é código fonte, trata-se de uma transformação M2C (*Model to Code*).

Figura 3.1: Um esquema de transformações de modelos no MDD.

Outros autores costumam se referir ao conceito de MDD utilizando outras terminologias, como, por exemplo, MDE (*Model-Driven Engineering*) (Schmidt, 2006), MDSD (Model-Driven Software Development) (Stahl *et al*, 2006) ou MD* (Völter, 2008 *apud* Lucrédio, 2009). Esses acrônimos se referem a uma abordagem comum, cujo objetivo é valorizar a importância dos modelos no processo de desenvolvimento como parte integrante do software (Lucrédio, 2009). A maior vantagem de desenvolver software com apoio de MDD é que os desenvolvedores passam a expressar conceitos independentes de linguagem de programação e mais próximos da realidade do domínio (Selic, 2003). Dessa forma, o domínio da aplicação é representado de forma completa, facilitando sua manutenção e evolução.

Dentre as vantagens do MDD, pode-se citar: i) representatividade do conhecimento de domínio; ii) produtividade, pois as tarefas repetitivas podem ser automatizadas; iii) reuso de

conhecimento, por meio da modelagem de alto nível; iv) portabilidade, uma vez que é possível aplicar transformações de modelos para diferentes tecnologias; v) melhoria nos processos de manutenção. As principais desvantagens referem-se à complexidade de utilização, uma vez que são necessários profissionais capacitados na construção de modelos e transformadores.

Para que um engenheiro de software utilize MDD, é necessário um método de construção de modelos, seja por apoio ferramental ou manualmente. Todo modelo construído deve ter um conjunto de atividades e regras para sua definição, que são especificadas em um metamodelo. Ao construir um modelo, o desenvolvedor pode utilizar modelos de propósito geral, como UML (OMG, 2012c) e BPMN (OMG, 2012d), por exemplo, ou criar modelos específicos de domínio, em um processo denominado Modelagem Específica de Domínio (*Domain-Specific Modeling - DSM*, em inglês) (Kelly *et al*, 2008). A modelagem específica de domínio é feita por meio de Linguagens Específicas de Domínio (*Domain-Specific Languages – DSL*, em inglês) (Fowler, 2010), ou mesmo, Linguagens de Modelagem Específicas de Domínio (*Domain-Specific Modeling Languages – DSML*, em inglês) (Kelly *et al*, 2008). A utilização de DSMLs para modelagem permite a especificação exata do que se deseja representar, expressando o domínio de maneira adequada e facilitando a construção de ferramentas de transformação.

Os modelos definidos devem, então, ser transformados em artefatos mais detalhados. A transformação pode ser feita com apoio de: ferramentas de terceiros, como as ferramentas apresentadas na Seção 2.5 ou a própria ferramenta Ruby-MDWA; uma ferramenta desenvolvida pela própria equipe de desenvolvimento do domínio; ou mesmo ser feita manualmente a partir de um conjunto fixo de regras para transformação. Transformações manuais costumam ser realizadas em níveis de abstração altos, quando os modelos estão mais próximos de linguagem natural do que elementos organizados em um diagrama.

3.3 Técnicas de desenvolvimento guiado por modelos

Nesta seção são apresentadas duas técnicas para o desenvolvimento guiado por modelos. Na Subseção 3.3.1 é apresentada a abordagem MDA e na Subseção 3.3.2 é apresentada a abordagem Eclipse EMF/Ecore.

3.3.1 MDA (*Model-Driven Development*)

Model Driven Architecture (MDA) é uma iniciativa proposta pelo *Object Management Group* (OMG, 2012b), para isolar os esforços referentes à especificação do software dos pontos particulares relacionados à sua implementação, aumentando o grau de independência de um sistema em relação à plataforma tecnológica utilizada. Isso é feito por meio de três níveis de abstrações, ilustrados na Figura 3.2:

i) Modelos Independentes de Computação (*Computacional Independent Model* – CIM, em inglês) representam abstrações de software de um ponto de vista que não depende de computação. Um modelo CIM não mostra detalhes do sistema e também é chamado de modelo do domínio (OMG, 2003);

ii) Modelos Independentes de Plataforma (*Platform Independent Model* – PIM, em inglês) tem enfoque na arquitetura do software, modelando os detalhes comuns para a operação em várias plataformas (Beydeda *et al*, 2005);

iii) Modelos Específicos de Plataforma (*Platform Specfic Model* – PSM, em inglês) combinam as especificações de modelos PIM com detalhes de operação e construção nas tecnologias de desenvolvimento. Modelos PSM podem ser transformados diretamente em código fonte (Costa; Gomes; Cagnin, 2007).

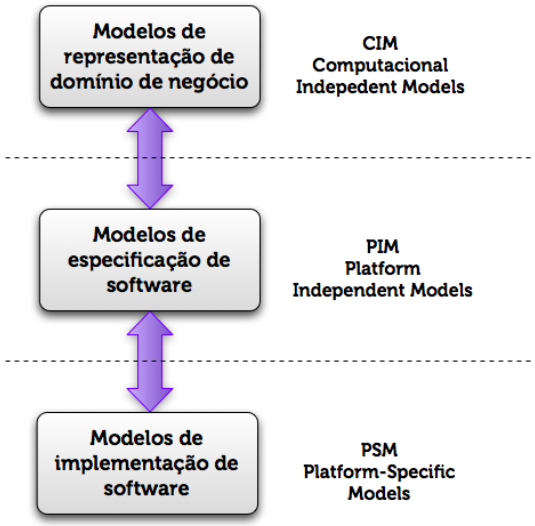


Figura 3.2: Esquema de níveis de abstração MDA.

MDA faz uso de MOF (OMG, 2012a) como o meta-metamodelo de referência. O MOF é um padrão com sintaxe similar à UML, que permite a definição de classes com

atributos e relacionamentos (Lucrédio, 2009). A utilização do padrão MOF garante a construção de modelos padronizados.

3.3.2 Eclipse EMF/Ecore

A IDE Eclipse (Eclipse, 2012) possui um conjunto de iniciativas que apóiam o desenvolvimento guiado por modelos. Essas iniciativas compõem o framework EMF (*Eclipse Modeling Framework*) (Gronback, 2009), que baseia suas implementações em um meta-metamodelo denominado Ecore.

Outro projeto voltado ao desenvolvimento guiado por modelos na IDE Eclipse é o GMF (*Graphical Modeling Framework*), com o qual é possível definir metamodelos e modelos usando um ambiente gráfico. O principal objetivo da iniciativa MDD do Eclipse é servir de oferecer suporte para trabalhos nessa área (Lucrédio, 2009), fornecendo componentes independentes para criação de modelos, bem como, ferramentas de transformação, geração de código e integração com outras tecnologias pela IDE. Inclusive, alguns dos trabalhos relacionados, apresentados na Seção 2.5, são construídos e distribuídos como *plug-ins* para a IDE Eclipse e baseiam seus metamodelos no Ecore.

3.4 Considerações finais

Neste Capítulo foram apresentados os conceitos de MDD, um método de desenvolver software que altera o paradigma utilizado em processos de desenvolvimento tradicionais, incorporando modelos como parte integrante da construção de aplicações.

A especificação de um software ou de um domínio em modelos beneficia o entendimento da equipe, por encorajar a modelagem específica de domínio, de modo a expressar todas as características inerentes ao desenvolvimento. Por esse motivo, resulta em benefícios referentes à manutenção e produtividade. MDD também favorece a reutilização de conhecimento, pois possibilita o reuso de modelos de alto nível.

Por outro lado, a construção de metamodelos, modelos específicos de domínio e ferramentas de transformação não é simples por requerer conhecimento em várias tecnologias. Dessa forma, ao utilizar MDD, o engenheiro de software deve considerar a realização de treinamentos intensivos aos desenvolvedores para garantir que os problemas dos modelos tradicionais de desenvolvimento não ocorram.

4 Uma Abordagem para Desenvolvimento de Software Web Guiado por Modelos

4.1 Considerações Iniciais

O desenvolvimento orientado a modelos tem como objetivo reduzir a distância semântica entre o domínio do problema e o domínio da solução (Fu *et al*, 2011), mudando a forma como o software é construído. Sua principal contribuição é o conceito de desenvolvimento sistemático, por meio de modelos e sucessivos refinamentos. Dessa forma, MDD pode melhorar os índices de produtividade, reuso e expressividade do domínio da aplicação.

Para auxiliar desenvolvedores a atingir esses benefícios, foram propostas várias abordagens que apóiam o desenvolvimento de software utilizando MDD, conforme descrito na seção 2.5. Essas abordagens apresentam métodos extensos para utilização de MDD em processos de desenvolvimento de software Web, e se baseiam em modelos dependentes de ambientes de programação, como por exemplo, Eclipse (Eclipse, 2012), aliados a *templates* de geração de código, buscando automatizar o processo de desenvolvimento utilizando modelos genéricos pré-definidos dependentes de ferramentas de transformação. Como forma de exemplo desse formato de trabalho, pode-se destacar: Castrejón *et al* (2011), Bernardi *et al* (2011), Kroiss *et al* (2009) e WebRatio (2012). Esse formato de abordagem é conflitante com a necessidade de construção de modelos específicos por domínio ou por aplicação, conforme a proposta do MDD. Segundo Fernandes e Melnikoff (2012), a utilização de MDD é complexa e são necessárias novas pesquisas para simplificar sua utilização. O ideal é que novas abordagens de desenvolver software possam ser combinadas com outras formas de reuso.

Este capítulo descreve uma abordagem de desenvolvimento de software Web orientado a modelos, denominada MDWA. É composta por um conjunto de etapas e modelos, bem como, um conjunto de diretivas para transformação entre os modelos.

Além desta introdução, este capítulo é dividido em sete seções. Na Seção 4.2 são descritos os conceitos e objetivos da abordagem MDWA e nas seções seguintes são apresentadas as etapas e artefatos presentes na abordagem. Na Seção 4.3 é apresentada a etapa 1, Mapeamento de Requisitos, e na Seção 4.4 é exibida a etapa 2, Identificação de

Personagens. Na Seção 4.5 é exibida etapa 3, Identificação de Entidades, e na Seção 4.6 é apresentada a etapa 4, Identificação de Processos. Na Seção 4.7 é tratada a etapa 5, Implantação da Infra-estrutura, e na Seção 4.8 é abordada a etapa 6, Detalhamento da Lógica de Negócio. Na Seção 4.9 é exibida a etapa 7, Transformação para Código Fonte, e na Seção 4.10 são apresentadas as considerações finais.

4.2 A Abordagem MDWA

A abordagem MDWA (***M**odel-**D**riven **W**eb **A**pplications*) apresenta um método de desenvolver aplicações Web com apoio de MDD. Com isso, pretende-se que a construção de software guiada por modelos seja menos propensa a falhas e encoraje a reutilização, como forma de melhorar a qualidade. A abordagem é composta por sete etapas, com um conjunto de atividades para a realização de cada uma, que são similares às práticas tradicionais de desenvolvimento de software, como por exemplo, elicitação de requisitos e implementação da arquitetura do software.

A representação do fluxo de execução das etapas é feita por meio de diagramas de atividades UML. Cada etapa MDWA é descrita por uma atividade UML, representada por caixas retangulares de borda arredondada, enquanto que os artefatos são representados por objetos, retângulos. Da mesma forma que o fluxo geral da abordagem MDWA, o detalhamento de cada etapa também é feito usando esses diagramas. Na Figura 4.1 é apresentado o diagrama de atividades UML para ilustrar as etapas e artefatos da abordagem MDWA.

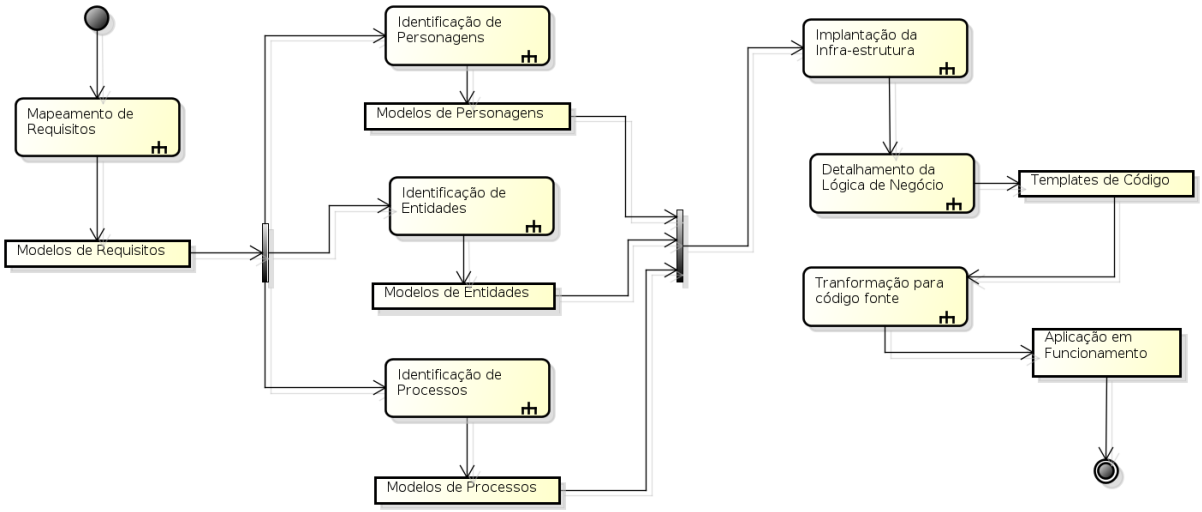


Figura 4.1: Diagrama de Atividades UML que representa as etapas da abordagem MDWA.

A construção de software com apoio da abordagem MDWA tem início com a etapa de Mapeamento de Requisitos, quando os projetistas devem identificar os requisitos funcionais da aplicação, decidir quais as formas de reuso que serão utilizadas para construção do software e separar os requisitos não-funcionais. Cada requisito coletado é mapeado em um modelo de requisito.

Todos os modelos propostos pela abordagem MDWA devem ser definidos pelos desenvolvedores, que por sua vez, precisam especificar as particularidades do domínio alvo. A ferramenta Ruby-MDWA, descrita no capítulo 5, apresenta um metamodelo genérico, e sua representação em DSML textual interna, para cada modelo proposto pela MDWA. Essa ferramenta tem por objetivo servir de base aos desenvolvedores que desejam construir modelos e transformadores específicos de domínio, alinhados com o conceito proposto pelo MDD.

Após a construção dos modelos de requisitos, os desenvolvedores podem realizar três etapas de forma concorrente:

- Identificação de Personagens: que representa a seleção dos usuários que utilizarão a aplicação, gerando modelos de personagens;
- Identificação de Processos: que consiste em identificar os processos que cada usuário pode desempenhar, gerando modelos de processos;
- Identificação de Entidades: que aponta os comportamentos das entidades da aplicação, gerando modelos de entidades.

A estrutura lógica do software composta pelos modelos de personagens, entidades e processos está definida ao final dessas etapas. A interpretação desses modelos é usada para gerar protótipos da aplicação.

A definição da arquitetura da aplicação ocorre na etapa de Implantação da Infra-estrutura. Neste momento, os desenvolvedores devem tomar decisões referentes à arquitetura, ao ambiente de testes, ao modelo de leiaute, à autenticação de usuários, à segurança e às demais particularidades envolvidas no desenvolvimento e implantação do software. Como resultado dessa etapa tem-se a geração de um ambiente que apóia a transformação dos modelos construídos previamente e a implementação das definições de negócio da aplicação que é realizada na etapa seguinte.

Na etapa de Detalhamento da Lógica de Negócio é feita a implementação efetiva das particularidades de negócio da aplicação com base em *templates* de código. Por meio dos *templates* são descritos meta-código, trechos de código fonte que assumem decisões de acordo com as informações disponíveis nos modelos de requisitos, de personagens, de processos e de entidades construídos anteriormente.

Por fim, os desenvolvedores devem realizar a etapa de Transformação para Código Fonte. A equipe de desenvolvimento deve construir ou utilizar um compilador de *templates* para a geração do código fonte da aplicação. A construção do compilador fornece maior conhecimento sobre a semântica da aplicação, dos modelos específicos e da tecnologia de destino. O artefato resultante dessa atividade é a aplicação em funcionamento.

O fluxo de atividades definido anteriormente descreve um processo de desenvolvimento de software independente de plataforma e de tecnologia. As decisões referentes à definição dos modelos propostos, ferramentas, compiladores e tecnologias são de responsabilidade dos desenvolvedores. Desse modo, a abordagem MDWA pode ser combinada com outras formas de reuso, como por exemplo, linguagens de padrões e frameworks, e com outros processos de desenvolvimento de software, como por exemplo, os métodos ágeis ou RUP (RUP, 2012).

As especificidades de cada uma das etapas e de seus artefatos são apresentadas nas seções subseqüentes.

4.3 Etapa 1: Mapeamento de Requisitos

Esta etapa é composta por sete atividades e a geração de modelos de requisitos ao final, como ilustrado na Figura 4.2, por meio de um diagrama de atividades UML.

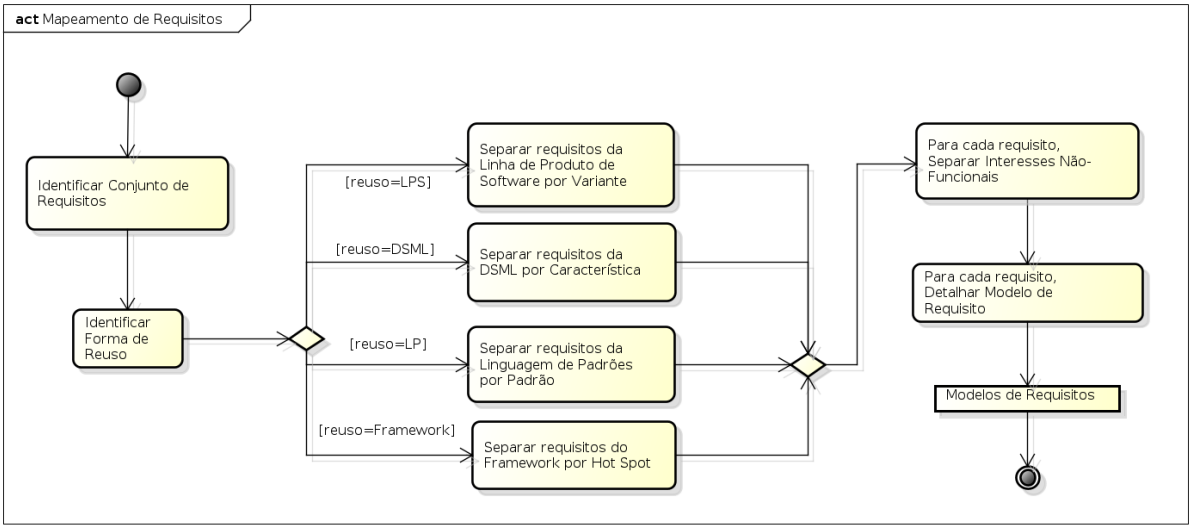


Figura 4.2: Diagrama de Atividades UML que representa a etapa de Mapeamento de Requisitos.

Identificar o Conjunto de Requisitos é a primeira atividade da etapa de Mapeamento de Requisitos, sendo que o documento de requisitos elaborado pode ser implementado total ou parcialmente. Assim, a equipe de software deve identificar o conjunto de requisitos necessário para o ciclo de implementação considerado, de acordo com o planejamento da entrega.

Com base no conjunto de requisitos selecionados, os desenvolvedores devem identificar as formas de reuso que serão utilizadas na construção do projeto e separar os requisitos de acordo com a forma de reuso identificada. O desenvolvedor pode utilizar diferentes mecanismos de reuso para a separação de requisitos. Por exemplo, se linhas de produtos de software forem utilizadas, a instanciação de aplicações inicia-se com a identificação das variantes em relação ao núcleo da linha (Clements; Northrop, 2001). Portanto, pode-se considerar um requisito como uma variante da linha. Se o desenvolvimento de aplicações for com o apoio de linguagens de padrões, deve-se considerar a implementação de cada padrão (Coplien, 2004). Dessa forma, um requisito pode ser um padrão ou um relacionamento entre padrões. Se frameworks forem utilizados como mecanismo de reuso, a instanciação de sistemas pode ser conduzida pela identificação dos *hot spots*, dos quais as especificidades de negócio podem ser estendidas do conjunto de funções provido pelo framework (Brugali; Sycara, 2000). Desse modo, um requisito pode ser visto como um *hot spot* explorado pela aplicação.

Outro mecanismo de reuso que pode ser considerado é o uso de DSLs e DSMLs (Kelly *et al*, 2010). Desse modo, a construção de software com apoio é coordenada pelas características inerentes ao domínio (Luoma *et al*, 2004), obtidas pelo processo de engenharia de domínio. Por esse motivo, um requisito pode ser separado como uma característica do domínio.

Na sequência da separação de requisitos por mecanismo de reuso, os desenvolvedores devem distinguir a descrição funcional da não-funcional dos requisitos. Essa prática separa o modo e a posterior implementação do requisito. Requisitos funcionais são abordados nas fases estruturais, como Identificação de Personagens, Entidades e Processos e no Detalhamento da Lógica de Negócio. Requisitos não-funcionais são tratados na Implantação da Infra-Estrutura.

Ao final, cada requisito criado deve ter sua especificação representada em um modelo de requisito. O formato do modelo de requisito pode ser definido pela equipe de desenvolvimento. Na seção 5.3.1 é apresentado o metamodelo desenvolvido para a geração de modelos de requisitos pela ferramenta Ruby-MDWA e pode ser empregado como base para a construção de modelos de requisitos específicos de domínio.

Um dos objetivos das técnicas de levantamento de requisitos é especificar as necessidades do cliente de forma adequada (Mishra *et al*, 2008). Processos de desenvolvimento de software bem estruturados devem ser apoiados em técnicas de elicitação de requisitos, pois se os requisitos não estiverem especificados de modo eficaz, podem ocorrer falhas na construção de software (Goguen; Linde, 1993).

4.4 Etapa 2: Identificação de Personagens

A seleção dos executores de ações em um software é uma etapa presente em todos os processos de desenvolvimento de software. Desde o início do ciclo de vida de software, a identificação dos possíveis usuários e suas características serve como base para o projeto de interface de usuário, controle de permissões, autenticação, entre outros. A modelagem de usuários é abordada pelos diagramas de casos de uso e sequência da UML (OMG, 2012c) e em diagramas BPMN (OMG, 2012d).

A etapa de Identificação de Personagens é uma das três atividades que podem ser executadas de forma concorrente após o termino da etapa de Mapeamento de Requisitos. Para a abordagem MDWA, um personagem é um tipo de usuário e o conjunto de ações que pode executar no contexto do software em execução. Personagens devem conter uma descrição

textual de suas ações e a especificação dos papéis que pode exercer no software em funcionamento. Na Figura 4.3 é apresentado o diagrama de atividades UML que representa as ações que devem ser executadas na etapa de Identificação de Personagens.

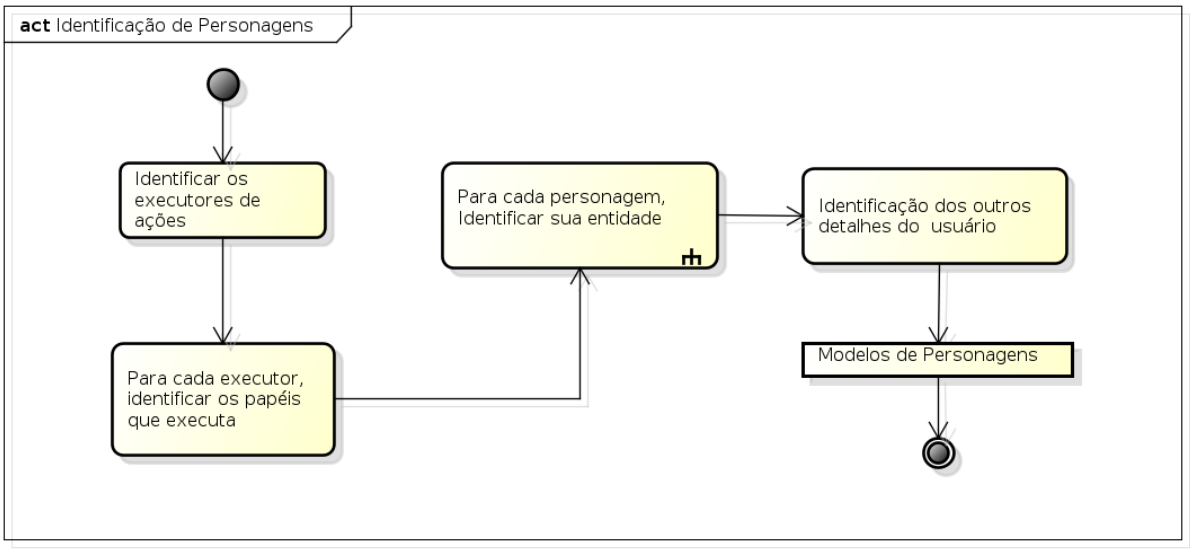


Figura 4.3: Diagrama de Atividades UML que representa a etapa de Identificação de Personagens.

A etapa se inicia com a atividade Identificar Executores de Ações no sistema em funcionamento. Um executor é uma pessoa física que realiza um conjunto de ações no software. Para cada executor identificado, devem ser verificados todos os papéis que ele pode realizar no sistema. Os executores devem ser agrupados de acordo com os papéis, que desempenham no sistema.

A partir da seleção dos executores de ações e seus papéis são criados os personagens. Posterior a identificação de cada personagem, deve-se identificar as suas particularidades, o que é realizado nas etapas de Identificação de Entidades e Detalhamento da Lógica de Negócio. Os demais detalhes inerentes aos personagens devem ser levantados e representados nos modelos de personagens.

4.5 Etapa 3: Identificação de Entidades

Uma entidade MDWA é similar à abstração de classe do diagrama de classes UML, contendo também as demais informações necessárias para sua implementação. Por exemplo, quando se tem um cadastro de produtos é desejável que se saiba quais são suas interfaces, estilos de visualização, restrições, entre outros. Na Figura 4.4 é apresentado o diagrama de atividades UML que apresenta as atividades desta etapa.

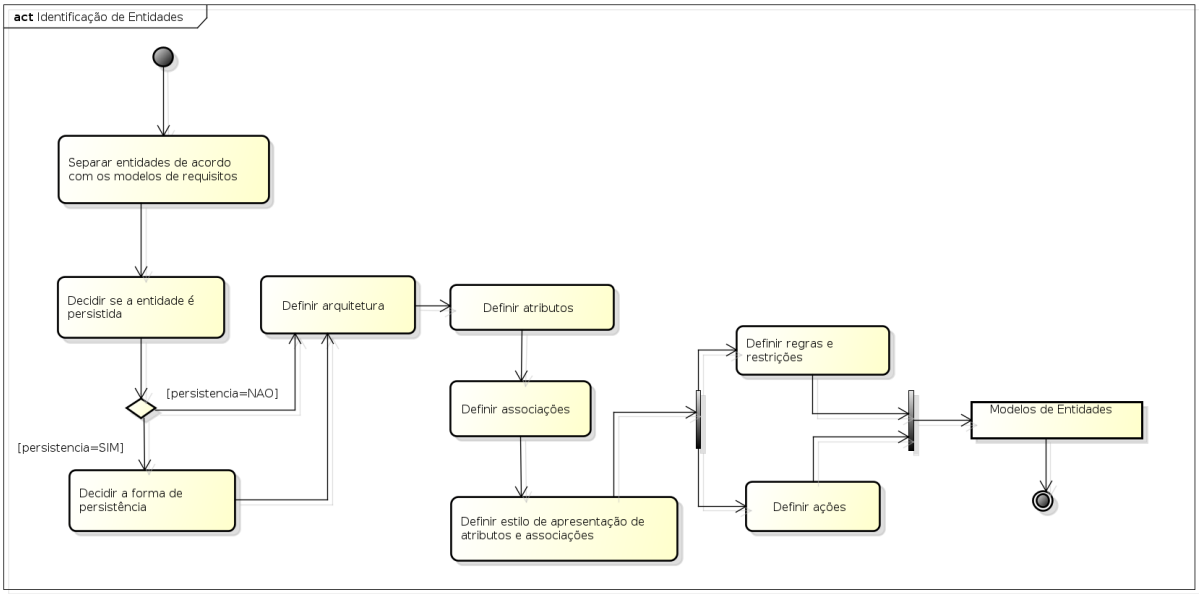


Figura 4.4: Diagrama de Atividades UML que representa a etapa de Identificação de Entidades.

A etapa se inicia com a separação das entidades de acordo com os modelos de requisitos, definidos na etapa de Mapeamento de Requisitos. Para cada entidade, a equipe de desenvolvimento deve decidir se a entidade é persistida, ou seja, se tem seus dados armazenados. Caso a entidade seja persistida, os desenvolvedores devem decidir de qual forma a persistência será efetuada, como por exemplo, utilizar banco de dados com envio de informações por AJAX (W3C, 2012) ou armazenamento em *webservices*.

A ação posterior é a definição da arquitetura que define como organizar os arquivos de implementação da entidade. Uma arquitetura comumente utilizada em aplicações Web é a arquitetura MVC (Gamma *et al*, 1995).

Os passos seguintes são Definir Atributos e Definir Associações, respectivamente, e devem selecionar os atributos e relacionamentos entre entidades, de forma similar aos atributos e relacionamentos presentes em um diagrama de classes UML.

Em seguida, os desenvolvedores devem definir os estilos de apresentação de cada atributo e de cada associação. Os estilos são utilizados nos *templates* de código fonte, que são os artefatos gerados pela etapa de Detalhamento da Lógica de Negócio, conforme exibido na Seção 4.8, e definem a forma que os atributos e associações são visualizados na interface de usuário. Posteriormente, a equipe de desenvolvedores pode realizar duas atividades de forma concorrente: Definir Restrições e Regras ou Definir Ações.

As regras e restrições de uma entidade são definições formais sobre o modo de funcionamento da entidade dentro dos contextos dos requisitos em que está presente. Por exemplo, um produto tem restrições quanto ao seu preço, que não pode ser menor que zero. Com isso, é possível que se efetue a geração de casos de testes automatizados ou de mecanismos de validação de formulários.

As ações MDWA são operações que aplicações executam quando requeridas via interface, seja gráfica ou outra. Em outras palavras, são todas as requisições que aplicações Web podem responder. A atividade de identificação de ações fornece todas as interfaces de requisições que uma entidade executa, de acordo com o formato dos dados de envio e de resposta. Por fim, para cada entidade, todos os dados coletados devem ser organizados em um modelo de entidade.

4.6 Etapa 4: Identificação de Processos

A etapa de Identificação de Processos pode ser realizada de forma concorrente com as etapas Identificação de Personagens e Identificação de Entidades, após o término da etapa de Mapeamento de Requisitos.

Um processo MDWA é uma seqüência de atividades que um ou mais personagens executam com o objetivo de prover um artefato ou serviço. Esse conceito de processo é baseado no conceito de processo proposto pela linguagem de modelagem BPMN (OMG, 2012d) e nos comportamentos modelados por diagramas de seqüência e atividades UML (OMG, 2012c). Na Figura 4.5 são ilustradas as atividades desta etapa.

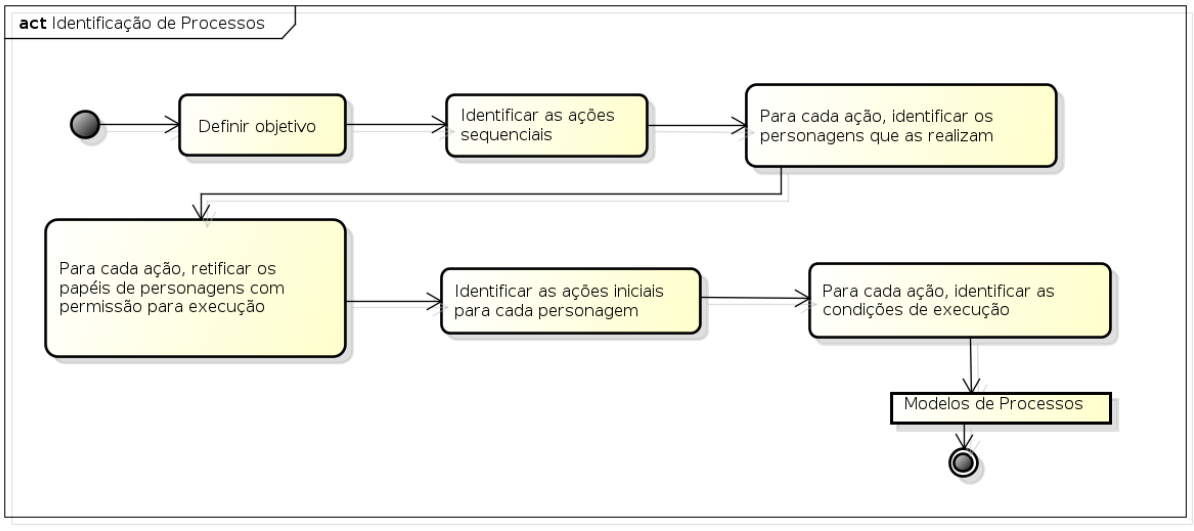


Figura 4.5: Diagrama de Atividades UML que representa a etapa de Identificação de Processos.

A atividade se inicia com a definição de um objetivo a ser atingido, podendo ser a realização de um serviço ou a produção de um artefato. Caso o objetivo possa ser alcançado diretamente com uma única ação, claramente, não se trata de um processo. Após a definição do objetivo, deve-se identificar a seqüência de ações necessárias para que o objetivo seja atingido.

Em seguida, a equipe de desenvolvimento deve apontar os personagens que podem executar cada ação detectada. Neste ponto do andamento da atividade, tem-se uma seqüência de ações, bem como, uma correspondência entre cada ação e seus possíveis executores. A seguir, é necessário retificar que as ações realizadas por cada personagem estejam presentes na lista de papéis que ele possui. Dessa forma, pode ser necessário efetuar alterações nos modelos de personagens.

A seguir, é necessário constatar em qual ação cada personagem inicia o processo. Na seqüência, os desenvolvedores devem formalizar as condições para transição entre as ações do processo. Finalmente, para cada processo identificado, as informações coletadas durante as atividades previamente apresentadas devem ser descritas em um modelo de processo.

4.7 Etapa 5: Implantação da Infra-estrutura

Após a conclusão das etapas que criam a estruturais, Identificação de Personagens, Entidades e Processos, tem-se um conjunto de modelos que representam a estrutura lógica da aplicação. Neste momento do desenvolvimento, é necessária a criação de um ambiente que forneça a arquitetura e trate dos requisitos não-funcionais da aplicação.

Esse ambiente deve prover a infra-estrutura para a fixação dos alicerces da estrutura do software, de forma similar à construção civil. Sua levar em conta várias necessidades do desenvolvimento de software Web em geral, como por exemplo, arquitetura, testes automatizados, segurança, autenticação de usuários, entre outros. Grande parte das necessidades de infra-estrutura pode ser suprida com o apoio de frameworks do tipo MIF (*Middleware Integration Framework*). Frameworks MIF fornecem arquitetura de projeto (Clements; Northrop, 2001), controlam o fluxo da aplicação (Larman, 2004) e, quando voltados para Web, oferecem implementações de sucesso para as necessidades desse domínio. Na Figura 4.6 é apresentado o diagrama de atividades UML que define o fluxo de etapas para implantação da infra-estrutura

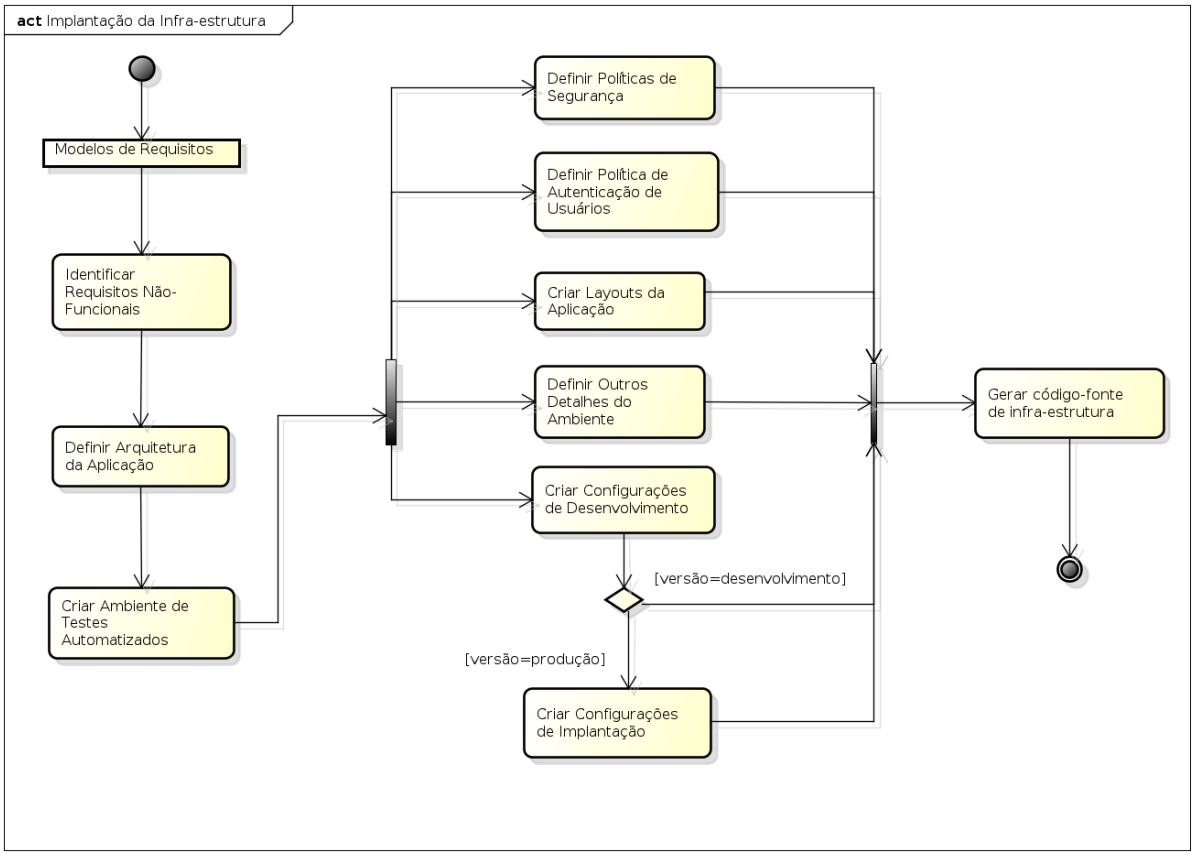


Figura 4.6: Diagrama de Atividades UML que representa a etapa de Implantação da Infra-estrutura.

Esta etapa se inicia com a identificação dos requisitos não-funcionais a partir dos modelos de requisitos criados anteriormente. Requisitos não-funcionais, como, por exemplo, segurança, privacidade, performance e usabilidade, (Urrego-Giraldo, 2004) são importantes para garantir o sucesso do software, especialmente quando baseados na Web (Jaramillo, 2011).

Na seqüência, os desenvolvedores devem decidir a arquitetura da aplicação. Conforme apresentado na Seção 4.5, a arquitetura mais comum em aplicações Web segue o padrão arquitetural MVC (*Model-View-Controller*) (Gamma *et al*,1995).

Após a definição da arquitetura, deve-se criar o ambiente de testes automatizados. A incorporação de testes automatizados agrega qualidade em software de todos os tamanhos (Kerry; Delgado, 2009), pois aumentam a eficiência ao verificar funcionalidades e facilitam o processo de manutenção.

Posteriormente, o desenvolvedor pode realizar um conjunto de ações em paralelo. Essas ações consistem em projetar e codificar os requisitos não-funcionais, como segurança,

autenticação de usuários, leiaute de interface, criar as configurações dos ambientes de desenvolvimento e produção, entre outros. A partir desse momento, os engenheiros de software devem gerar o código que constrói a infra-estrutura, resultando em um protótipo da aplicação em funcionamento, sem sua especificação de negócio.

4.8 Etapa 6: Detalhamento da Lógica de Negócio

Templates são arquivos descritos em DSLs textuais que representam meta-código, pois são empregados na geração código fonte em uma linguagem de programação específica. Possuem comandos especiais, descritos pela DSL utilizada, que são sub-rotinas que acessam as informações de outros modelos. O acesso às informações de modelos permite ao desenvolvedor decidir como será a implementação de uma dada função do software. As atividades desta etapa são apresentadas na Figura 4.7.

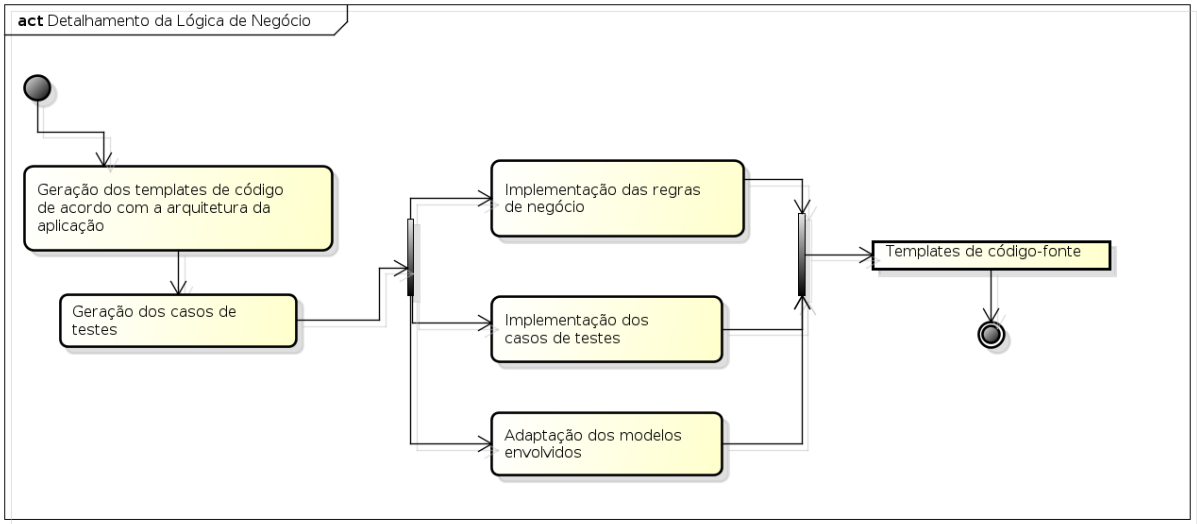


Figura 4.7: Diagrama de Atividades UML que representa a atividade de Detalhamento da Lógica de Negócio.

A etapa se inicia com a geração dos *templates* de código fonte de acordo com a arquitetura definida para as entidades e para a infra-estrutura. Posteriormente, as restrições descritas nos modelos de entidades representam os casos de testes que devem ser realizados para verificar o funcionamento da lógica das entidades.

Posteriormente, os desenvolvedores podem efetuar três atividades de forma concorrente: a implementação das regras de negócio da aplicação, a codificação dos casos de testes ou a adaptação dos modelos envolvidos no ciclo de detalhamento da lógica de negócio. A atividade de adaptação dos modelos garante que a consistência entre modelos, código e

testes seja mantida durante esta etapa, uma vez que é comum que a especificação de um requisito seja modificada durante sua implementação. A escolha da ordem, ou mesmo a alternação entre as etapas fica a cargo dos desenvolvedores.

Portanto, é possível que a equipe permaneça utilizando as práticas as quais já está habituada para a codificação das regras do negócio. Dessa forma, desenvolvedores habituados com técnicas guiadas por testes, como, por exemplo, TDD (*Test-driven Development)* (Janzen; Saiedian, 2008), podem continuar com as práticas de *test-first* e *red-green-refactor* para realizar a implementação da lógica de negócio.

Ao final, os *templates* de código fonte, gerados no início da etapa, agora contém a implementação das funções identificadas nos requisitos.

4.9 Etapa 7: Transformação para Código Fonte

A última etapa da abordagem MDWA é responsável pela compilação dos *templates* em código fonte, resultando na aplicação em funcionamento. O processo de compilação deve interpretar os *templates* e substituir as referências aos modelos, previamente definidos, pelos dados contidos nos modelos.

A compilação deve ser feita por uma ferramenta geradora de código a partir de *templates*, como por exemplo, Acceleo (Acceleo, 2012), JET (Java Emitter Templates, 2012), WebRatio (WebRatio, 2012) ou o mecanismo ERB da linguagem Ruby (Ruby, 2012). A transformação não-automatizada de *templates* é lenta e prejudica a eficiência do desenvolvimento. Além da geração de código, o compilador deve executar um conjunto de atividades para garantir o funcionamento completo da aplicação. Na Figura 4.8 é apresentado o diagrama de atividades UML que representa as atividades da etapa de Transformação para Código Fonte.

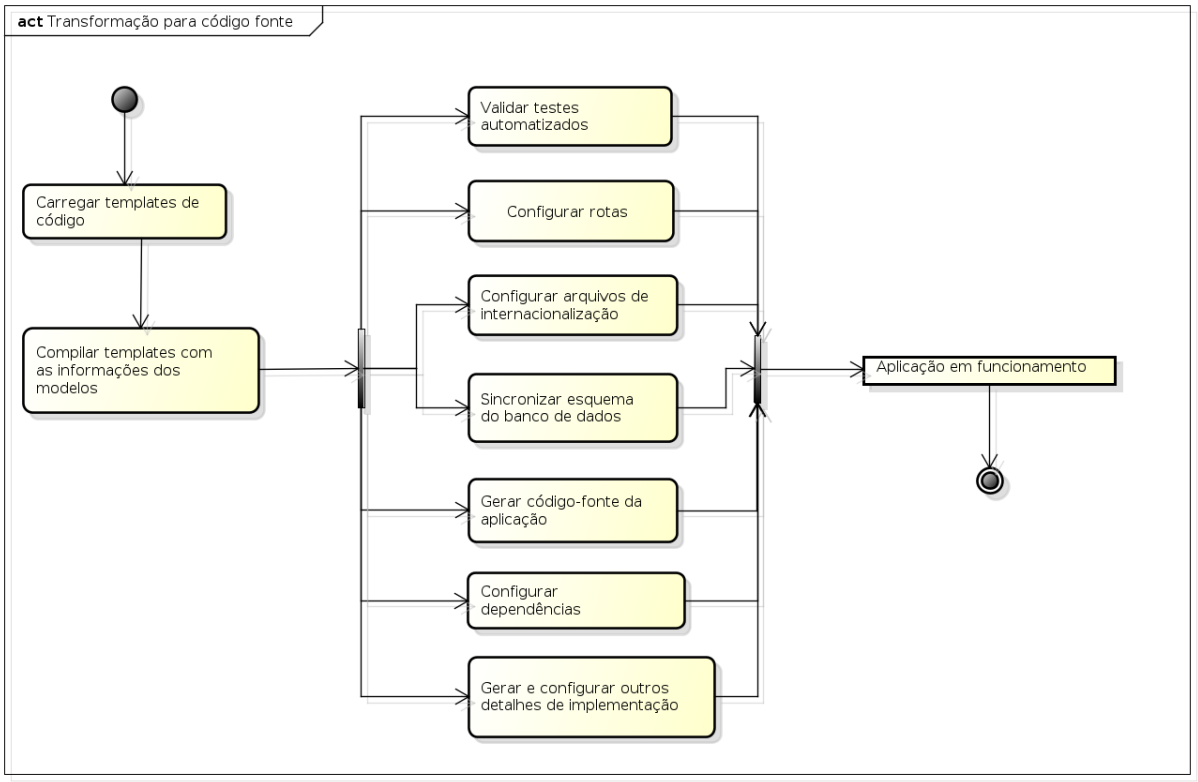


Figura 4.8: Diagrama de Atividades UML que representa a etapa de Transformação para Código-Fonte.

A etapa se inicia com o carregamento dos *templates* de código e com o princípio da compilação. Na seqüência, o compilador deve considerar uma lista de atividades concorrentes que automatizam as tarefas repetitivas de desenvolvimento. Dentre essas atividades, algumas devem ser realizadas para grande parte das aplicações Web, como:

- Executar e validar os testes automatizados, que garantem que o código gerado esteja em funcionamento;
- Configurar dependências de componentes, frameworks, bibliotecas, entre outros;
- Mapear as rotas URL para as ações designadas;
- Geração dos arquivos de internacionalização;
- Sincronizar o esquema de banco de dados, prevendo alteração de tabelas e colunas, de acordo com as informações dos modelos de entidades;

Finalmente, o compilador gera o código fonte que compõe a aplicação final em funcionamento.

De acordo com os conceitos de desenvolvimento guiado por modelos, manutenções devem ser indicadas nos modelos de alto nível e, posteriormente, transformadas aos modelos de baixo nível até a geração de código. Nesse contexto, o andamento do desenvolvimento com apoio de MDWA deve ser especificado inicialmente na etapa de Mapeamento de Requisitos e seguir o fluxo de etapas até a transformação para código-fonte, de modo a manter a consistência entre todos os artefatos do software.

4.10 Considerações Finais

Uma das atividades deste projeto de mestrado consistia da construção de uma ferramenta de apoio a construção de aplicações baseadas no framework Ruby on Rails utilizando linguagens de modelagem específicas de domínio. Entretanto, durante a realização dessa atividade, constatou-se que a construção de uma ferramenta com essas características deve ser baseada em um processo que guie o andamento do desenvolvimento. Também foi observado que esse processo pode ser descrito de forma genérica, sem dependência de tecnologias. A partir desse trabalho, foi extraída a abordagem MDWA.

A abordagem MDWA auxilia o engenheiro de software a desenvolver aplicações Web guiadas por modelos. A definição do modo de execução das ações é de responsabilidade do engenheiro de aplicação, que pode aplicar os métodos e práticas propostos em modelos de processos ou adotados por sua equipe. Além disso, a especificação dos modelos também é de responsabilidade do engenheiro de software, que pode planejar a construção de modelos adequados ao domínio da aplicação.

Portanto, o projeto de aplicações Web com auxílio da abordagem é fortemente guiado pelas necessidades do domínio e estimula a utilização de boas práticas e reuso de software, de forma independente de tecnologias.

5 A Ferramenta Ruby-MDWA

5.1 Considerações iniciais

Com o advento do desenvolvimento de software guiado por modelos, é possível construir aplicações de forma mais rápida e expressar de forma mais adequada às particularidades de cada domínio. Esses benefícios são atingidos por meio da criação de modelos de alto nível e sucessivos refinamentos até a geração de código do software em funcionamento.

Porém, mesmo com o suporte de processos e abordagens guiadas por modelos, os desenvolvedores inexperientes podem encontrar dificuldade em construir modelos específicos de domínio e transformadores M2M e M2C, principalmente, nos primeiros contatos com MDD.

Essa dificuldade pode ser atenuada com a utilização de ferramentas que fornecem modelos básicos e transformadores entre modelos e código fonte. Dessa forma, o desenvolvedor não necessita criar DSMLs e transformadores automáticos do início, podendo dar enfoque na representação do domínio e implementação da aplicação.

Com esse propósito, a ferramenta Ruby-MDWA oferece suporte ao fluxo de atividades e artefatos propostos pela abordagem MDWA, por meio de modelos genéricos, baseados em DSML textuais, e transformadores de modelos e código fonte.

Além desta introdução, as demais seções deste capítulo mostram de forma detalhada o funcionamento e a utilização da ferramenta proposta e estão organizadas da seguinte forma: na Seção 5.2 é apresentada a arquitetura da ferramenta Ruby-MDWA, seus modelos e transformadores; na Seção 5.3 é exibido o metamodelo de requisitos e na Seção 5.4 é representado o metamodelo de personagens; na Seção 5.5 é apresentado o metamodelo de entidades e na Seção 5.6 é exibido o metamodelo de processos; na Seção 5.7 é abordada a implantação da infra-estrutura e na Seção 5.8 são tratados os *templates* de código; na Seção 5.9 é discutida a transformação de *templates* para código fonte; na Seção 5.10 são apresentadas as considerações finais.

5.2 Arquitetura da Ferramenta Ruby-MDWA

A ferramenta Ruby-MDWA fornece apoio para o desenvolvimento de aplicações Web guiado por modelos, baseado na abordagem MDWA, tendo sua implementação na linguagem Ruby com suporte do framework Ruby on Rails. Sua implementação é encapsulada em um componente *Rubygem* e seu código fonte é *open-source*. Sua construção e seus métodos originaram a abordagem MDWA, que pode ser generalizada para qualquer tecnologia de desenvolvimento Web. Para sua utilização, o desenvolver deve possuir conhecimentos intermediários em Ruby e Ruby on Rails, além de conhecimento básico sobre MDD e a abordagem MDWA.

A ferramenta provê um conjunto de modelos e transformadores M2M e M2C. A representação dos modelos é feita com DSMLs textuais, construídas utilizando conceitos de módulos e blocos da linguagem Ruby, caracterizando uma DSL interna (Fowler, 2010). O transformadores foram implementados com suporte dos mecanismos de geração de código do framework Ruby on Rails e podem ser acessados por comandos de terminal, independentes de sistema operacional.

A utilização de DSMLs textuais internas baseadas em código Ruby simplifica a definição de linguagens de modelagem, e principalmente, as transformações entre modelos. Nesse formato, as estruturas de dados contendo as informações dos modelos são carregadas automaticamente, sem necessidade de tratamento ou interpretação de dados, e inclusive, diminui o tempo de processamento das transformações. Além disso, é possível que as informações dos modelos textuais sejam transportadas para modelos gráficos, que são mais fáceis de interpretar. Por outro lado, a utilização de DSLs internas para especificação de modelos pode dificultar a utilização por não-programadores e restringe a expressividade desses modelos pela dependência da sintaxe da linguagem Ruby.

Ao desenvolver aplicações Web com o apoio da ferramenta Ruby-MDWA, o engenheiro de software deve seguir as etapas da abordagem MDWA e utilizar a ferramenta para gerar os modelos necessários e efetuar suas transformações. Na Figura 5.1 é ilustrado o esquema de transformação entre os modelos da ferramenta Ruby-MDWA, bem como os nomes dos comandos de transformação. Embora a abordagem MDWA e a ferramenta de apoio não adotem formalmente a arquitetura MDA (OMG, 2012b), os modelos propostos podem ser divididos de acordo com as camadas CIM, PIM e PSM.

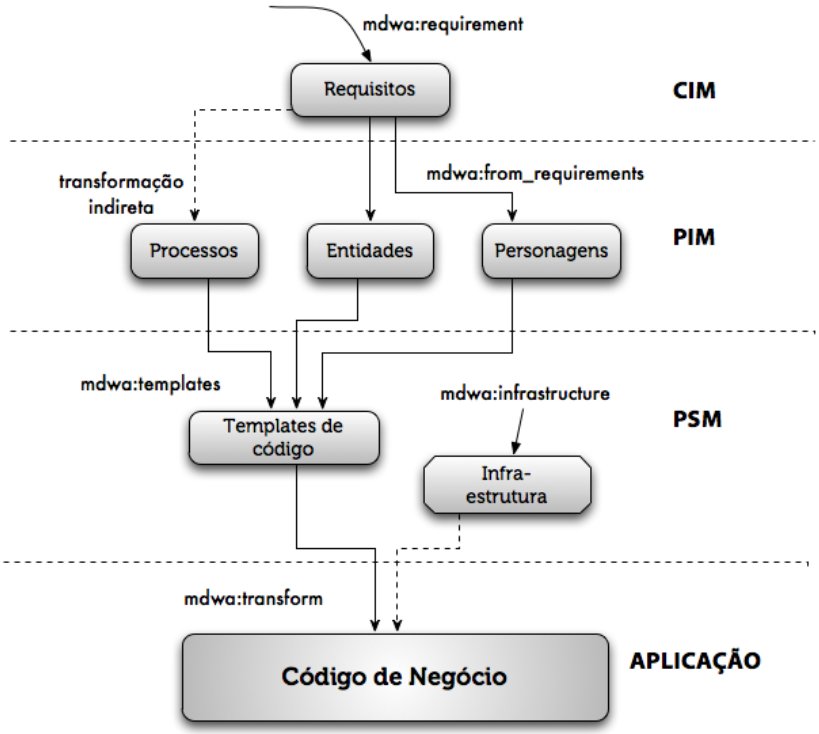


Figura 5.1: Esquema de transformação entre modelos da ferramenta Ruby-MDWA.

O primeiro artefato a ser abordado é o modelo de requisitos, que armazena a descrição funcional e não-funcional do requisito, além de uma lista de entidades e personagens que podem ser identificados com as descrições do modelo.

A transformação de requisitos para processos é indireta, pois um processo pode ser referente a vários requisitos e descrever diversas funções. Por outro lado, a transformação de requisitos para entidades e personagens é feita diretamente via linha de comando.

Os modelos de processos, entidades e personagens caracterizam os modelos estruturais da aplicação e o detalhamento da lógica de negócio é feita em *templates* de código organizados pela arquitetura do software. A transformação dos modelos estruturais para *templates* é feita automaticamente por comando de terminal.

De forma paralela a essa transformação, o engenheiro de software deve gerar o ambiente de infra-estrutura. Como as decisões mais importantes da etapa de Implantação da Infra-Estrutura, definição da arquitetura e segurança, são providas pelo framework Ruby on Rails, a geração de código dos outros requisitos não-funcionais não precisa ser executada antes da criação dos *templates*.

Finalmente, a transformação dos *templates* em código fonte e geração das outras configurações é realizada com o comando *mdwa:transform*, resultando na aplicação em funcionamento.

5.3 Metamodelo de Requisito

Na etapa de mapeamento de requisitos MDWA, o projetista de software seleciona os requisitos compreendidos na iteração ou manutenção em questão. Então, as informações levantadas de cada requisito são representadas em modelos de requisitos.

Um modelo de requisito possui título, descrição da parte funcional e da parte não-funcional, uma lista de entidades e uma lista de personagens envolvidos com sua implementação, conforme é ilustrado na Figura 5.2, enquanto que na Figura 5.3 é apresentado um modelo de requisito MDWA representado na DSML textual Ruby-MDWA.

Com essa modelagem, é possível expressar o que deve ser feito em linguagem natural, analisar o que já pode ser determinado em nível de estrutura da aplicação e identificar quem são os usuários que participam daquele requisito.

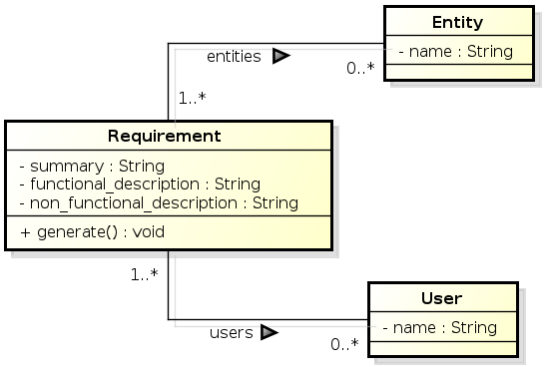


Figura 5.2: Metamodelo de Requisito MDWA.

```
MDWA::DSL.requirements.register do |r|

  r.summary = 'Manage project and tasks'
  r.description = %q{
    The team member must be able to manage projects, tasks and deadlines.
    Projects must be categorized.
    Project attributes are: name, due date and responsible.
    Tasks have title, responsible and due date.
    Deadlines have title, description, responsible and due date.
    Deadlines due dates must appear in calendar view.
  }

  r.non_function_description = %q{
    The deadlines calendar view must reload every 5 seconds.
    Tasks forms should open as modalbox and save as ajax request.
  }

  r.entities  = ["ProjectGroup", "Project", "Task", "Milestone"]
  r.users     = ["Administrator", "TeamMember"]

end
```

Figura 5.3: Modelo de requisitos descrito na DSML Ruby-MDWA.

A ferramenta Ruby-MDWA fornece comandos executados via terminal para criação de modelos e suas transformações. A geração de modelos de requisitos, conforme ilustrado na Figura 5.3, é feita com o comando abaixo, onde o título do requisito é o argumento principal:

```
rails generate mdwa:requirement "<titulo do requisito>"
```

Posteriormente à representação dos dados nos modelos de requisitos, é efetuada uma transformação M2M que deve gerar modelos de entidades e personagens, com o comando:

```
rails generate mdwa:from_requirements
```

A transformação gera modelos de entidades, indicando a relação entre a entidade e o requisito. O mesmo ocorre com os personagens. A ferramenta não cria restrições para a quantidade de requisitos processada antes de efetuar transformações, ou seja, é possível criar um requisito e transformá-lo, individualmente, possibilitando que o engenheiro de software utilize métodos ágeis.

5.4 Metamodelo de Entidades

A identificação dos usuários que irão interagir com um software é uma parte fundamental da etapa de levantamento de requisitos. Com essas informações, desenvolvedores e projetistas podem elaborar um projeto de interface e navegação dedicado a esses usuários. Para a abordagem MDWA, os personagens são os usuários que entram em contato com a aplicação em execução em algum momento. Todo personagem tem papéis a executar no sistema. O controle de permissões por papéis facilita a verificação de autorização dos personagens aos recursos da aplicação. Esse conceito é uma simplificação dos padrões Role Pattern (Baumer *et al*, 1997) e Extension Object (Gamma *apud* Baumer, 1997). Na

Figura 5.4 é apresentado o metamodelo de personagens que utiliza esse conceito e na Figura 5.5 são apresentados modelos de personagem *Administrator* e *TeamMember*, representados com a DSML Ruby-MDWA.

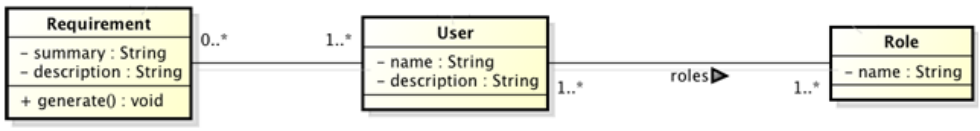


Figura 5.4: Metamodelo de Personagens MDWA.

Por exemplo, considere uma funcionária cujo cargo é secretária. Em um software de atendimento, a secretária tem as funções "Iniciar o atendimento de clientes" e "Agendar reuniões". Na modelagem proposta, o personagem é "Secretária" e seus papéis são "Iniciar atendimento" e "Agendar reuniões". O diretor da empresa não atende clientes, mas pode agendar suas próprias reuniões, então o usuário "Diretor" também possui o papel "Agendar reuniões", com a restrição de serem suas próprias reuniões.

A transformação dos modelos de requisitos para modelos de personagens também gera um modelo de entidade para cada personagem, pois possuem informações inerentes à sua implementação.

```
MDWA::DSL.users.register 'Administrator' do |u|
  u.description = 'Administrator user. Have super admin power.'
  u.roles = ['Director', 'CompanyOwner', 'SoftwareEngineer', 'Programmer']
end


MDWA::DSL.users.register 'TeamMember' do |u|
  u.description = 'Member of the programmers team.'
  u.roles = ['Programmer', 'InterfaceDesigner']
end
```

Figura 5.5: Modelos de Personagens gerados pela ferramenta Ruby-MDWA.

5.5 Metamodelo de Entidades

Entidades são abstrações do mundo real, conforme o conceito proposto no desenvolvimento orientado a objetos. Uma entidade MDWA é similar a abstração de classe do diagrama de classes UML (OMG, 2012c), contendo também as demais informações necessárias para sua implementação. Na Figura 5.6 é apresentado o metamodelo de entidades MDWA.

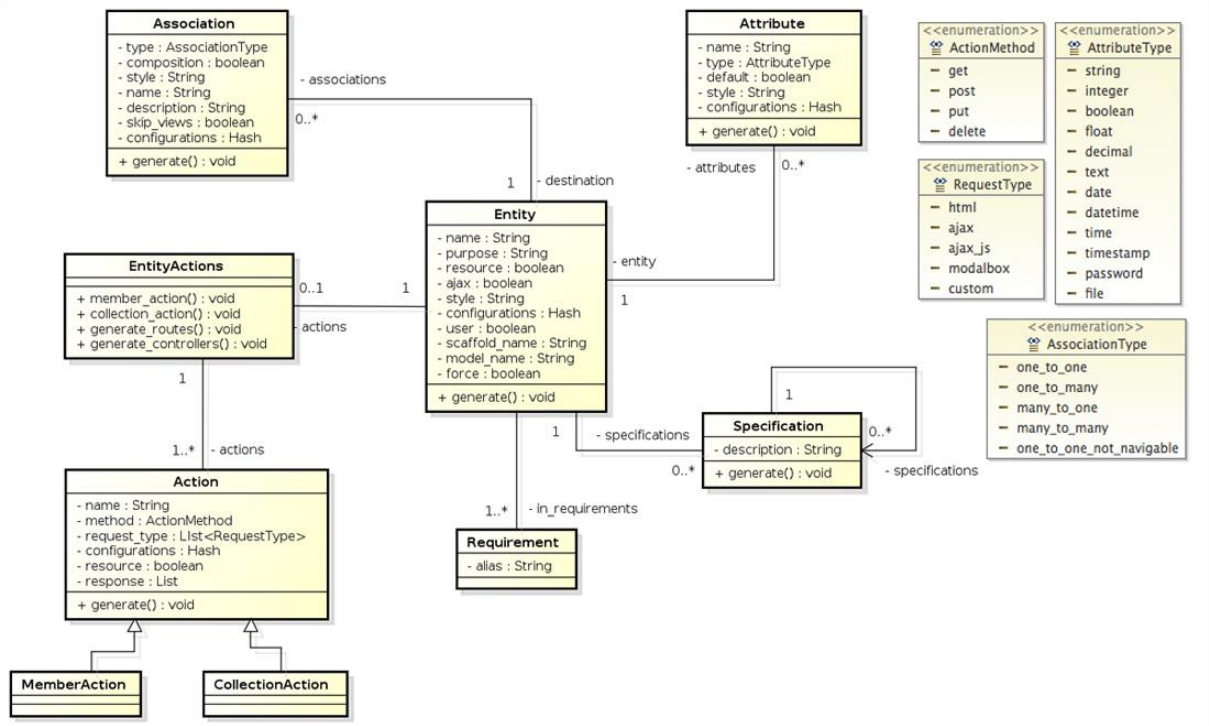


Figura 5.6: Metamodelo de Entidades MDWA.

Modelos de entidades são adequados para gerar código estrutural, portanto devem representar detalhes de implementação. Como o framework Ruby on Rails utiliza MVC como padrão arquitetural, os *templates* de entidades devem gerar um *model*, um *controller*, *views*, *helpers* e casos de testes automatizados. Segundo o metamodelo, além do relacionamento direto com os requisitos que implementa, uma entidade MDWA deve armazenar:

- Zero ou vários atributos para a classe do *model*, com seus métodos acessores, e mapeamento para colunas da tabela do banco de dados;

- Zero ou várias associações com outras entidades, que especificam relacionamentos entre *models* e geram chaves estrangeiras no banco de dados;

- Zero ou várias ações de *controller*, com suas *views* correspondentes;

- Zero ou várias especificações, que geram casos de testes detalhados.

Uma *Entity* possui várias propriedades: *name* (nome único da entidade), *purpose* (propósito da entidade na lógica da aplicação), *resource* (indica se a entidade gera cadastros), *user* (indica se a entidade gera código de cadastro de usuário), *ajax* (indica se o código gerado deve ser ajax), *scaffold_name* (nome dos cadastros gerados), *model_name* (nome do model gerado), *force* (indica se deve sobre-escrever o código sem fazer perguntas), *style* (padrão de

estilo de leiaute) e *configurations* (permite indicar configurações personalizadas para a geração de código).

Um *Attribute* representa um atributo do model gerado pela entidade e seu mapeamento para o banco de dados. Suas propriedades são: *name* (nome), *type* (um dos valores de AttributeType), *default* (indica se o atributo é o padrão para a Entity), *style* (padrão de estilo de leiaute) e *configurations* (permite indicar configurações personalizadas para a geração de código).

Uma *Association* representa um relacionamento entre entidades. Suas propriedades são: *destination* (nome da entidade referenciada), *type* (um dos valores de AssociationType), *composition* (indica se o relacionamento é uma composição), *name* (altera o nome padrão do relacionamento), *description* (descrição do relacionamento, por exemplo, projeto pertence a um grupo), *skip_views* (indica se o relacionamento não deve ser apresentado nas views), *style* (padrão de estilo de leiaute) e *configurations* (permite indicar configurações personalizadas para a geração de código).

EntityActions é uma classe que armazena métodos para inclusão e geração de *Actions* em uma *Entity*. *Actions* representam ações de *controller* e suas respectivas *views*. *Actions* podem ser classificadas entre *MemberActions*, que são *actions* que tratam de apenas um objeto, e *CollectionActions*, que tratam vários objetos. Essa separação é necessária para a montagem adequada de rotas URL. As propriedades de uma *Action* são: *name* (nome da ação), *method* (um dos valores de *ActionMethod*), *request_type* (um conjunto de valores de RequestType), *resource* (indica se a action refere-se a uma operação de cadastro), *configurations* (permite indicar configurações personalizadas para a geração de código) e *response* (um hash que mapeia o *request_type* com sua resposta, como por exemplo, um redirecionamento para formato HTML e a renderização de um arquivo Javascript para chamadas AJAX).

Uma *Entity* pode possuir várias *Specifications*, que representam restrições aplicadas ao funcionamento da entidade e são utilizadas para geração de casos de testes. Uma *Specification* tem apenas uma propriedade: *description*, que é um texto com uma descrição da restrição. Uma *Specification* pode conter um conjunto de *Specifications*, como forma de detalhamento de um algoritmo referente à lógica de negócio da aplicação, por exemplo, "Validar produtos no carrinho de compras" pode ter *Specifications* "Quantidade deve maior

que zero", "Produtos sem estoque não podem ser comprados", "Produtos com instâncias devem ter seu item correto selecionado".

```
MDWA::DSL.entities.register "Project" do |e|

  e.purpose   = %q{Holds project and status} # what does this entity do?
  e.resource  = true      # should it be stored like a resource?
  e.ajax      = true      # scaffold with ajax?
  e.scaffold_name = 'a/project' # mdwa sandbox specific code?

  ##
  ## Define entity attributes
  e.attribute 'name','string'
  e.attribute 'active', 'boolean'
  e.attribute 'actual_status', 'text'

  ##
  ## Define entity associations
  e.association do |a|
    a.type = 'many_to_one'
    a.destination = 'ProjectGroup' # entity name
    a.description = 'Belongs to a group'
  end
  e.association do |a|
    a.type = 'one_to_many'
    a.destination = 'Milestone' # entity name
    a.description = 'Projects have deadlines'
  end

  #
  # Controller actions
  e.member_action :publish, :method => :get, :request_type => :ajax
  e.collection_action :export, :method => :post, :request_type => [:csv, :xls]

  #
  # Specifications and restrictions
  e.specify "fields should be valid" do |s|
    s.such_as "name should not be blank"
    s.such_as "project group should not be null"
    s.such_as "actual status can be blank"
  end

end

MDWA::DSL.entity('Project').in_requirements << 'Manage projects and status'

MDWA::DSL.entity('Project').code_generations << %q{mdwa:scaffold a/project name:string active:boolean actual_status:text
  projectgroup:a/project_group:name:belongs_to milestone:a/milestone:name:has_many --ajax --skip_rake_migrate  --skip-questions}
```

Figura 5.7: Modelo de entidade MDWA representado na DSML Ruby-MDWA.

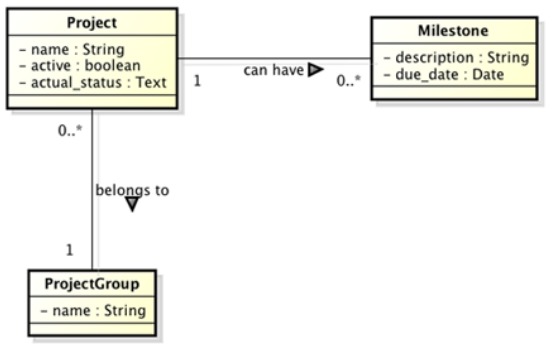


Figura 5.8: Diagrama de classes UML que representa os dados apresentados na Figura 5.7.

Na Figura 5.8 é apresentado um diagrama de classes UML que é traduzido para o modelo de entidade ilustrado na Figura 5.7. Ambos representam os mesmos dados, embora de forma diferente.

O modelo de classes UML não possui a representatividade adequada para expressar todos os dados referentes às entidades, como por exemplo, como o cadastro deve ser gerado ou suas restrições para casos de testes. Uma maneira de contornar esse problema é com a criação de estereótipos UML (OMG, 2012c), contudo esse modo também não é adequado, por acrescentar complexidade e dificultar a manutenção do diagrama de classes. Modelos específicos de domínio podem representar todas as informações necessárias e favorecem a tarefa de geração de código.

Dessa forma, modelos de entidade podem gerar código fonte de CRUD, por meio de *templates*, contendo operações de cadastro, *models* e relacionamentos, rotas URL, *views*, *controllers*, atualização do estado do banco de dados (tabelas, atributos e tipos de dados) e casos de testes. A transformação de modelos de entidades para templates de código é tratada na Seção 5.8 deste capítulo.

5.6 Metamodelo de Processos

Ao final da Etapa 4 da abordagem MDWA, Identificação de Processos, o artefato gerado é um modelo de processo para cada processo identificado. Um processo MDWA é um conjunto de atividades que um ou mais personagens executam de forma ordenada, com o objetivo de fornecer um artefato ou serviço. Inspirados em diagramas BPMN (OMG, 2012d) e diagramas de atividades UML (OMG, 2012c), a ferramenta Ruby-MDWA define um metamodelo para representação de processos de negócio, conforme ilustrado na Figura 5.9. Assim como os demais metamodelos propostos anteriormente neste capítulo, a instanciação de modelos de processos é efetuada por meio de uma linguagem de modelagem textual, provida pela ferramenta proposta. Um exemplo de modelo de processo representado pela DSML textual é apresentado na Figura 5.10.

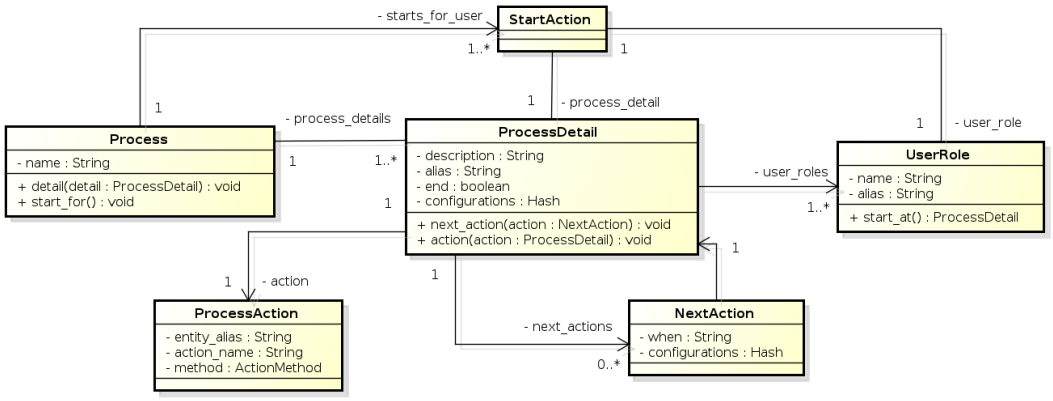


Figura 5.9: Metamodelo de Processos MDWA.

```
MDWA::DSL.workflow.register 'Manage projects' do |p|

  # The process start point for every user role
  # Params: :user_role, alias or entity listing
  p.start_for 'ProjectManager', 'new_project_group'
  p.start_for 'TeamMember', 'new_task'

  p.detail 'New Project Group' do |d|
    d.user_roles = ['ProjectManager']
    d.action 'project_group', 'new'
    d.next_action 'create_project_group', :method => :post
  end

  p.detail "Create project group" do |d|

    d.user_roles = ['ProjectManager', 'TeamMember']
    d.action 'project_group', 'create'

    d.next_action 'new_project_group', :when => 'save failed'
    d.next_action 'project_group_list', :redirect => true, :when => 'save ok'
    d.next_action 'new_project', :when => 'clicked save & new project'

  end

end
```

Figura 5.10: Modelo de Processos baseado na DSL Ruby-MDWA.

Um modelo de processo, de acordo com seu metamodelo, é representado pela classe *Process* e deve ser identificado com um nome único, representado pela propriedade *description*.

Cada atividade realizada durante o processo é denotada pela classe *ProcessDetail* e possui como propriedades: uma descrição (*description*), um nome único (*alias*), um indicador de final de processo (*end*) e um conjunto de configurações para geração de código (*configurations*). Além das propriedades, uma atividade possui um ação de entidade correspondente, representada pela classe *ProcessAction*, um grupo de papéis de personagens que possuem permissão de execução da ação, identificado pela lista *user_roles*, e por último, um conjunto de possíveis próximas ações, representadas pela lista *next_actions* da classe

NextAction. As próximas ações podem ser realizadas de acordo com uma condição, identificada pela propriedade *when*, além de um conjunto de configurações (*configurations*) para geração de código.

Um modelo de processo também deve representar como cada personagem inicia seu fluxo de ações. Esse comportamento é modelado pela propriedade *starts_for_user* da classe *Process*, que é uma lista de mapeamentos entre cada papel de usuário (*UserRole*) e sua ação inicial (*ProcessDetail*).

Conforme o fluxo de atividades de geração de código ilustrado na Figura 5.1, a criação de um modelo de processo é realizada a partir de uma transformação indireta dos modelos de requisitos, pois um processo pode ser identificado com base em vários requisitos. Por isso, a criação de um modelo de processo é feita manualmente com comando de terminal:

```
rails generate mdwa:process "<nome do processo>"
```

5.7 Implantação da Infra-estrutura

Aplicações web, em sua maioria, possuem um conjunto de requisitos não-funcionais voltados a criar um ambiente pronto para receber a lógica de domínio. Esses requisitos fazem referência à arquitetura de software, segurança, controle e autenticação de usuários, criação e separação entre leiautes, configurações, manutenção de dependências de reuso, entre outros.

A construção desse ambiente é uma tarefa repetitiva, sem padrões e sujeita a falhas. A reutilização de software surge como forma de resolver esse problema, por meio de frameworks e padrões de projeto. Porém, é desejável que frameworks forneçam reuso em um domínio específico, assim como padrões. A implementação de todos esses requisitos requerem a combinação entre frameworks, padrões de software e tecnologias diferentes. A combinação desses fatores aumenta a complexidade de construção desse ambiente e dificultam sua manutenção durante o ciclo de vida do software. É desejável que uma vez o ambiente criado, seja possível reutilizar seu código e configurar suas particularidades.

O ambiente de infra-estrutura tem como objetivo implementar uma variedade de particularidades inerentes ao domínio antes do desenvolvimento efetivo da aplicação. Ele deve englobar: arquitetura; conexão com banco de dados; segurança e encriptação de dados sigilosos; controle e permissões de usuários; criação de múltiplos leiautes e apresentação personalizada; gerenciamento de dependências de bibliotecas e frameworks; configurações;

inclusão de dados padrões, como por exemplo, criação de super usuário e tipos pré-definidos de permissões; preparo do ambiente de testes automatizados.

A implementação de todos os requisitos não-funcionais identificados anteriormente é encapsulada em um gerador de código. Dessa forma, garante-se que as soluções funcionem entre si e possam ser reusadas e configuradas de acordo com as necessidades de cada aplicação. Para invocar a geração da infra-estrutura com a ferramenta Ruby-MDWA, utiliza-se o comando de terminal:

```
rails generate mdwa:infrastructure
```

Na Figura 5.11 é apresenta a página pública gerada como parte da infra-estrutura, enquanto que na Figura 5.12 é apresentada a página de *login* de usuários, que restringe o acesso as páginas administrativas, ilustrada na Figura 5.13. Assim, pode-se visualizar a separação entre leiautes, permissões de usuários e operações de usuários geradas pelo ambiente de infra-estrutura. A arquitetura da aplicação é fornecida pelo framework Ruby on Rails.

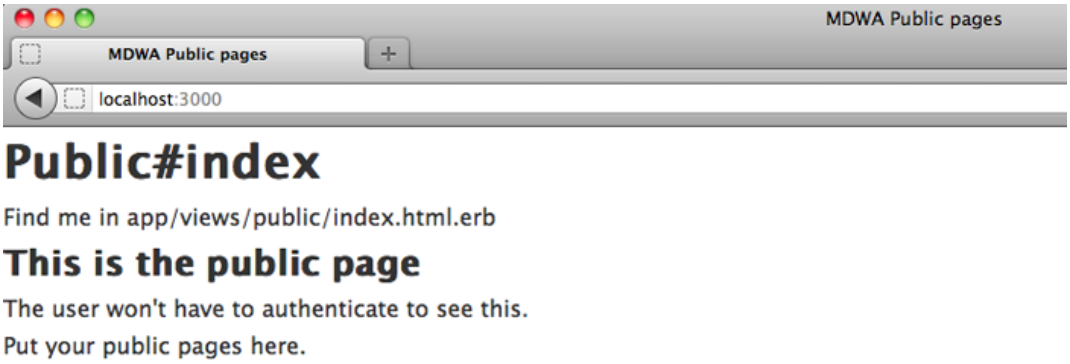


Figura 5.11: Página pública MDWA.

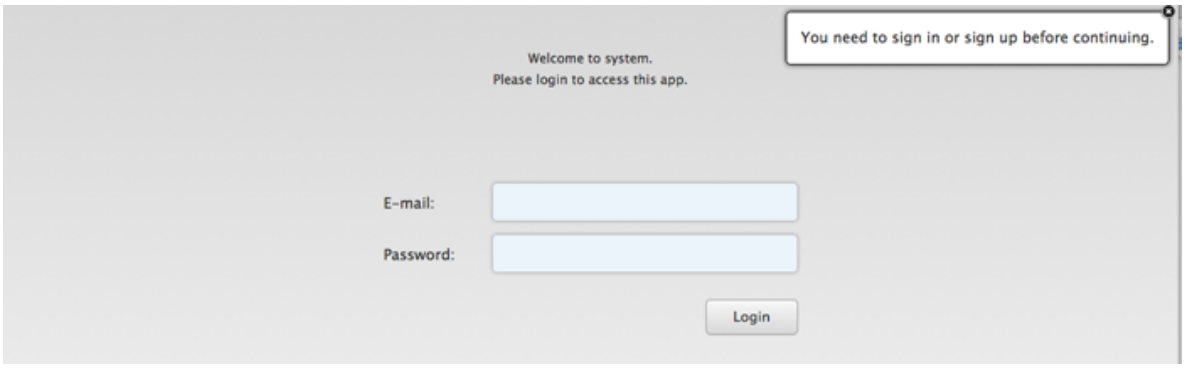


Figura 5.12: Página de login de usuários MDWA.

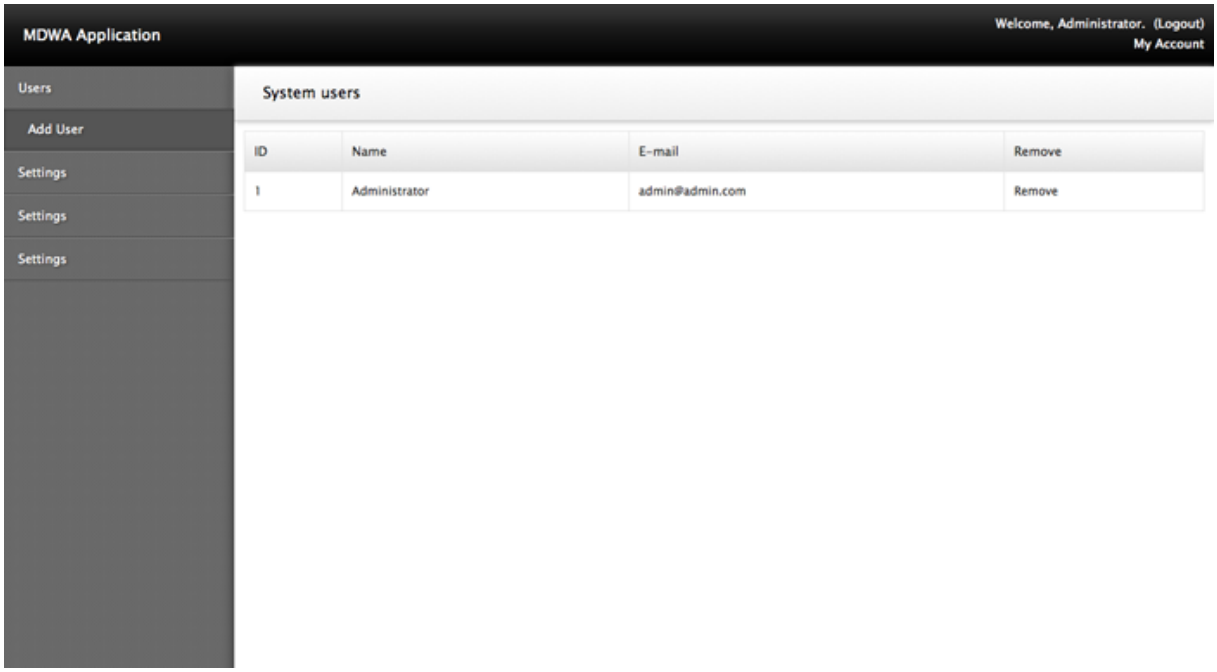


Figura 5.13: Página administrativa MDWA.

A construção da infra-estrutura é um passo intermediário entre os modelos de entidades, personagens e processos e suas transformações para *templates* de código, segundo o diagrama de atividades da abordagem MDWA, ilustrado na Figura 4.1. Tomando como base as necessidades de sua aplicação e sua infra-estrutura, o desenvolvedor deve decidir entre efetuar a transformação para *templates* de código ou alterar o ambiente de infra-estrutura gerado.

5.8 *Templates* de Código

De acordo com a definição previamente apresentada, *templates* são arquivos que representam meta-código, pois, com base em uma DSL textual, acessam as informações especificadas em modelos para tomada de decisões referentes à geração do código fonte da aplicação.

A ferramenta Ruby-MDWA propõe a adoção de *templates* descritos em formato ERB, uma DSL nativa da linguagem Ruby, que possuem um compilador embutido no núcleo da linguagem. A existência desse compilador evitou a tarefa de construção ou seleção de uma ferramenta similar, pois se trata de uma funcionalidade provida pela própria linguagem de programação e seu funcionamento não conflita com os demais componentes utilizados.

Com t*emplates* ERB é possível gerar código fonte em qualquer linguagem de programação, inclusive novos *templates* ERB de forma recursiva, conforme é ilustrado na Figura 5.15.

Outra vantagem da adoção de *templates* ERB é que o método de acesso aos dados dos modelos também é feito em Ruby. Assim, o desenvolvedor escreve código Ruby para processar as informações dos modelos e para implementar a lógica de negócio da aplicação. Dessa forma, não é necessário que os desenvolvedores sejam introduzidos a outras linguagens de programação ou DSLs, minimizando o aprendizado necessário para o desenvolvimento. Isso pode ser observado na Figura 5.14 que ilustra o *template* ERB equivalente a implementação do *model* MVC para o framework Ruby on Rails.

```
===entity_code===
class <%= @model.klass %> < <%= !@entity.user? ? 'ActiveRecord::Base' : 'User' %>

<%- # model attributes -%>
<%- unless @model.attributes.count.zero? -%>
    attr_accessible <%= @model.attributes.collect {|a| ":" + a.name }.join(', ') %>
<%- end -%>
<%- # paperclip file uploads -%>
<%- unless @model.attributes.select{|attr| attr.type.to_sym == :file}.count.zero? -%>
    has_attached_file <%= @model.attributes.select{|attr| attr.type.to_sym == :file}.collect {|a| ":" + a.name }.join(', ') %>
<%- end -%>

<%- if @entity.user? -%>
  <%- require_all "#{MDWA::DSL::USERS_PATH}#{@entity.file_name}.rb" -%>
  <%- @roles =  MDWA::DSL.user(@entity.name).nil? ? @roles = [@model.name] : MDWA::DSL.user(@entity.name).user_roles -%>
  <%- @roles.each do |role| -%>
    after_create :create_<%= role.underscore %>_permission
    def create_<%= role.underscore %>_permission
      <%= role.underscore %>_permission = Permission.find_by_name('<%= role.underscore %>')
      <%= role.underscore %>_permission = Permission.create(:name => '<%= role.underscore %>') if <%= role.underscore %>_permission.nil?
      self.permissions.push <%= role.underscore %>_permission
    end
  <%- end -%>
<%- end -%>

<%- # model associations -%>
<%- @model.associations.each do |association| -%>
  <%- if association.belongs_to? -%>
    belongs_to :<%= association.model2.singular_name %>, :class_name => '<%= association.model2.klass %>'
    attr_accessible :<%= association.model2.singular_name.foreign_key %>
  <%- end -%>
  <%- if association.has_one? -%>
    has_one :<%= association.model2.singular_name %>, :class_name => '<%= association.model2.klass %>'
  <%- end -%>
  <%- if association.has_many? -%>
    has_many :<%= association.model2.plural_name %>, :class_name => '<%= association.model2.klass %>'
  <%- end -%>
  <%- if association.has_and_belongs_to_many? -%>
    has_and_belongs_to_many :<%= association.model2.plural_name %>, :join_table => :<%= association.ordered.first.plural_name %>_<%= association.ordered.last.plural_name %>
  <%- end -%>
  <%- if association.nested_one? -%>
    belongs_to :<%= association.model2.singular_name %>, :class_name => '<%= association.model2.klass %>'
    attr_accessible :<%= association.model2.singular_name %>_attributes, :<%= association.model2.singular_name.foreign_key %>
    accepts_nested_attributes_for :<%= association.model2.singular_name %>, :allow_destroy => true
  <%- end -%>
  <%- if association.nested_many? -%>
    has_many :<%= association.model2.plural_name %>, :class_name => '<%= association.model2.klass %>', :dependent => :destroy
    attr_accessible :<%= association.model2.plural_name %>_attributes
    accepts_nested_attributes_for :<%= association.model2.plural_name %>, :allow_destroy => true
  <%- end -%>
<%- end -%>

end
```

Figura 5.14: *Template* que representa um *model* na implementação da arquitetura MVC do framework Ruby on Rails.

```
===entity_code===
<div id="<%= @model.plural_name %>_new" class="mdwa_new">
  <div class="page_header">
    <h1><%%= t('<%= @model.plural_name %>.new_title') %></h1>
  </div>

  <div class="inside">
    <%%= render 'form' %>
  </div>
</div>
```

Figura 5.15: *Template* que representa código HTML gerado dinamicamente com ERB.

A etapa 6 da abordagem MDWA, Detalhamento da Lógica de Negócio, se inicia com a geração dos *templates* em conforme com a arquitetura da aplicação, definida na etapa de implantação da infra-estrutura. Visto que o framework Ruby on Rails utiliza o padrão arquitetural MVC com a adição de *helpers*, os *templates* gerados são organizados por entidade e respeitam a estrutura apresentada na Figura 5.16, onde a pasta "a" representa o *namespace* das páginas administrativas, conforme definido pelo ambiente de infra-estrutura.

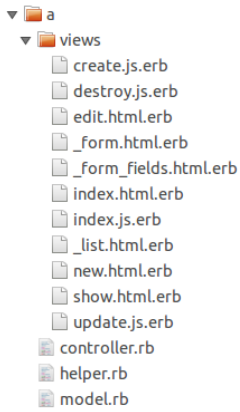


Figura 5.16: Estrutura de organização dos *templates* para um entidade qualquer.

A geração de *templates* é executada com o comando de terminal abaixo e tem como parâmetro não obrigatório uma lista de entidades separadas por vírgula:

```
rails g mdwa:templates [Entidade1 Entidade2 ...]
```

Os *templates* gerados para cada entidade, fornecem implementações padrões para: operações de persistência de objetos em banco de dados, criação de casos de testes automatizados e fluxo de navegação para processos.

Posterior a geração de código, os desenvolvedores devem codificar nos *templates* as regras de negócio da aplicação e escrever os casos de testes para verificação do funcionamento dos requisitos.

5.9 Transformação para Código Fonte

Ao iniciar o planejamento de uma entrega, a equipe de software deve identificar um conjunto de requisitos necessário para o ciclo de implementação considerado e criar modelos de requisitos. Então, os requisitos devem ser refinados e transformados em modelos de personagens, entidades e processos, de forma que a estrutura lógica da aplicação seja definida. Posteriormente, devem ser gerados *templates* que utilizam as informações modeladas para a tomada de decisões referentes à implementação das particularidades de negócio, bem como, criar casos de testes para validação do funcionamento desejado para a aplicação.

Nesse momento do desenvolvimento, todos os comportamentos que o software deve realizar são representados em modelos e suas particularidades definidas em *templates*. Dessa forma, o desenvolvedor deve a realizar a transformação desses modelos para a aplicação em funcionamento.

Essa transformação é feita com a compilação dos templates de código, bem como: sincronização entre os modelos e o esquema de banco de dados, mapeamento de rotas *URL* para ações de *controller* MVC, arquivos de linguagens para apresentação de conteúdo em diversas linguagens, configuração de dependências e validação dos testes. A realização dessas funções é efetuada pelo comando de terminal abaixo:

```
rails g mdwa:transform [Entidade1 Entidade2 ...]
```

Nas figuras Figura 5.17, Figura 5.18 e Figura 5.19 são apresentadas as telas de código gerado e customizado para uma entidade Projeto e um personagem Programador. Elas são parte do caso de estudo desenvolvido na Seção 6.4.1.

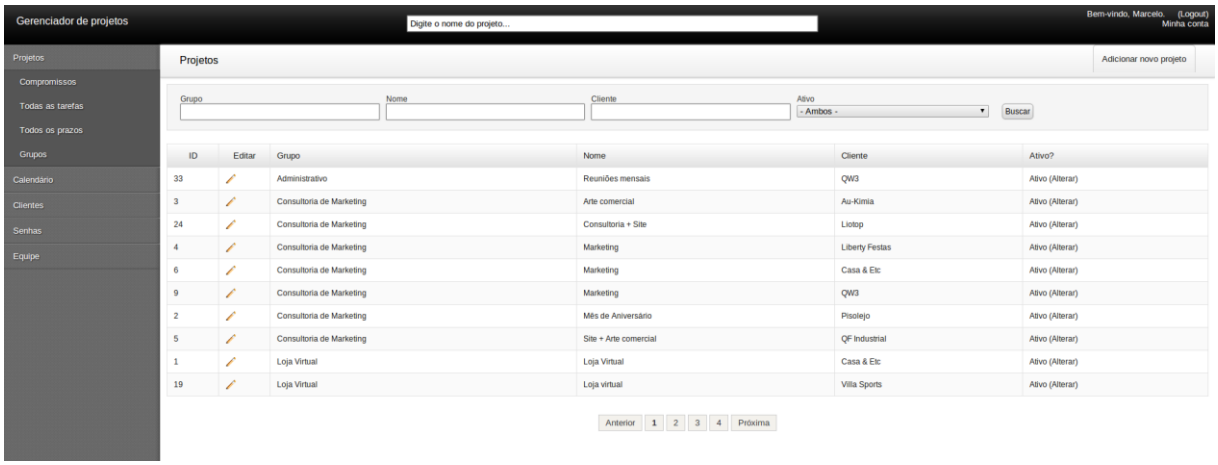


Figura 5.17: Exemplo de código gerado pela transformação da entidade Projeto.

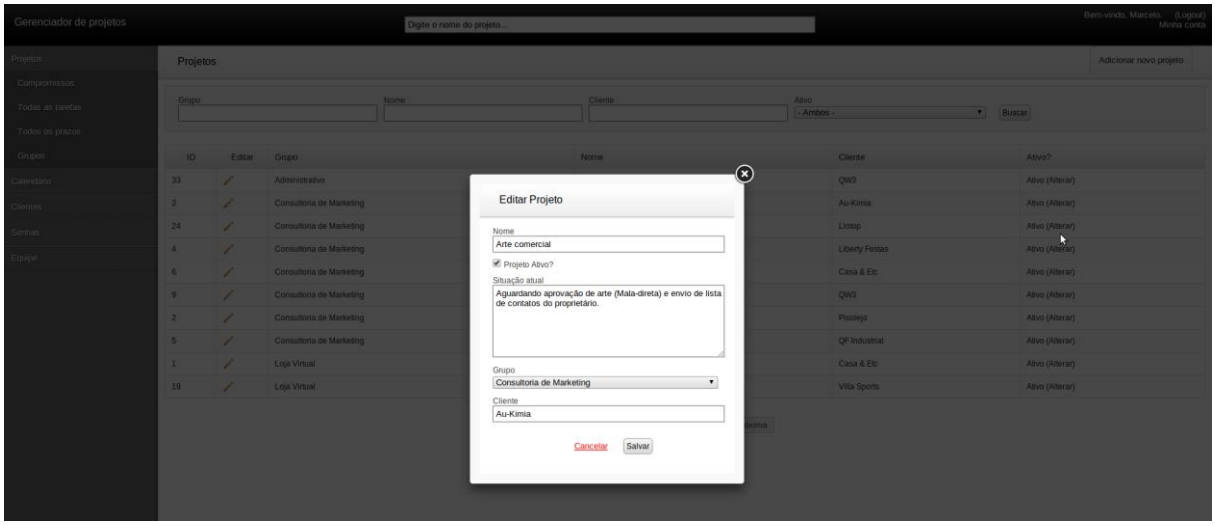


Figura 5.18: Exemplo de código gerado para edição de um projeto.

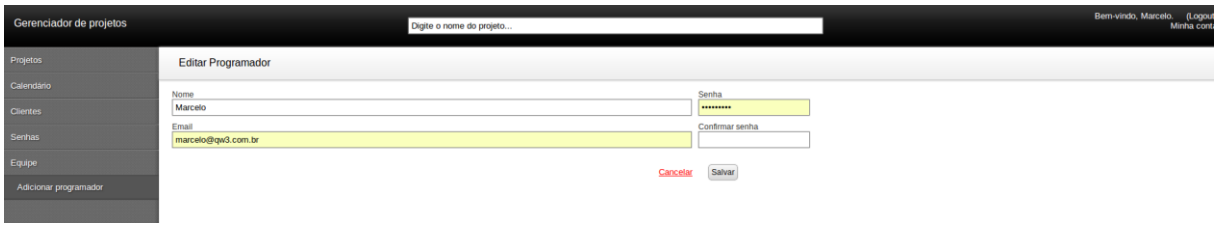


Figura 5.19: Exemplo de código gerado por um modelo de personagem Programador.

5.10 Considerações finais

A ferramenta Ruby-MDWA baseia-se em conceitos avançados da linguagem Ruby para criação de uma DSML textual interna que é utilizada para a representação dos modelos de requisitos, entidades, processos e personagens, definidos pela abordagem MDWA.

Essa decisão reduziu o esforço de construção da ferramenta e possibilitou sua utilização independentemente do sistema operacional ou de ambientes de desenvolvimento, como Eclipse, por exemplo. Além disso, a curva de aprendizado por parte dos desenvolvedores é reduzida, pois é requerido o conhecimento de somente uma linguagem de programação para a construção de modelos, *templates* e programação da lógica de negócio.

Portanto, modelos textuais podem ser representados por DSLs baseadas em código fonte, ou por linguagens declarativas já conhecidas, como por exemplo, XML (W3C, 2012), YAML (YAML, 2012) ou JSON (JSON, 2012). Dessa forma, os projetistas de software podem desenvolver aplicações guiadas por modelos utilizando as tecnologias que já estão

habituados. Por outro lado, a DSL interna possui problemas quanto à expressividade, pois sua sintaxe é baseada na linguagem Ruby e pode dificultar a utilização por não-programadores.

Essa experiência pode ser reproduzida em outras linguagens de programação com apoio de outros frameworks, permitindo a reutilização da abordagem de desenvolvimento, dos métodos de representação de modelos e da construção dos transformadores. Com isso, os conceitos propostos pelo MDD são mantidos e os esforços para sua utilização são reduzidos.

6 Estudo de Caso Comercial

6.1 Considerações Iniciais

A abordagem MDWA auxilia o engenheiro de software a desenvolver aplicações Web guiadas por modelos. O apoio da ferramenta Ruby-MDWA fornece modelos baseados em uma DSML textual e mecanismos de transformação entre eles. O experimento apresentado no Capítulo 7 indica que a combinação da abordagem com a ferramenta oferece maior produtividade e qualidade na construção de aplicações Web. Porém, a utilização de desenvolvedores sem experiência na linguagem Ruby e no framework Ruby on Rails reduz a precisão dos resultados obtidos.

Por esse motivo, foi elaborado um estudo de caso em conjunto com uma empresa de software, denominada QW3, instalada na cidade de São Carlos – SP. Este Capítulo descreve o planejamento e execução do estudo de caso e apresenta uma análise sobre os resultados obtidos.

Além desta Seção, o Capítulo está organizado da seguinte maneira: na Seção 6.2 é descrita a aplicação desenvolvida, um software de gerenciamento de projetos e na Seção 6.3 é apresentado o planejamento do estudo; na Seção 6.4 são exibidos os dados coletados e na Seção 6.5 é feita a análise dos resultados obtidos; na Seção 6.6 são apresentadas as considerações finais.

6.2 Descrição da Aplicação: Software de Gerenciamento de Projetos

O software escolhido para o estudo de caso foi um sistema de gerenciamento de projetos, pois: i) trata-se de uma aplicação maior que as aplicações testadas no experimento; ii) a equipe de desenvolvimento possui conhecimento avançado em Ruby e Ruby on Rails; iii) a empresa tem interesse em utilizar o software para seu processo de funcionamento interno e iv) a empresa tem interesse em efetuar manutenções evolutivas.

O sistema de gerenciamento de projetos deve controlar projetos e suas tarefas a serem realizadas, bem como, deve gerenciar os prazos de entregas e reuniões de um projeto. Além disso, é necessário armazenar os dados do cliente de cada projeto e seus contatos. Na Tabela 6.1: Requisitos do Software Gerenciador de Projetos. estão exibidos alguns requisitos do software de gerenciamento de projetos e na Figura 6.1: Diagrama de classes UML do

software de gerenciamento de projetos. é apresentado o modelo de classes UML que representa os requisitos da aplicação.

#	Descrição
1	O sistema deve gerenciar projetos. Projetos são classificados por um grupo., suas tarefas e prazos de entrega. Tarefas e prazos devem armazenar o status de andamento, as datas previstas e de entrega e o responsável pela execução.
2	Um projeto pode ter zero ou várias tarefas e prazos de entrega. Uma tarefa possui título, descrição, datas de entrega e finalização, *status* e prioridade. Prazos de entrega possuem título, descrição, datas de entrega e finalização e um indicativo se foi entregue.
3	Um projeto atende a um cliente. Um cliente pode ter um ou mais contatos. As informações do cliente são: nome, telefone, documento e observações.
4	O acesso dos membros da equipe de desenvolvimento deve ser autenticado por email e senha.
5	Um projeto pode ter zero ou várias reuniões agendadas. Uma reunião contém data, descrição e responsável.
6	Um projeto pode ter zero ou várias interações entre os membros. Uma interação é composta por data, descrição e responsável.

Tabela 6.1: Requisitos do Software Gerenciador de Projetos.

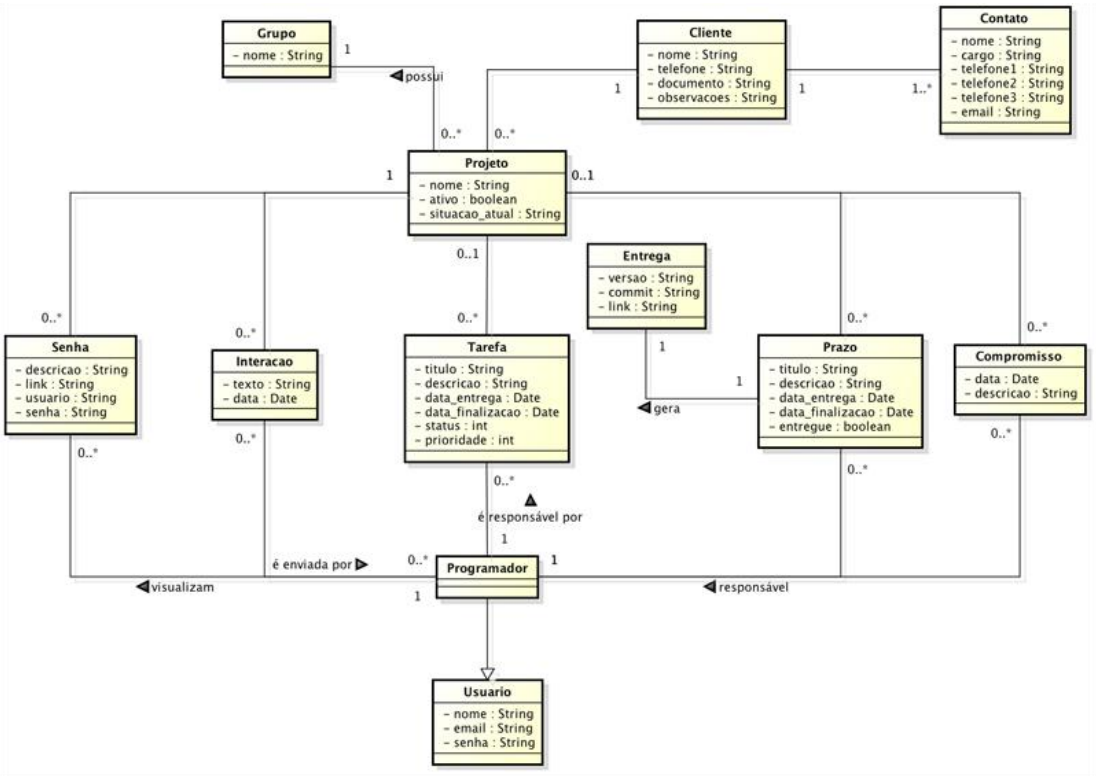


Figura 6.1: Diagrama de classes UML do software de gerenciamento de projetos.

6.3 Planejamento do Estudo de Caso

O estudo de caso tem por objetivo verificar que a ferramenta Ruby-MDWA auxilia na construção e manutenção do software descrito anteriormente. Pretende-se também verificar o tempo consumido nas atividades de modelagem e de transformação entre modelos. Este estudo de caso teve a finalidade também de realizar refinamentos na ferramenta Ruby-MDWA e na abordagem MDWA, atendendo às solicitações dos participantes do estudo de caso.

Neste estudo foram analisados: a qualidade da geração de código fonte estrutural, com base em modelos de entidades, processos e personagens; o tempo gasto no desenvolvimento e o comportamento dos desenvolvedores quando forem realizar manutenções guiadas por modelos e *templates* de código. O processo empregado nesta verificação foi:

i) Para verificar a qualidade da geração de código fonte foi utilizada a métrica *code churn* (Nagapann; Ball, 2005) que mede mudanças no código, identificando o número de linhas de código acrescentadas, modificadas e removidas. Foi utilizada a ferramenta *open-source* Cloc (CLOC, 2012) para efetuar as medições.

ii) Medição do tempo de desenvolvimento e de manutenção. O tempo de desenvolvimento trata-se do período entre a primeira reunião do projeto e a publicação da primeira versão em produção. O tempo de manutenção refere-se ao período após a publicação até a última modificação efetuada. A medição do tempo foi dividida entre: i) construção e alteração dos modelos Ruby-MDWA; ii) implementação da lógica de negócio nos *templates*; iii) execução das transformações entre modelos; iv) codificação e execução de testes. Para as medições de tempo foram utilizados relógios digitais comuns.

Os participantes foram três funcionários da empresa com dois anos ou mais de experiência em programação com Ruby e Ruby on Rails, porém sem experiência em desenvolvimento guiado por modelos.

A execução do estudo de caso foi dividida em três fases: i) treinamento, ii) planejamento e desenvolvimento da primeira versão e iii) manutenção. Inicialmente, foi ministrado um treinamento de doze horas aos desenvolvedores sobre a abordagem proposta e sua ferramenta de apoio, com a finalidade de fornecer embasamento teórico para o desenvolvimento.

Na segunda fase, foi planejado o cronograma da primeira entrega, bem como a divisão de tarefas entre a equipe. Após o desenvolvimento do software, obteve-se um protótipo construído a partir da geração de código estrutural da ferramenta Ruby-MDWA. Na terceira fase, foram efetuadas manutenções evolutivas, a partir da introdução do código de negócio.

6.4 Dados Coletados

Nesta Seção são apresentados os dados coletados da realização desse estudo de caso. Os desenvolvedores seguiram o fluxo de etapas de geração de código da ferramenta Ruby-MDWA, ilustrado na Figura 5.1, para construção da aplicação guiados por modelos, *templates* de código e transformações. Para o levantamento dos dados utilizou-se o código fonte da aplicação, desconsiderando os *templates* de código. A versão de desenvolvimento do software foi gerada a partir dos *templates* padrões gerados pela ferramenta. A versão final da aplicação foi gerada como produto da customização dos *templates* com base na lógica de negócio da aplicação.

Nas Tabelas Tabela 6.2 e Tabela 6.3 são apresentados dados referentes a comparação entre a versão de desenvolvimento, gerada completamente pela ferramenta Ruby-MDWA, e a versão final da aplicação, produzida ao final do processo de manutenção. Essa comparação está representada na Tabela 6.2: Modificações no código na comparação entre as versões de desenvolvimento e final. por meio da métrica *code churn* (Nagapann; Ball, 2005), possibilitando a análise da quantidade de código gerado pela Ruby-MDWA no final do desenvolvimento.

Elemento	MDWA Gerado	Modificações	Criações	Remoções	LOC Final	LOC MDWA Final	LOC MDWA Final (%)
***Models* MVC**	86	25	124	4	206	53	26%
***Controllers* MVC**	544	28	172	30	686	456	66%
***Views* MVC**	1474	159	399	278	1595	759	48%
Javascripts	257	0	157	0	414	257	62%
Estilos CSS	1403	3	62	2	1463	1396	95%
Configurações	606	37	48	66	588	437	74%
Rails *Migrations*	136	3	0	0	136	133	98%
Soma	**4506**	**255**	**962**	**380**	**5088**	**3491**	**Média: 69%**

Tabela 6.2: Modificações no código na comparação entre as versões de desenvolvimento e final.

Na Tabela 6.3 é apresentado o resumo dos dados descritos na Tabela 6.2: Modificações no código na comparação entre as versões de desenvolvimento e final., agrupando *models, views* e *controllers* em um componente "Lógica de Negócio MVC",

arquivos Javascript e CSS em "*Assets*" e configurações e *migrations.* As configurações representam principalmente, arquivos de tradução e diretivas de configuração de servidor, como por exemplo, fuso-horário e regras de persistência.

Componente	% Código gerado MDWA
Lógica de Negócio – MVC	47%
Assets - JS e CSS	79%
Configurações + Rails *Migrations*	86%

Tabela 6.3: Resumo da quantidade de código gerado pela ferramenta Ruby-MDWA por componente.

Nas Tabelas Tabela 6.4 e Tabela 6.5 são apresentados tempos gastos com as atividades de desenvolvimento e manutenção, respectivamente. Foi efetuada a medição do tempo de quatro atividades: i) construção e alteração dos modelos Ruby-MDWA; ii) implementação da lógica de negócio nos *templates* de código; iii) codificação dos testes e execução e iv) execução das transformações.

Atividade	Tempo (em horas)	% do total
Construção e alteração dos modelos Ruby-MDWA	21,7	55%
Implementação da lógica de negócio	12,95	33%
Codificação e execução de testes	4,32	11%
Execução de transformações	0,25	1%
Soma	**39,22**	**100%**

Tabela 6.4: Distribuição de atividades do tempo de desenvolvimento.

Atividade	Tempo (em horas)	% do total
Construção e alteração dos modelos Ruby-MDWA	22,11	24%
Implementação da lógica de negócio	41,35	45%
Codificação e execução de testes	26,32	29%
Execução de transformações	1,56	2%
Soma	**91,34**	**100%**

Tabela 6.5: Distribuição das atividades do tempo de manutenção.

6.5 Análise dos Resultados

De acordo com os dados apresentados nas Tabelas Tabela 6.2 e Tabela 6.3, respectivamente, verifica-se que 69% do código final da aplicação foi gerado pela ferramenta Ruby-MDWA, em decorrência da geração de grande parte dos arquivos de estilos CSS (95%),

configurações (74%) e *migrations* (98%). Em referência a lógica de negócio, 47% do código é gerado pela ferramenta.

Na Figura 6.2 é apresentado o gráfico de barras que representa a porcentagem de código gerado pela ferramenta Ruby-MDWA na versão final da aplicação. A linha azul representa a porcentagem de cada elemento e linha vermelha representa a média de código gerado pela ferramenta.

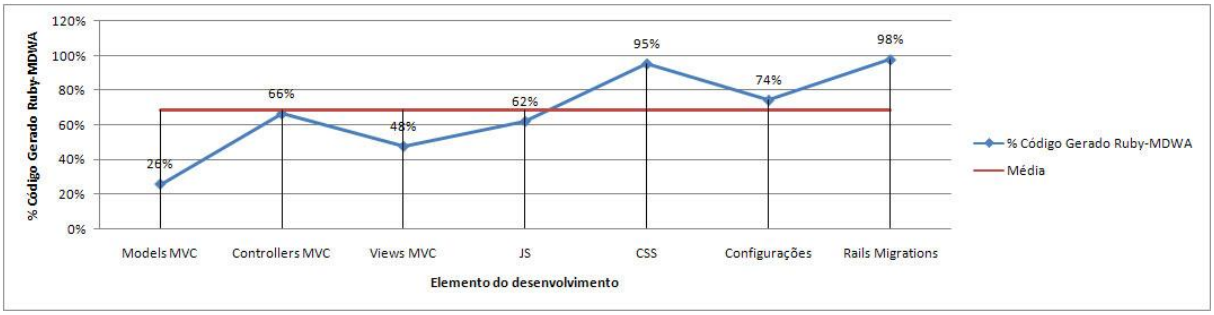


Figura 6.2: Porcentagem de código gerado pela ferramenta Ruby-MDWA presente na versão final.

Com base nos índices apresentados na Figura 6.2, constata-se que os modelos de entidade representam com sucesso o relacionamento entre entidades, pois 98% das *migrations* são geradas corretamente. O índice de 26% da preservação do código gerado nos *models* MVC também confirma essa constatação, visto que a ferramenta gera somente atributos e associações nos *models* MVC e o restante representa as particularidades de negócio da aplicação. Verifica-se também que os modelos de processo e entidade representam corretamente as ações que a aplicação executa, visto que 66% do código gerado é preservado nos *controllers* MVC.

A camada de apresentação, *views* MVC, reutiliza 48% do código gerado, indicando que os cadastros gerados pela ferramenta são úteis, assim como o alto índice de reaproveitamento dos arquivos de estilos CSS. Caso os desenvolvedores escolham outro estilo de apresentação, o índice de reaproveitamento de CSS deve ser reduzido. Quanto ao índice de 74% de reutilização do código gerado de configurações, trata-se do resultado da criação do ambiente de infra-estrutura e do algoritmo de tradução que faz parte da geração de entidades.

Na Figura 6.3 é apresentado um gráfico de pizza que representa o tempo de cada atividade realizada durante o desenvolvimento e manutenção. Pode-se notar que a atividade de modelagem utilizando os modelos da ferramenta Ruby-MDWA consome mais tempo na

versão de desenvolvimento (55%) em comparação a manutenção (24%). A implementação da lógica de negócio é mais demorada na manutenção (45%), assim como ocorre na atividade de codificação de testes (29%).

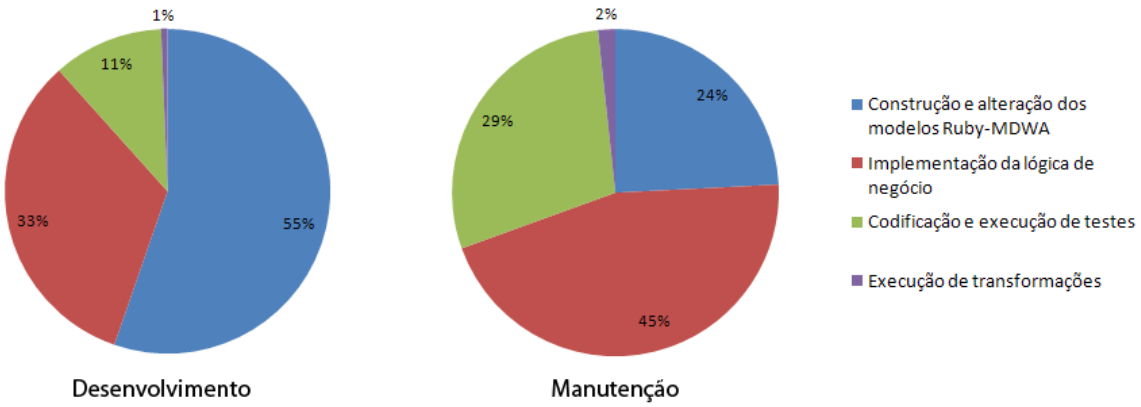


Figura 6.3: Gráfico de pizza que representa a distribuição de tempo das atividades de desenvolvimento e manutenção.

Esse cenário mostra que a atividade de modelagem consome mais tempo no início do desenvolvimento de um conjunto de requisitos e diminui à medida que as regras de negócio ficam mais complexas e necessitam ser implementadas e testadas. Em relação ao tempo de execução das transformações, o índice de 2% é alto, em decorrência do fato de a linguagem Ruby ser interpretada e por esse motivo, as transformações de modelos e *templates* são demoradas.

6.6 Ameaças à Validade

Pelo fato de o estudo de caso ter sido conduzido com três desenvolvedores da mesma empresa, os resultados obtidos podem ter sido influenciados pela cultura dos participantes e por trabalharem em conjunto. Por esse motivo, se o mesmo estudo de caso for aplicado a um grupo diferente de participantes, os resultados obtidos podem ser diferentes.

O fato de os desenvolvedores não possuírem experiência em desenvolvimento guiado por modelos pode influenciar os resultados obtidos, pois suas práticas são comuns a cultura de desenvolvimento guiado por código fonte. Os resultados podem ser diferentes se aplicados a uma população com experiência em MDD.

Como o estudo constituiu-se do desenvolvimento de uma aplicação de baixa complexidade em um curto período de tempo, os resultados das medições de tempo podem ser diferentes quando aplicados no desenvolvimento de um software mais complexo.

6.7 Considerações Finais

Neste Capítulo foi descrito um estudo de caso, realizado em conjunto com uma empresa de software, que permitiu analisar as vantagens providas pela ferramenta Ruby-MDWA e verificar o tempo de atividades de desenvolvimento com o apoio dessa ferramenta. Porém, outros estudos de caso podem ser futuramente realizados com o intuito de melhorar a qualidade dessa ferramenta.

A participação de desenvolvedores com experiência na linguagem Ruby e no framework Ruby on Rails facilitou a compreensão dos conceitos propostos e a utilização da ferramenta.

Com os dados obtidos, foi possível constatar que a ferramenta Ruby-MDWA auxilia o desenvolvimento de sistemas Web, automatizando tarefas repetitivas e provendo reuso baseado em modelos, geração de código e conhecimento de domínio.

O estudo de caso identificou que o esforço referente à modelagem é maior no início do desenvolvimento e diminui à medida que as particularidades de domínio precisam ser implementadas. Isso indica que a codificação da lógica de negócio ainda é uma atividade que demanda um tempo considerável de execução. Portanto é correto concentrar esforços em reuso como forma de melhorar a qualidade dos softwares e a produtividade dos desenvolvedores.

7 Avaliação do Uso da Ferramenta Ruby-MDWA

7.1 Considerações iniciais

Experimentos de software são conduzidos com o propósito de avaliar soluções em diferentes áreas de software, estabelecendo evidências empíricas sobre seu funcionamento em determinadas condições (Basili et al,1986). São usados para avaliar o uso da solução proposta frente a outros procedimentos existentes. Esses experimentos são conduzidos considerando um determinado número de participantes.

O desenvolvimento de software guiado por modelos com apoio ferramental pode melhorar os índices de produtividade, reuso e expressividade do domínio da aplicação. Porém, nem sempre somente vantagens podem ser citadas nesses casos. O desenvolver que está acostumado a trabalhar de uma determinada forma pode encontrar problemas quando usa uma diferente. Assim, os primeiros desenvolvimentos utilizando técnicas que utilizam modelos ao invés de código fonte, apresentam elevado número de artefatos. Com isso, desenvolvedores sem experiência em MDD encontram dificuldades quando iniciam o desenvolvimento de software com base em modelos.

Por outro lado, o desenvolvimento *ad-hoc*, baseado em código fonte, pode gerar problemas quanto à manutenibilidade e diminuição do conhecimento do domínio durante o ciclo de vida do software. A dificuldade em manter um software surge sempre que o desenvolvedor se ausenta do local de trabalho ou que os documentos por ele deixados não representam todos os detalhes contidos no software. Considerando essas premissas, a abordagem MDWA e ferramenta Ruby-MDWA foram desenvolvidas e apresentadas nos Capítulos 4 e 5, respectivamente.

Este Capítulo tem a finalidade de avaliar, por meio de um experimento controlado, o desenvolvimento de software Web, de forma *ad-hoc* com apoio do framework Ruby on Rails e desenvolvimento guiado por modelos com apoio da ferramenta Ruby-MDWA. Nessa avaliação foram considerados o tempo despendido com o desenvolvimento e qualidade das aplicações resultantes nos dois casos.

A apresentação do experimento realizado, é feita utilizando-se o formato descrito por Wholin et al (2000). Assim, o experimento foi definido, planejado, executado e seus dados

avaliados em comparação com as hipóteses levantadas. Na Seção 7.2 é apresentada a Definição do Experimento e na Seção 7.3 é descrito o Planejamento do Experimento. A Execução do Experimento e a análise dos resultados obtidos são apresentados, respectivamente, nas Seções 7.4 e 7.5. Por fim, na Seção 7.6 são exibidas as Considerações Finais.

7.2 Definição do Experimento

O objetivo deste experimento é avaliar duas formas de desenvolvimento de aplicações Web, com e sem apoio de ferramenta. Deseja-se avaliar se com o uso da ferramenta Ruby-MDWA os desenvolvedores de software obtêm aplicações em menor espaço de tempo e com maior qualidade de produto.

Duas aplicações serão desenvolvidas por todos os participantes do experimento. Os participantes serão separados em grupos e irão desenvolver uma aplicação utilizando a ferramenta Ruby-MDWA e uma outra aplicação de modo ad-hoc. O resultado esperado que o grupo que utilizar a ferramenta Ruby-MDWA apresente a solução em menor tempo do que o que está desenvolvendo de forma *ad-hoc*. Quanto a qualidade da aplicação produzida, espera-se que a de melhor qualidade seja a que foi desenvolvida com apoio da ferramenta Ruby-MDWA

Dois experimentos serão realizados em ambiente universitário: um com alunos de pós-graduação e outro com alunos de graduação. Os dois grupos de alunos terão a preparação para o experimento em momentos separados, bem como a realização dos mesmos. Dois sistemas serão apresentados para que os grupos de alunos desenvolvam as aplicações com e sem o apoio da ferramenta. Como os alunos de graduação e pós não tem contato, as aplicações solicitadas serão:

i) O tempo limite estipulado será de uma hora e esse será comparado com o tempo gasto por cada participante para o desenvolvimento de uma aplicação *ad-hoc* e outra utilizando a ferramenta Ruby-MDWA, independentemente de suas conclusões. Para isso, a métrica de produtividade (Pr), descrita pela Fórmula (1) será utilizada. Quanto mais alto o índice Pr do participante, melhor seu desempenho;

ii) A quantidade de funções implementadas durante o período de tempo estipulado para o desenvolvimento da aplicação será calculado. As métricas de

funcionalidades, descritas na Tabela 7.1, serão as utilizadas para essa medição. As métricas para avaliar produtividade PMC, PCC, PVC e PRM aceitam valores entre 0,0 e 1,0, que correspondem ao valor numérico da porcentagem de *models, contollers, views* e relacionamentos, respectivamente criados, de forma que 0 é o mais baixo e 1 é o mais alto. A métrica ILN, que verifica a implementação da lógica de negócio foi feita corretamente, aceita 1 quando a lógica está implementada corretamente e 0 caso contrário. O índice NE, que corresponde ao número de erros, pode ser representado por qualquer valor inteiro. Os erros contados pela métrica NE correspondem somente aos *models*, *views*, *controllers* e relacionamentos construídos corretamente, desconsiderando os que estão incompletos. A métrica QI, qualidade de implementação, é representada pela soma das porcentagens PMC, PCC, PVC e PRM, da lógica de negócio, ILN, e subtraído o número de erros, NE, privilegiando os participantes que produzem aplicações sem erros, mesmo que com requisitos parcialmente implementados. Todas as métricas, exceto NE, quanto maior o índice, melhor o desempenho do participante.

Pr = QI / T, Fórmula (1)

onde QI é o valor da métrica Qualidade de Implementação e T é o tempo gasto (em horas) por participante para desenvolvimento de cada aplicação.

Tabela 7.1: Métricas para avaliar a quantidade de funções implementadas durante o período de tempo estipulado.

Métrica	**Descrição**
PMC	Porcentagem de *Models* MVC Criados: representa a porcentagem de *models* MVC criados corretamente. É contado o número de *models* criados corretamente, cujos cadastros funcionam e dividido pelo número total de *models* indicados nos requisitos (5 para as duas aplicações).
PCC	Porcentagem de *Controllers* MVC Criados: representa a porcentagem de *controllers* MVC criados corretamente. É contado o número de *controllers* criados corretamente, cujos cadastros funcionam e dividido pelo número total de *controllers* indicados nos requisitos (5 para as duas aplicações).

PVC	Porcentagem de *Views* MVC Criadas: representa a porcentagem de *views* MVC criados corretamente. É contado o número de *views* criadas corretamente, cujos cadastros funcionam e dividido pelo número total de *views* indicadas nos requisitos (5 para as duas aplicações).
PRM	Porcentagem de Relacionamentos Mapeados: indica a porcentagem de relacionamentos corretamente apresentados na aplicação. É contado o número de associações criadas corretamente e dividida pelo número total de associações possíveis (8 para as duas aplicações).
ILN	Implementação da Lógica de Negócio: especifica se a lógica de negócio indicada nos requisitos foi implementada corretamente. Aceita 1 quando toda a lógica de negócio da aplicação está implementada corretamente e 0, caso contrário.
NE	Número de Erros: descreve o número de erros encontrados dentre as funções implementadas. Um erro é uma tela com erro, um problema ao salvar um recurso ou qualquer comportamento inesperado da aplicação.
QI	Qualidade da Implementação: PMC + PCC + PVC + PRM + ILN – NE

7.3 Planejamento do Experimento

7.3.1 Seleção do Contexto

O estudo foi realizado com estudantes de graduação em Bacharelado em Ciência da Computação e mestrado em Ciência da Computação da Universidade Federal de São Carlos (UFSCar), no segundo semestre de 2012. Os alunos de graduação estão cursando a disciplina Metodologias de Desenvolvimento de Software e os alunos de mestrado estão cursando a disciplina de Tópicos em Engenharia de Software: Reuso de Software. Nesse estudo, a tarefa dos participantes foi desenvolver duas aplicações Web, sendo uma de forma *ad-hoc* e outra com apoio da ferramenta Ruby-MDWA.

7.3.2 Definição das Hipóteses

Duas hipóteses foram elaboradas para esse experimento, uma refere-se à qualidade (H_{QI}) e a outra se refere à produtividade (H_{Pr}) de desenvolvimento.

Hipótese H_{Pr}	O apoio da ferramenta Ruby-MDWA torna o desenvolvimento de aplicações Web mais produtivo em comparação ao desenvolvimento *ad-hoc*.
Hipótese H_{QI}	A ferramenta Ruby-MDWA auxilia o desenvolvedor a construir aplicações Web com mais qualidade.

Tabela 7.2: Hipóteses do Experimento.

As hipóteses são baseadas nos estudos de caso desenvolvidos durante a construção da ferramenta e do estudo de caso real envolvendo uma empresa de software, apresentado no Capítulo 6. Os resultados obtidos indicam aumento de produtividade do desenvolvimento quando a ferramenta Ruby-MDWA é utilizada, pois uma parte do código é gerada e os desenvolvedores deixam de executar tarefas repetitivas e se concentram em implementar as particularidades de negócio da aplicação.

7.3.3 Seleção de Variáveis

As variáveis independentes são todas aquelas que são manipuladas e controladas durante o experimento. No contexto desse experimento, as formas de desenvolvimento, seja com ou sem o apoio da ferramenta Ruby-MDWA, são as variáveis independentes. As variáveis dependentes são aquelas que estão sob análise. Nesse contexto, as variáveis dependentes são produtividade e qualidade, além do tempo de desenvolvimento. Essas variáveis são medidas pelas métricas de produtividade e qualidade, propostas na Seção 7.2.

7.3.4 Critério de Separação dos Participantes

Os participantes do estudo foram selecionados por meio de amostragem não-probabilística por conveniência, ou seja, a probabilidade de todos os elementos da população pertencer à mesma amostra é desconhecida.

Para realizar os experimentos, foram selecionados quatro grupos de alunos, dois da turma de mestrado e dois da turma de graduação, respectivamente:

i) (G) G1: Grupo 1 da turma de graduação, formado por 8 participantes, de P1 até P8;

ii) (G) G2: Grupo 2 da turma de graduação, formado por 7 participantes, de P9 até P15;

iii) (M) G1: Grupo 1 da turma de mestrado, formado por 7 participantes, de P1 até P7;

iv) (M) G2: Grupo 2 da turma de mestrado, formado por 6 participantes, de P8 até P13;

Os participantes não possuíam conhecimento relevante sobre os assuntos envolvidos com a realização do experimento: MDD, Ruby, Ruby on Rails, MDWA e ferramenta Ruby-MDWA.

7.3.5 Projeto do Experimento

A distribuição dos participantes do experimento foi feita com base na resposta de um questionário distribuído, e apresentado no Apêndice A, para a identificação de conhecimento de cada um. As perguntas que os participantes responderam foram sobre sua experiência em linguagens de programação, frameworks, reuso de software e MDD, bem como a sua dificuldade em execução das atividades que envolveram o uso da ferramenta em questão e o desenvolvimento *ad-hoc* apresentado e as atividades sobre isso realizadas durante o treinamento.

A mesmo forma de realização do experimento foi aplicada às duas turmas diferentes que realizaram o experimento, pois cada turma conta com diferentes perfis de participantes. Os alunos de mestrado possuem experiência com desenvolvimento de software comercial e maior conhecimento em linguagens de programação, enquanto os alunos de graduação têm apenas experiência acadêmica no desenvolvimento de aplicações e uso de linguagens de programação.

Os participantes responderam onze perguntas para determinar o grau de conhecimento referente aos assuntos propostos, sendo três dissertativas e oito de alternativas identificadas por um número de 1 a 5 (sendo 1 – nenhum; 2 – básico; 3 – médio; 4 – avançado; 5 – especialista). Tanto a turma de mestrado, como a turma de graduação, foram divididas em dois grupos, da seguinte forma:

i) Mestrado – participaram 13 alunos distribuídos da seguinte forma: Grupo G1, com participantes de P1 até P7, e Grupo G2, com participantes de P8 até P13. A média de conhecimento dos participantes do G1 foi de 8,71 e dos participantes do G2 foi de 8,5. A média de conhecimento de todos os participantes da turma de mestrado foi de 8,615.

ii) Graduação – participaram 15 alunos distribuídos da seguinte forma: Grupo G1, com participantes de P1 até P8, e Grupo G2, com participantes de P9 até P15. A média de conhecimento dos participantes do G1 foi de 8,5 e dos participantes do G2 foi de 8,43. A média de conhecimento de todos os participantes da turma de graduação foi de 8,467.

As médias dos grupos são próximas entre si, indicando que os grupos são balanceados dentro de suas turmas. Os membros foram distribuídos nos grupos de acordo com o conhecimento que declararam, de forma que ficassem balanceados. Esse balanceamento refere-se ao numero de membros e o conhecimento de cada um. Nas Figuras Figura 7.1 e

Figura 7.2 estão apresentados o grau de conhecimento de cada participante da turma de mestrado e graduação (barras azuis), respectivamente, e a média da turma é representada pela linha vermelha.

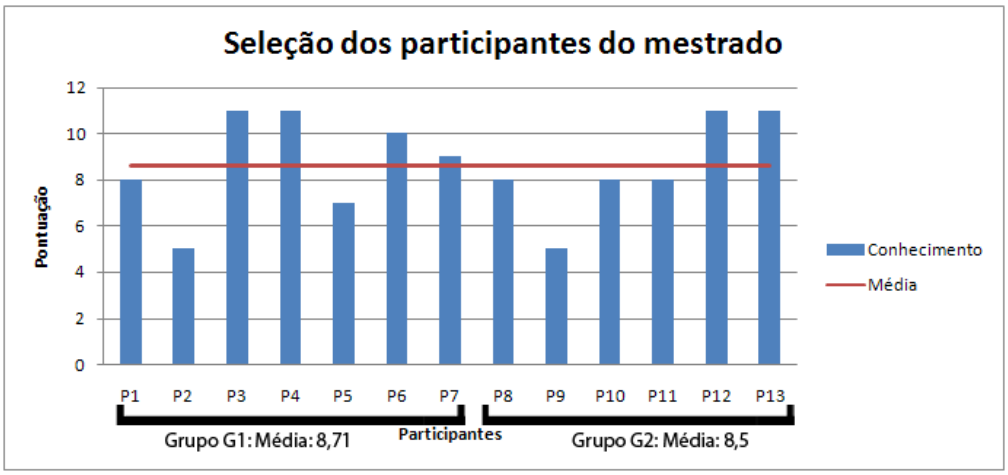


Figura 7.1: Conhecimento dos participantes da turma de mestrado.

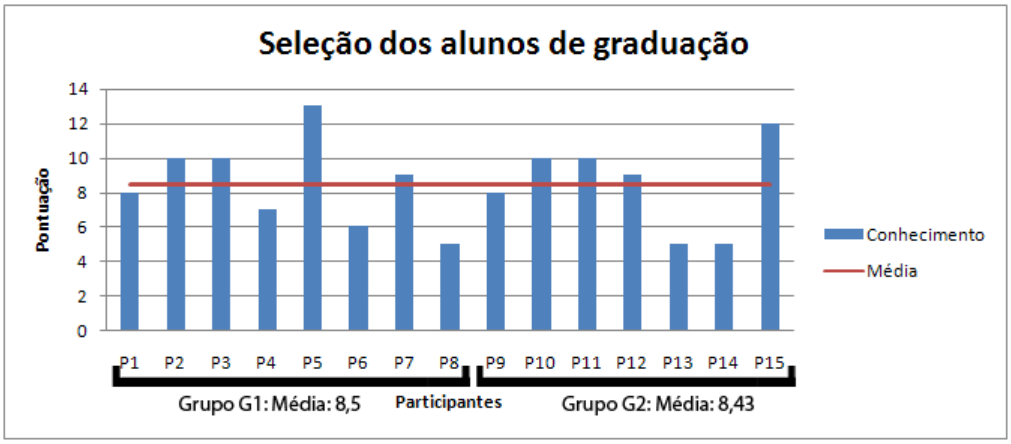


Figura 7.2: Conhecimento dos participantes da turma de graduação.

Os documentos utilizados no experimento foram:

i) Formulários de medição de tempo de desenvolvimento, preenchidos por cada participante para cada uma das aplicações desenvolvidas, descrito no Apêndice D. Neste formulário foi solicitado o nome do participante, o nome da aplicação desenvolvida, a forma de desenvolvimento realizada, com ou sem o apoio da ferramenta Ruby-MDWA, e um campo de observações para que o participante relate quaisquer considerações que ocorreram durante o desenvolvimento realizado no experimento;

ii) Especificação das duas aplicações a serem desenvolvidas. A Aplicação 1 foi referente a um sistema de gerenciamento de hotel e a Aplicação 2 foi referente a um sistema de locação de carros. As especificações das aplicações são apresentadas no Apêndice B e constaram de uma lista de requisitos e de um diagrama de classes UML;

iii) Um guia de referência com a explicação do fluxo de etapas para criação e transformação de modelos com apoio da ferramenta Ruby-MDWA, apresentado no Apêndice C, e similar a descrição apresentada na Seção 5.2.

7.4 Execução

O experimento realizado foi dividido em três fases: treinamento, piloto e execução, conforme apresentado na Tabela 7.3.

Tabela 7.3: Fases do experimento.

Fases	**Softwares Utilizados**	**Desenvolvimento *ad-hoc***	**Desenvolvimento apoiado por MDWA**
1ª Etapa: Treinamento	Sistema de Gerenciamento de Projetos	Todos os participantes	Todos os participantes
2ª Etapa: Piloto	Sistema de Vendas de Produtos	Todos os participantes	Todos os participantes
3ª Etapa: Execução	A1 – Sistema de Hotel; A2 – Sistema de Locadora de Carros.	***Mestrado:*** G1 – A1; G2 – A2. ***Graduação:*** G1 – A2; G2 – A1.	***Mestrado:*** G2 – A1; G1 – A2. ***Graduação:*** G2 – A2; G1 – A1.

Na primeira fase do experimento (Treinamento), cada turma recebeu treinamento de doze horas-aula, como forma de homogeneização do conhecimento dos participantes em relação à linguagem Ruby, o framework Ruby on Rails, MDD e a utilização da abordagem MDWA e a ferramenta Ruby-MDWA. Além disso, foram aplicados dois exercícios para verificação do conhecimento dos participantes e esclarecimento de dúvidas.

Na segunda fase (Piloto), os participantes receberam instruções de como o experimento seria conduzido e desenvolveram um software referente a um sistema de vendas de produtos, com e sem o apoio da ferramenta proposta.

Na terceira fase (Execução), em uma primeira etapa da turma de mestrado, o Grupo 1 desenvolveu a aplicação A1 de forma *ad-hoc*, enquanto o Grupo 2 desenvolveu a mesma aplicação A1 com apoio da ferramenta Ruby-MDWA. Na segunda etapa, o Grupo 1 desenvolveu a aplicação A2 com apoio da ferramenta Ruby-MDWA, enquanto o Grupo 2 desenvolveu a aplicação A2 de forma *ad-hoc*. Dessa forma, as duas implementações de cada aplicação são efetuadas por grupos diferentes, de maneira que a complexidade do software desenvolvido é irrelevante aos dados coletados. A turma de graduação efetuou o mesmo procedimento com a ordem diferente de aplicações. Na Tabela 7.3 é apresentada a distribuição de aplicações por grupo nas duas turmas, bem como sua forma de desenvolvimento, *ad-hoc* ou com o apoio da ferramenta.

7.5 Análise dos Resultados

Nesta Seção são apresentadas as informações coletadas após a realização do experimento nas duas turmas. Os dados correspondem aos fatores de qualidade de implementação de aplicações *ad-hoc* e com o apoio da ferramenta, além do tempo gastos por cada participante para cada software desenvolvido. Na Seção 7.5.1 são apresentados todos os dados coletados e na Seção 7.5.2 os dados são analisados. Na Seção 7.5.3 são apresentadas as ameaças à validade do experimento.

7.5.1 Dados Coletados

Os dados coletados no experimentopara cada turma, são apresentados em forma de tabelas com o desempenho de cada participante em cada uma das etapas de acordo com as métricas escolhidas e apresentadas definidas na Seção 7.2.

Para os alunos de Mestrado, os dados referentes a cada aplicação desenvolvida e a forma como foi realizada (com ou sem apoio de MDWA), estão exibidos na Tabela Tabela 7.4. Esses memos dados referentes aos alunos de Graduação estão apresentados na Tabela Tabela 7.6.

As informações individuais de cada participante de acordo com seus resultados que apresentaram referentes ao desenvolvimento *ad-hoc* e com apoio de MDWA, bem como seu grau de conhecimento e a diferença entre o desempenho dos desenvolvimentos realizados, estão apresentadas nas Tabelas Tabela 7.5 e Tabela 7.7, respectivamente, para a turma de Mestrado e de Graduação.

i) Mestrado

GRUPO 1 – AP 1 SEM MDWA									GRUPO 2 – AP 1 COM MDWA						
	P1	P2	P3	P4	P5	P6	P7	Média	P8	P9	P10	P11	P12	P13	Média
PMC	1	1	0,6	1	1	0,2	1	**0,8**	**0,8**	1	0,8	1	0,4	0,8	**0,8**
PCC	1	1	0,6	1	1	0,2	1	**0,8**	**0,8**	1	0,8	1	0,4	0,8	**0,8**
PVC	1	1	0,6	1	1	0,2	1	**0,8**	**0,8**	1	0,8	1	0,4	0,8	**0,8**
PRM	0,5	0,75	0,12	0,2	0,5	0	0	**0,271**	**0,37**	0,5	0,62	0,87	0,25	0,37	**0,5**
ILN	0	0	0	0	0	0	0	**0**	**0**	0	0	0	0	0	**0**
NE	1	0	0	1	1	0	1	**0,5**	**2**	1	0	0	0	1	**0,667**
QI	2,5	3,75	1,92	2,2	2,5	0,6	2	**2,217**	**0,78**	2,5	3,02	3,87	1,45	1,77	**2,233**
T (min)	54	65	65	65	65	65	65	**63**	**65**	65	61	65	57	65	**103**
Pr	2,78	3,46	1,77	2,03	2,31	0,55	1,85	**2,11**	0,72	2,31	2,97	3,57	1,53	1,63	**1,30**
GRUPO 1 – AP 2 COM MDWA									**GRUPO 2 – AP 2 SEM MDWA**						
	P1	P2	P3	P4	P5	P6	P7	Média	P8	P9	P10	P11	P12	P13	Média
PMC	1	1	1	1	1	1	0	**0,857**	**0**	0,4	0,6	1	0,6	0,6	**0,533**
PCC	1	1	1	1	1	1	0	**0,857**	**0**	0,4	0,6	1	0,6	0,6	**0,533**
PVC	1	1	1	1	1	1	0	**0,857**	**0**	0,4	0,6	1	0,6	0,6	**0,533**
PRM	0,87	0,87	0,87	0,75	0,87	0,87	0	**0,732**	**0**	0	0	0,5	0	0,25	**0,125**
ILN	1	0	0	1	0	0	0	**0,286**	**0**	0	0	0	0	0	**0**
NE	0	0	0	1	0	1	0	**0,286**	**0**	0	0	0	1	1	**0,333**
QI	4,87	3,87	3,87	3,75	3,87	2,87	0	**3,304**	**0**	1,2	1,8	3,5	0,8	1,05	**1,392**
T (min)	52	39	65	65	65	65	64	**59**	**62**	65	65	68	64	65	**64**
Pr	5,62	5,95	3,57	3,46	3,57	2,65	0,00	**3,36**	0,00	1,11	1,66	3,09	0,75	0,97	**1,31**

Tabela 7.4: Métricas das aplicações desenvolvidas pelos alunos da turma de mestrado.

	Desempenho individual dos alunos de mestrado					
		QI MDWA	QI *adhoc*	Diferença	Diferença (%)	Conhecimento
GRUPO 1	P1	4,875	2,500	2,375	48,7%	8
	P2	3,875	3,750	0,125	3,2%	5
	P3	3,875	1,925	1,950	50,3%	11
	P4	3,750	2,250	1,500	40,0%	11
	P5	3,875	2,500	1,375	35,5%	7
	P6	2,875	0,600	2,275	79,1%	10
	P7	0,000	2,000	-2,000	0,0%	9
GRUPO 2	P8	0,775	0,000	0,775	100,0%	8
	P9	2,500	1,200	1,300	52,0%	5
	P10	3,025	1,800	1,225	40,5%	8
	P11	3,875	3,500	0,375	9,7%	8
	P12	1,450	0,800	0,650	44,8%	11
	P13	1,775	1,050	0,725	40,8%	11
	Média	**2,810**	**1,837**	**0,973**	**34,6%**	**8,615**

Tabela 7.5: Desempenho individual dos participantes da turma de mestrado.

ii) Graduação

GRUPO 1 - AP 1 COM MDWA									
	P1	P2	P3	P4	P5	P6	P7	P8	Média
PMC	1	0,4	1	1	0	1	1	1	**0,8**
PCC	1	0,4	1	1	0	1	1	1	**0,8**
PVC	1	0,4	1	1	0	1	1	1	**0,8**
PRM	0,88	0,25	0,88	0,88	0	1	1	0,88	**0,72**
ILN	1	0	0	0	0	1	0	0	**0,25**
NE	0	0	0	0	0	0	0	0	**0**
QI	4,88	1,45	3,88	3,88	0	5	4	3,88	**3,37**
T (min)	58	61	48	61	60	62	48	49	**59**
Pr	5,05	1,43	4,85	3,82	0,00	4,84	5,00	4,75	**3,43**

GRUPO 2 - AP 1 SEM MDWA								
	P9	P10	P11	P12	P13	P14	P15	Média
PMC	0,4	0	0,8	1	1	0,4	0,6	**0,6**
PCC	0,4	0	0,8	1	1	0,4	0,6	**0,6**
PVC	0,4	0	0,8	1	1	0,4	0,6	**0,6**
PRM	0,13	0	0,13	0,25	0,75	0	0	**0,18**
ILN	0	0	0	0	1	0	0	**0,14**
NE	1	0	1	2	2	0	0	**0,86**
QI	0,33	0	1,53	1,25	2,75	1,2	1,8	**1,26**
T (min)	35	65	48	56	61	30	46	**48**
Pr	0,57	0,00	1,91	1,34	2,70	2,40	2,35	**1,58**

GRUPO 1 - AP2 SEM MDWA									
	P1	P2	P3	P4	P5	P6	P7	P8	Média
PMC	1	0	1	1	1	1	0,8	0,4	**0,78**
PCC	1	0	1	1	1	1	0,8	0,4	**0,78**
PVC	1	0	1	1	1	1	0,8	0,4	**0,78**
PRM	0,13	0	0,5	0,63	0	0,88	0,38	0	**0,31**
ILN	0	0	0	0	0	0	0	0	**0**
NE	1	0	2	2	2	0	1	1	**1,13**
QI	2,13	0	1,5	1,63	1	3,88	1,78	0,2	**1,51**
T (min)	53	55	58	61	60	60	65	57	**59**
Pr	2,41	0,00	1,55	1,60	1,00	3,88	1,64	0,21	**1,54**

GRUPO 2 - AP 2 COM MDWA								
	P9	P10	P11	P12	P13	P14	P15	Média
PMC	0,6	1	1	1	1	1	1	**0,94**
PCC	0,6	1	1	1	1	1	1	**0,94**
PVC	0,6	1	1	1	1	1	1	**0,94**
PRM	0,25	1	0,88	1	1	0,88	0,63	**0,8**
ILN	0	0	0	0	1	0	0	**0,14**
NE	1	0	0	0	0	0	1	**0,29**
QI	1,05	4	3,88	4	5	3,88	2,63	**3,49**
T (min)	42	48	65	63	50	50	62	**50**
Pr	1,50	5,00	3,58	3,81	6,00	4,66	2,55	**4,19**

Tabela 7.6: Métricas das aplicações desenvolvidas pelos alunos da turma de graduação.

	Desempenho Individual dos alunos de graduação					
		QI MDWA	QI *ad-hoc*	Diferença	Diferença (%)	Conhecimento
GRUPO 1	P1	4,88	2,13	2,75	56,41%	8
	P2	1,45	0,00	1,45	100,00%	10
	P3	3,88	1,50	2,38	61,29%	10
	P4	3,88	1,63	2,25	58,06%	7
	P5	0,00	1,00	-1,00	0,00%	13
	P6	5,00	3,88	1,13	22,50%	6
	P7	4,00	1,78	2,23	55,63%	9
	P8	3,88	0,20	3,68	94,84%	5
GRUPO 2	P9	1,05	0,33	0,73	69,05%	8
	P10	4	0,00	4,00	100,00%	10
	P11	3,875	1,53	2,35	60,65%	10
	P12	4	1,25	2,75	68,75%	9
	P13	5	2,75	2,25	45,00%	5
	P14	3,875	1,20	2,68	69,03%	5
	P15	2,625	1,80	0,83	31,43%	12
	Média	**3,43**	**1,40**	**2,03**	**59,22%**	**8,47**

Tabela 7.7: Desempenho individual dos participantes da turma de graduação.

7.5.2 Análise dos Resultados

De acordo com os resultados apresentados nas Tabelas Tabela 7.5 e Tabela 7.7, respectivamente, verifica-se que, independente do grau de conhecimento dos participantes, a maioria deles produziram aplicações de melhor qualidade com o apoio da ferramenta Ruby-MDWA em comparação ao desenvolvimento *ad-hoc*. Apenas dois participantes tiveram desempenho inferior nessa comparação. Os ganhos em qualidade foram, em média, 34,6% para alunos de mestrado e 59,22% para alunos de graduação. Essas constatações demonstram que as hipóteses H_{QI} e H_{Pr} podem ser validadas. Nos gráficos de barras das Figuras Figura 7.3 e Figura 7.4 são apresentados os gráficos de desempenho individual dos participantes com as métricas QI MDWA (azul escuro), QI *ad-hoc* (vermelho), Pr MDWA (roxo) e Pr *ad-hoc* (azul claro).

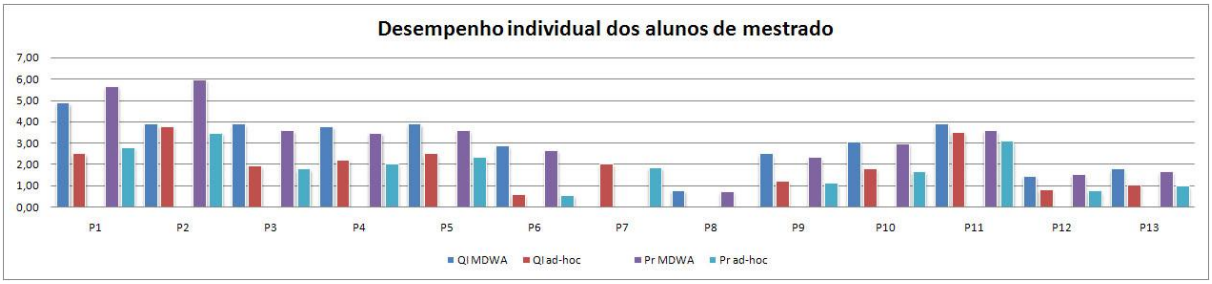


Figura 7.3: Desempenho individual dos alunos de mestrado na comparação entre QI MDWA e *ad-hoc* e Pr MDWA e *ad-hoc*.

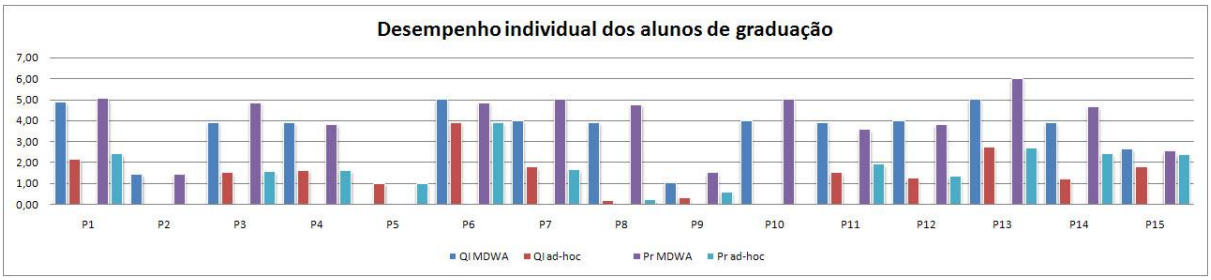


Figura 7.4: Desempenho individual dos alunos de graduação na comparação entre QI MDWA e *ad-hoc* e Pr MDWA e *ad-hoc*.

Considerando as versões das aplicações desenvolvidas de forma *ad-hoc* e com Ruby-MDWA, com base nas Tabelas Tabela 7.4 e Tabela 7.6, constata-se que, as métricas de qualidade (QI) e produtividade (Pr) possuem melhores índices quando apoiadas pela ferramenta. O desenvolvedor que utiliza a ferramenta Ruby-MDWA necessita especificar modelos e executar transformações, porém a quantidade de código escrito manualmente é

reduzida por evitar tarefas repetitivas. Dessa forma, o desenvolvedor pode se concentrar nas particularidades de negócio.

Nas Figuras Figura 7.5 e Figura 7.6 estão apresentados os resultados obtidos com a comparação de produtividade e qualidade, respectivamente, entre as aplicações desenvolvidas com e sem o suporte da ferramenta Ruby-MDWA. O símbolo (M) representa a turma de Mestrado e (G) representa a turma de Graduação, enquanto os símbolos A1 e A2 representam as aplicações desenvolvidas. As duas colunas mais à direita representam as médias de produtividade e qualidade para as aplicações desenvolvidas. A média de produtividade foi 86% maior na aplicação A1 e 90% na aplicação A2, comparando a versão desenvolvida com o apoio de Ruby-MDWA em relação à versão *ad-hoc*. A média de qualidade foi 61% maior na aplicação A1 e 134% na aplicação A1, na mesma comparação. Esses resultados evidenciam que a hipóteses H_{QI} e H_{Pr} podem ser comprovadas. Porém, é necessário que esse experimento seja replicado em outras populações para comprovar a validade dessa hipótese.

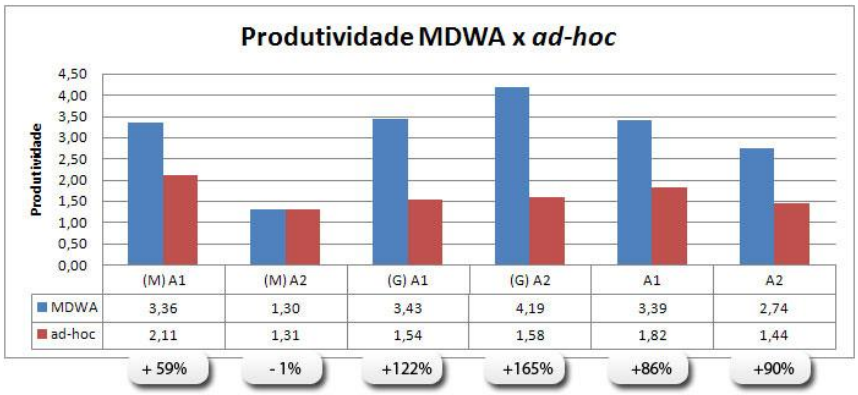


Figura 7.5: Comparação da métrica de produtividade (Pr) entre MDWA e *ad-hoc*, considerando as aplicações desenvolvidas.

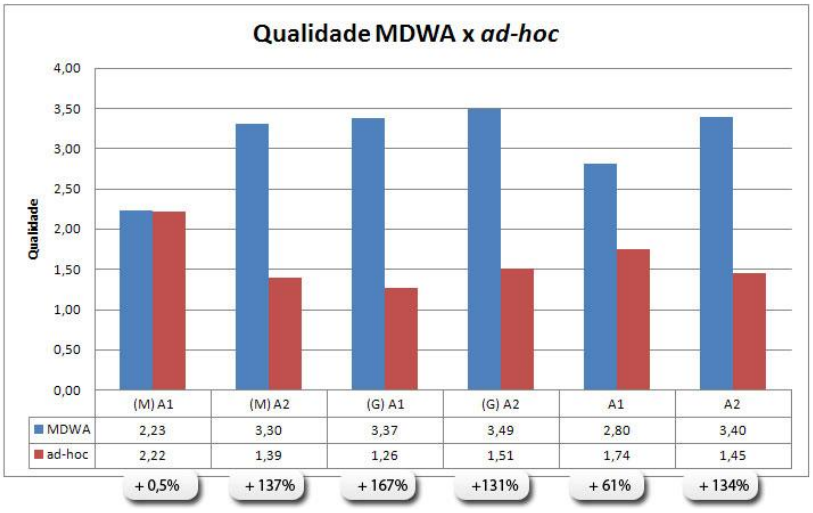


Figura 7.6: Comparação da métrica de qualidade (QI) entre MDWA e *ad-hoc*, considerando as aplicações desenvolvidas.

Além desses resultados, deve-se ressaltar que nas avaliações das implementações *ad-hoc* das aplicações A1 e A2, foi verificado apenas o caráter funcional das aplicações. Foram ignoradas as características geradas da infra-estrutura, como por exemplo, autenticação e controle de permissões de usuário e separações e formatações de leiaute. Essa decisão foi tomada, pois os participantes não teriam tempo hábil para implementação desses requisitos de forma *ad-hoc*.

7.5.3 Ameaças à Validade

Pelo fato do experimento ter sido aplicado a alunos de Graduação e Mestrado, e não a desenvolvedores de software, os resultados obtidos podem ter sido influenciados pela falta de experiência dos participantes com os conceitos utilizados. A amostra de participantes foi pequena, em comparação com o número de desenvolvedores de software, indicando que se o mesmo experimento for aplicado em outra amostra, os resultados podem ser diferentes, assim como ocorreu diferença entre as turmas testadas.

Apesar da realização de treinamentos de doze horas para cada turma, não existe total garantia de que os participantes possuem o mesmo nível de conhecimento em relação aos assuntos abordados. Como não houve nenhuma forma de incentivo aos participantes, que não o conhecimento adquirido, alguns participantes apresentaram resultados discrepantes e que implicaram no descarte de algumas amostras por falta de informações para processamento. Esses participantes não foram contabilizados no número de participantes. Esses fatores podem ter influenciado no resultado final.

Além disso, é necessário avaliar a abordagem MDWA e sua ferramenta de apoio em comparação com outras abordagens MDD, utilizando desenvolvedores com conhecimento na área, para medir quais os ganhos efetivos que este trabalho oferece ao desenvolvimento guiado por modelos. A comparação MDD *versus ad-hoc* tende a gerar bons resultados em favor do MDD, principalmente com o apoio de ferramentas de geração de código.

O experimento realizado baseou-se nessa comparação, pois o tempo de treinamento de alunos sem experiência nos assuntos que envolvem este trabalho é longo e não havia garantias de que os resultados seriam conclusivos.

7.6 Considerações finais

Neste Capítulo foi descrito um experimento realizado em duas turmas com a finalidade de medir as contribuições da ferramenta Ruby-MDWA a respeito de qualidade de software e produtividade de desenvolvimento. Para isso, os participantes desenvolveram aplicações Web de forma *ad-hoc* e com o apoio da ferramenta.

O resultado obtido evidencia que a ferramenta Ruby-MDWA pode melhorar a produtividade dos desenvolvedores de software Web, em virtude automatizar tarefas repetitivas e gerar código.

Considerando a qualidade, as aplicações construídas com o apoio da ferramenta possuem menor de taxa de erros de execução e maior índice de qualidade. Também implementam requisitos não-funcionais, como por exemplo, controle e autenticação de usuários, configurações para implantação e leiaute de apresentação finalizado.

8 Conclusões

8.1 Considerações iniciais

As abordagens MDD, em sua grande maioria, propõem soluções para automatizar a transformação entre modelos de alto nível de abstração e modelos geradores de código. Modelos gráficos de alto nível de abstração são ideais para visualização de diversos domínios, porém as transformações desses modelos não são triviais, como apresentado no Capítulo 1 desta dissertação.

A partir do contexto e da motivação também apresentados no Capítulo 1, foi desenvolvida uma abordagem com métodos simples de aplicação e independente de plataformas de programação que apóia o desenvolvimento orientado a modelos para softwares Web. Essa abordagem é denominada MDWA (*Model-Driven Web Applications*), e a ferramenta, denominada Ruby-MDWA, apóiam o desenvolvimento de aplicações Web guiadas por modelos e estimulam reuso de software.

A abordagem MDWA propõe um processo genérico que favorece a utilização de MDD, com base em um conjunto de atividades e modelos específicos de domínio, definidos pelos próprios desenvolvedores de software, e que pode ser associado com qualquer tecnologia Web.

Ela foi extraída a partir da construção da ferramenta Ruby-MDWA. Essa ferramenta fornece um conjunto de modelos, descritos em uma DSML textual, e transformadores M2M (*Model to Model*) e M2C (*Model to Code*), que são utilizados para a construção de aplicações Web com apoio do framework Ruby on Rails e da linguagem de programação Ruby. Dessa forma, o esforço de engenheiros de software é amenizado com a utilização de MDD para desenvolver aplicações Web, pois não necessitam definir modelos nem codificar seus transformadores.

As contribuições e limitações deste trabalho são descritas nas seções 8.2 e 8.3, respectivamente. Na Seção 8.4 são apontadas sugestões de trabalhos futuros que podem ser realizados como complementação deste.

8.2 Contribuições

Este trabalho propõe a utilização da abordagem MDWA para facilitar o desenvolvimento de aplicações Web guiadas por modelos. Seu conjunto de atividades é baseado em práticas presentes em modelos de processos tradicionais, facilitando a sua utilização por desenvolvedores. Seus modelos e artefatos devem ser definidos pelos engenheiros de software de forma que o domínio seja representado de forma completa.

O desenvolvimento de software com apoio da abordagem MDWA se inicia com a etapa de mapeamento de requisitos que tem a finalidade de encontrar e separar requisitos funcionais e não-funcionais e descrevê-los em modelos de alto nível, denominados modelos de requisitos. A identificação de requisitos pode ser efetuada com auxílio das práticas utilizadas tradicionalmente, como por exemplo, entrevistas ou criação de casos de uso.

Os desenvolvedores, posteriormente, devem identificar entidades, processos e personagens, gerando modelos de entidades, processos e personagens que representam a estrutura geral da aplicação. A partir da especificação desses modelos, os desenvolvedores podem construir ou reutilizar ferramentas de geração de código para a criação de protótipos. A utilização de protótipos facilita a obtenção de *feedbacks* do cliente, permitindo a definição de requisitos com maior clareza.

Em seguida, deve-se construir a infra-estrutura que define a arquitetura da aplicação e as demais particularidades envolvidas no desenvolvimento e implantação do software. Frameworks do tipo MIF oferecem a arquitetura e os demais recursos necessários. O reuso de software por meio de frameworks MIF reduz o tempo de construção do ambiente de infra-estrutura e aumenta a sua qualidade.

Na próxima etapa da abordagem, os desenvolvedores de software devem implementar as particularidades de negócio da aplicação ou do domínio utilizando *templates* de código, organizados segundo a arquitetura definida anteriormente. *Templates* de código oferecem interfaces para recuperação das informações descritas nos modelos, de forma que a tomada de decisões de desenvolvimento seja facilitada. *Templates* possibilitam reuso por meio da construção de componentes cujos comportamentos sejam definidos por modelos.

Por fim, os desenvolvedores devem construir ou utilizar um transformador de *templates* para código fonte com base na interpretação dos modelos descritos previamente. A

transformação de *templates* em código fonte produz como artefato, uma aplicação que atende os requisitos identificados no início do processo.

Atendendo aos pontos expostos nas seções 1.2 e 1.3, a abordagem MDWA une as práticas de modelos de processos tradicionais, conhecidas pelos desenvolvedores, com o desenvolvimento guiado por modelos, incentivando o reuso em todas as etapas do desenvolvimento, como forma de maximizar ganhos em produtividade e qualidade.

A ferramenta Ruby-MDWA foi construída com apoio do framework Ruby on Rails e da linguagem Ruby para complementar a abordagem MDWA. Essa ferramenta define metamodelos para cada modelo proposto pela abordagem MDWA e transformadores M2M e M2C, possibilitando que os desenvolvedores construam software guiados por modelos, mesmo que não desejem construir modelos e ferramentas de transformação específicas de domínio desde o início. Como a ferramenta é *open source*, seu código fonte pode ser customizado para descrever modelos e transformadores específicos de domínio.

A ferramenta Ruby-MDWA utiliza conceitos avançados da linguagem Ruby para criação de uma DSML textual interna que representa modelos de requisitos, entidades, processos e personagens, definidos pela abordagem MDWA.

O esforço de construção da ferramenta foi reduzido e possibilitou sua utilização independentemente do sistema operacional ou de ambientes de desenvolvimento, como Eclipse, por exemplo. Além disso, a curva de aprendizado por parte dos desenvolvedores é reduzida, pois é requerido somente o conhecimento da linguagem Ruby para a construção de modelos, *templates* e programação da lógica de negócio.

8.3 Limitações

Dentre as limitações deste trabalho pode-se citar:

i) A ferramenta Ruby-MDWA foi desenvolvida na linguagem Ruby com apoio do framework Ruby on Rails. Os transformadores entre modelos são implementados utilizando bibliotecas da linguagem e do framework. Como a linguagem Ruby é interpretada, a execução de código Ruby é lenta em comparação com linguagens compiladas. Com isso, as execuções das transformações M2M e M2C são lentas, especialmente em máquinas que utilizem o sistema operacional Windows.

ii) Os modelos utilizados pela ferramenta Ruby-MDWA são especificados em uma DSL interna (*embedded DSL*) que facilitam sua utilização por desenvolvedores com pouca experiência em MDD ou em Ruby-MDWA. Por outro lado, esses modelos possuem problemas quanto à expressividade e restrições para extensão, pois a gramática desses modelos depende da sintaxe da linguagem Ruby;

iii) Os *templates* de código utilizados pela ferramenta Ruby-MDWA são implementados com código ERB, uma biblioteca para a criação de *scripts* da linguagem Ruby. Com isso, desenvolvedores necessitam de apenas uma linguagem de programação para acessar os dados dos modelos e implementar a lógica de negócio. Por outro lado, como se trata de uma linguagem para duas aplicações diferentes, os *templates* de código da ferramenta são de difícil compreensão para desenvolvedores inexperientes.

iv) A criação de apenas um tema de apresentação das páginas geradas pela ferramenta Ruby-MDWA. Caso os desenvolvedores tenham interesse em customizar o leiaute de apresentação da aplicação, os arquivos de estilos CSS devem ser alterados manualmente, por não haver um método simples para trocar ou acrescentar estilos de apresentação à ferramenta.

8.4 Sugestões de trabalhos futuros

A abordagem MDWA e a ferramenta Ruby-MDWA podem ser utilizadas para apoiar o desenvolvimento de software guiado por modelos ou outros projetos de pesquisa. Nesta Seção, são apresentadas sugestões de trabalhos futuros que podem amenizar as restrições citadas anteriormente ou aumentar os recursos e a qualidade da abordagem e da ferramenta apresentadas.

i) Implementar ferramentas de apoio a abordagem MDWA em outras linguagens programação e frameworks: existem linguagens de programação e frameworks MIF mais populares que Ruby e Ruby on Rails, respectivamente. A construção de ferramentas que geram código baseado em outras linguagens e frameworks pode ser similar ao método de construção da ferramenta Ruby-MDWA e caracteriza uma forma de popularizar a abordagem MDWA.

ii) **Construir, com o apoio da ferramenta Ruby-MDWA, geradores de aplicações baseadas em linguagens de padrões:** a linguagem de padrões GRN (Braga, 2002), possui um framework de apoio, o GRENJ (Durelli, 2008) (Oliveira, 2010), e o gerador de aplicações GRENJ-Gens (Viana, 2009). Um gerador de aplicações baseado na ferramenta Ruby-MDWA, pode criar modelos de entidades, personagens e processos de acordo com a linguagem de padrões GRN, efetuando transformações posteriormente. O mesmo conceito pode ser aplicado com outras linguagens de padrões. Com isso pode-se avaliar os benefícios obtidos com a combinação entre a ferramenta Ruby-MDWA com linguagens de padrões.

iii) **Criar ou integrar os modelos textuais propostos pela ferramenta Ruby-MDWA com ferramentas de visualização:** a apresentação de modelos de forma visual pode facilitar a compreensão do domínio por parte de desenvolvedores e especialistas de domínio. Com o advento das bibliotecas de desenho e manipulação de vetores do HTML5 (W3C, 2012), é possível interpretar os modelos definidos pela ferramenta Ruby-MDWA e construir diagramas visualizados diretamente no navegador. Outra forma de representar modelos graficamente é efetuando a integração com o framework GMF da IDE Eclipse (Eclipse, 2012).

iv) **Modularizar temas e acrescentar componentes visuais à ferramenta Ruby-MDWA:** pode-se definir um mecanismo para criação e alteração de novos temas de leiaute de apresentação. Além disso, é possível criar componentes visuais, como por exemplo, máscaras para campos de telefone ou calendários para campos de datas, que seriam utilizados para customizar a apresentação de atributos e associações de entidades. Dessa forma a customização visual das aplicações construídas com apoio de Ruby-MDWA é facilitada.

v) **Ajustar a abordagem MDWA para criar aplicações Web com apoio de frameworks Javascript para interfaces ricas:** a utilização de frameworks Javascript, como por exemplo, AngularJS (AngularJS, 2012) e Backbone (BackboneJS, 2012), é uma forma de construir aplicações Web que fazem uso intensivo de AJAX (W3C, 2012). Esses frameworks tem se popularizado e podem mudar a forma como software Web é construído atualmente. É necessário verificar se é possível adaptar a abordagem MDWA para criar aplicações com apoio desses frameworks.

vi) **Adaptar a abordagem MDWA para outras plataformas de desenvolvimento:** o conjunto de etapas e modelos descritos pela abordagem MDWA também pode ser utilizado para apoiar o desenvolvimento guiado por modelos de softwares *desktop*, móveis e embarcados. As particularidades de cada uma dessas plataformas de desenvolvimento podem ser especificadas na abordagem MDWA para adaptar seu uso.

9 Referências bibliográficas

Abi-Antoun, M. (2007). “Making Frameworks Work: A Project Retrospective”. Conference on Object Oriented Programming Systems Languages and Applications, pp. 1004-1018.

Acerbis, R.; Bongio, A.; Brambilla, M.; Tisi, M.; Ceri, S.; Tosetti, E. (2007). “Developing eBusiness Solutions with a Model Driven Approach: The Case of Acer EMEA”. Industrial paper at 7th International Conference on Web Engineering (ICWE), pp. 539-544.

Armbrust, M.; Fox, A.; Griffith, R.; Joseph, A. D.; Katz, R.; Konwisnki, A.; Lee, G.; Patterson, D.; Rabkin, A.; Stoica, I.; Zaharia, M. (2010). “A view of cloud computing”. Communicatons of the ACM, vol. 53, issue 4, pp. 50-58.

AndroMDA (2012). “Generate Components Quickly with AndroMDA”. Disponível em: http://www.andromda.org/. Acessado em: 01/08/2012.

AngularJS (2012). “AngularJS – Javascript MVW Framework by Google”. Disponível em: http://angularjs.org/. Acessado: 01/07/2012.

BackboneJS (2012). “Backbone.JS – Middleware Javascript Framework”. Disponível em: http://backbonejs.org/. Acessado em: 25/10/2012.

Bachle, M.; Kirchberg, P. (2007) “Ruby on Rails”. Software, IEEE , vol.24, no.6, pp.105-108.

Baumer, D.; Riehle, D.; Siberski, W.; Wulf, M. (1997). “The Role Pattern Object”. Plop - Pattern Languages of Programs Conference. Disponível em: http://hillside.net/plop/plop97/Proceedings/riehle.pdf.

Bernardi, M.L.; Di Lucca, G.A.; Distante, D. (2011). “A model-driven approach for the fast prototyping of web applications”. Web Systems Evolution (WSE), 2011 13th IEEE International Symposium on , vol., no., pp.65-74, 30-30.

Beydeda, S.; Book, M.; Gruhn, V. (2005). “Model-Driven Software”. Springer-Verlag Berlin Heidelberg.

Braga, R. T. V. (2002). “Um Processo para Construção e Instanciação de Frameworks Baseados em uma Linguagem de Padrões para um Domínio Específico”. Tese de doutorado, ICMC/USP, São Carlos - SP.

Breitman, K. K.; Leite, J. C. S. P.; Finkelstein, A. (1999). “The world´s a stage: a survey on requirements engineering using a real-life case study.” J. Braz. Comp. Soc., Campinas, v. 6, n. 1.

Brugali, D.; Sycara, K. (2000). “Frameworks and pattern languages: a intriguing relationship”. Journal ACM Computing Surveys (CSUR), vol 32, issue 1.

Casteleyn, S.; Daniel, F.; Dolog, P.; Mattera, M. (2009). “Engineering Web Applications – Data-centric Systems and Applications”. Springer.

Castrejón, J.C.; López-Landa, R.; Lozano, R. (2011). “Model2Roo: A model driven approach for web application development based on the Eclipse Modeling Framework and Spring Roo”. Electrical Communications and Computers (CONIELECOMP), 2011 21st International Conference on , vol., no., pp.82-87.

Chelimsky, D.; Astels, A; Dennis, Z.; Hellesøy A.; Helmkamp, B.; North, D. (2010) “The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber and Friends”. 1ed. 448p. The Pragmatic Bookshelf.

Cirilo, C.; Prado, A.; Souza, W. L.; Zaina, L. A. (2010). “Model Driven RichUbi - Processo Dirigido a Modelos para a Construção de Interfaces Ricas de Aplicações Ubíquas Sensíveis ao Contexto”. SBES, Simpósio Brasileiro de Engenharia de Software.

Clements, P. e Northrop, L. M. (2001). “Software Product Lines: Practices and Patterns”. SEI Series in Software Engineering. Addison-Wesley Professional.

CLOC (2012). “Cloc – Count Lines of Code”. Disponível em: http://cloc.sourceforge.net/. Acessado em: 21/04/2012.

Coplien, J. (2004). “Patterns of engineering”. Potentials, IEEE , vol.23, no.2, pp. 4-8.

Costa, T.; Gomes, F.; Cagnin, M. I. (2007). “Estudo para Adaptação de um Processo Ágil de Desenvolvimento baseado em Framework para apoiar o Desenvolvimento de Software baseado em Modelos”. WDRA: Workshop em Desenvolvimento Rápido de Aplicações.

Daissaoui, A. (2010). “Applying the MDA approach for the automatic generation of an MVC2 web application”. Research Challenges in Information Science (RCIS), 2010 Fourth International Conference on , vol., no., pp.681-688, 19-21.

Deursen, A. v.; Klint, P.; Visser, J. (2000). “Domain-Specific Languages:An Annotated Bibliography”. ACM Sigplan Notices vol. 35 ed. 6. Pp 26-36.

Distante, D.; Pedone, P.; Rossi, G.; Canfora, G. (2007). “Model-Driven Development of Web Applications with UWA, MVC and JavaServer Faces”. Lecture Notes in Computer Science. vol. 4607/2007. pp. 457-472.

Durelli, V. H. S. (2008). “Reengenharia Iterativa do Framework GREN”. Dissertação de Mestrado. PPGCC – UFSCAR, São Carlos – SP.

Durski, R.; Spinola, M.; Burnett, R.; Reinehr, S. (2004). “Linhas de Produto de Software: riscos e vantagens de sua implantação”. VI Simpósio Internacional de Melhoria de Processos de Software.

Eclipse (2012). “Eclipse Modeling: M2T Project – JET”. Disponível em: http://www.eclipse.org/modeling/m2t/?project=jet#jet. Acesso em 25/05/2012.

Eickhof, C.; Geiger, N.; Hahn, M.; Zundorf, A. (2011). “Developing Enterprise Web Applications using the Story Driven Modeling approach”. 7th Model-Driven Web Engineering Workshop (MDWE).

Fayad, M. E. e Schmidt, D. C. (1997). “Object-Oriented Application Frameworks”. Communications of the ACM, vol. 40, no. 10, p. 32-38.

Fernandes, S. M.; Melnikoff, S. S. S. (2012). “A Catalog of modeling formalisms for domain-specific modeling languages design”. 9th International Conference on Information Systems and Technology Management (CONTECSI).

Fernandez, O. (2010). “The Rails 3 Way”. Addison-Wesley.

Fowler, M. (1997). “Analysis Patterns: Reusable Object Models”. Addison-Wesley.

Fowler, M. (2010). “Domain-Specific Languages”. Addison-Wesley.

France, R.; Rumpe, B. (2007). “Model-driven development of complex software: A research roadmap”. 29th International Conference on Software Engineering: IEEE Computer Society. p. 37–54.

Fu J.; Hao W.; Bastani, F.B.; Yen I. (2011). “Model-Driven Development: Where Does the Code Come From?”. Fifth IEEE International Conference on Semantic Computing (ICSC), vol., no., pp.255-262.

Gamma, E., Helm, R., Johnson, R. e Vlissides, J. (1995). “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley.

Geer, D. (2006). “Will Software Developers Ride Ruby on Rails to Success?”. IEEE Computer Society Press.

Goguen, J.A.; Linde, C. (1993). “Techniques for requirements elicitation”. Requirements Engineering, 1993., Proceedings of IEEE International Symposium on , vol., no., pp.152-164.

Gomaa, H. (2005). “Designing Software Product Lines with UML”. Addison-Wesley.

Greenfield, J.; Short, K.; Cook, S.; Kent, S.; Crupi, J. (2004) “Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools”. Wiley.

Gronback, R. C. (2009). “Eclipse Modeling Project – A Domain-Specific Language Toolkit”. Addison-Wesley.

Jaramillo, A.F. (2011). “Non-functional requirements elicitation from business process models”. Research Challenges in Information Science (RCIS), 2011 Fifth International Conference on , vol., no., pp.1-7.

Janzen, D.S.; Saiedian, H. (2008). “Does Test-Driven Development Really Improve Software Design Quality?”. Software, IEEE , vol.25, no.2, pp.77-84.

Java (2012). “Java Programming Language”. Disponível em: http://java.com. Acesso em: 02/11/2012.

Johnson, R. E. (1997a). “Frameworks = (Components + Patterns)”. Communications of the ACM, vol. 40, no. 10, p. 39-42.

Johnson, R. E. (1997b). “Components, Frameworks, Patterns”. In: 1997 Symposium on Software Reusability, ACM, p. 10-17.

JSON (2012). “Javascript Object Notation”. Disponível em: http://www.json.org. Acesso em: 27/07/2012.

Kelly, S.; Tolvanen, J. (2008). “Domain-Specific Modeling: Enabling Full Code Generation”. Wiley - IEEE Computer Society Press.

Kerry, E.; Delgado, S. (2009) “Applying software engineering practices to produce reliable, high-quality and accurate automated test systems”. AUTOTESTCON, 2009 IEEE , vol., no., pp.69-71.

Kleppe, A.;Warmer, J.; Bast, W. (2003). “MDA Explained: The Model Driven Architecture: Practice and Promise” , Addison-Wesley.

Koch, N. (2006). “Transformation Techniques in Model-Driven Development Processo of UWE”, International Conference on Web Engineering (ICWE).

Kraus, A.; Knapp, A., Koch, N. (2007). “Model-Driven Generation of Web Applications in UWE”, 3rd International Workshop on Model-Driven Web Enginnering (MDWE, 2007).

Kroiss, C.; Koch, N.; Knapp, A. (2009). “UWE4JSF: A Model-Driven Generation Approach for Web Applications”, 9th Internation Conference on Web Engineering (ICWE), 2009. pp. 493-496.

Kun Ma; Bo Yang. (2010). “A Hybrid Model Transformation Approach Based on J2EE Platform”. Education Technology and Computer Science (ETCS), 2010 Second International Workshop on , vol.3, no., pp.161-164.

Larman, C. (2004). “Utilizando UML e Padrões: Uma introdução à análise e ao projeto orientados a objetos e ao Processo Unificado”. 2a Ed. Bookman.

Ledeczi, A.; Bakay, A.; Maroti, M.; Volgyesi, P.; Nordstrom, G.; Sprinkle, J.; Karsai, G. (2001). “Composing domain-specific design environments”. Computer , vol.34, no.11, pp.44-51.

Lucrédio, D. (2009). “Uma Abordagem Orientada a Modelos para Reutilização de Software”. 287p. Tese (Doutorado em Ciências de Computação e Matemática Computacional) – ICMC-USP – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos-SP.

Luoma, J.; Kelly, S.; Tolvanen, J.P. (2004). “Defining Domain-Specific Modeling Languages: Collected Experiences”. 4th OOPSLA Workshop on Domain-Specific Modeling.

Medeiros, A. L. F. (2008). “MARISA-MDD: Uma Abordagem para Transformações entre Modelos Orientados a Aspectos: dos Requisitos ao Projeto Detalhado”. 112p. Tese (Mestrado em Sistemas e Computação) – Departamento de Informática e Matemática Aplicada, Universidade Federal de Rio Grande do Norte, Natal-RN.

Mellor, S. J.; Clark, A. N.; Futagami, T. (2003). “Model-driven development”. IEEE Software, v. 20, n. 5, p. 14–18.

Merilinna, J.; Parssinen, J. (2007). “Comparison Between Different Abstraction Level Programming: Experiment Definition and Initial Results”. OOPSLA Workshop on Domain-Specific Modeling, Montreal, Canada.

Mili, H.; Mili, F.; Mili, A. (1995). “Reusing Software: Issues and Research Directions”. IEEE Transactions on Software Engineering, vol 21, nº 6.

Mishra, D.; Mishra, A.; Yazici, A. (2008). “Successful requirement elicitation by combining requirement engineering techniques," Applications of Digital Information and Web Technologies. ICADIWT 2008. First International Conference on the , vol., no., pp.258-263.

MIT (2012). “The MIT Licensing”. Disponível em: http://opensource.org/licenses/mit-license.php. Acesso em: 21/04/2012.

Nagappan, N.; Ball, T. (2005). “Use of Relative Code Churn Measures to Predict System Defect Density”. ICSE ’05, International Conference on Software Engineering.

Obeo (2012a). “Acceleo”. Disponível em: http://obeo.fr/pages/acceleo/en. Acesso em: 12/05/2012.

Obeo (2012b). “ATL (Atlas Transformation Language)”. Disponível em: http://obeo.fr/pages/atl-pro/en. Acesso em: 12/05/2012.

Oliveira, A. L. (2010). “Modularização com Orientação a Aspectos de Frameworks Desenvolvidos com Linguagens de Padrões de Análise”. 178p. Tese (Mestrado em Ciência da Computação) – Departamento de Computação, Universidade Federal de São Carlos, São Carlos-SP.

(OMG) Object Management Group (2012a). “Documents Associated With Meta Object Model Facility (MOF) 2.0 Query/View/Transformation 1.1”. Disponível em: http://www.omg.org/spec/QVT/1.1/. Acesso em: 10/06/2012.

(OMG) Object Management Group (2012b). “MDA Guide Version 1.0.1. [S.l.], 2003”. Disponível em: http://www.omg.org/mda. Acesso em: 10/06/2012.

(OMG) Object Management Group (2012c). “Document Associated with UML Version 2.0, 2005”. Disponível em: http://www.omg.org/spec/UML/2.0/. Acesso em: 10/06/2012.

(OMG) Object Management Group (2012d). “BPMN - Business Process Model and Notation”. Disponível em: http://www.bpmn.org/. Acesso em: 11/06/2012.

Perl (2012). “Perl Programming Language”. Disponível em: http://www.perl.org/. Acesso em 01/11/2012.

Python (2012). “Python Programming Language”. Disponível em: http://www.python.org/. Acesso em: 04/11/2012.

Rahmouni, M.; Mbarki, S. (2011). “MDA-based ATL transformation to generate MVC2 Web models”, International Journal of Computer Science & Information Technology (IJCSIT) vol. 3, n. 4, p. 57-70.

Rails (2012). “Ruby on Rails Framework”. Disponível em: http://rubyonrails.org. Acessado em: 06/06/2012.

Riordan, R. M. (2008). “Head First Ajax”. 527p., O’Reilly Media.

Rivero, J. M.; Koch, N.; Grigera, J.; Rossi, G.; Luna, E. R. (2011). “Improving Agility in Model-Driven Web Engineering”. The 23rd International Conference on Advanced Information Systems Engineering (CAISE).

Ruby, S.; Thomas, D.; Hansson, D. H. (2011). “Agile Web Development With Rails”. 4ed. 480p. The Pragmatic Bookshelf.

Ruby (2012). “The Ruby Language”. Disponível em: http://ruby-lang.org. Acessado em: 06/06/2012.

RUP (2012). “IBM Rational Unified Process”. Disponível em: http://www-01.ibm.com/software/awdtools/rup/. Acessado em: 09/04/2012.

Schmidt, D. C. (2006). “Guest editor’s introduction: Model-driven engineering”. IEEE Computer, v. 39, n. 2, p. 25–31.

Schmidt, D. C., Fayad, M. e Johnson, R. E. (1996). “Software Patterns”. Communications of the ACM, vol. 39, no. 10, p. 37-39.

Selic, B. (2003) “The Pragmatics of Model-Driven Development”. IEEE Software.

Sendall, S.; Kozaczynski, W. (2003). “Model transformation: the heart and soul of model-driven software development”. Software, IEEE , vol.20, no.5, pp. 42- 45.

SmallTalk (2012). “SmallTalk Programming Language”. Disponível em: http://www.smalltalk.org/main/. Acesso em: 01/11/2012.

Solis, C.; Wang, X. (2011). “A Study of the Characteristics of Behaviour Driven Development”, Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on , vol., no., pp.383-387.

Stahl, T.; Volter, M.; Czarnecki, K. (2006). “Model-Driven Software Development: Technology, Engineering, Management”. John Wiley.

Steinberg, D.; Budinsky, F.; Paternostro, M.; Merks, E. (2008). “EMF Eclipse Modeling Framework”. 2ed. 722p. Addison-Wesley.

Struts (2012). “Apache Struts Framework”. Disponível em: http://struts.apache.org/. Acessado em: 26/10/2012.

Sugumaran, V., Park, S. e Kang, K. C. (2006). “Software Product Line Engineering”. Communications of the ACM, vol. 49, no. 12, p. 29-32.

Urrego-Giraldo, G. (2004). “Reasoning nonfunctional goals and features in Web systems”. Information and Communication Technologies: From Theory to Applications. Proceedings. 2004 International Conference on , vol., no., pp. 643- 644.

UWA (2002). “The UWA Approach to Modeling Ubiquitous Web Applications”. Disponível em: http://www.music.tuc.gr/Project.show?ID=5. Acesso em: 06/08/2012.

UWE (2012). “The UML-Based Web Engineering”. Disponível em: http://uwe.pst.ifi.lmu.de/. Acesso em: 08/08/2012.

Valim, J., (2011). “Crafting Rails Applications: Expert Practices for Everyday Rails Development”. 1ed. 184p. The Pragmatic Bookshelf.

Van Zul, P.; Kourie, D.; Boake, A. (2006). “Comparing the Performance of Object Databases and ORM Tools”. SAICSIT ’06 Proceedings.

Viana, M. C. (2009). “Construção da Camada de Interface Gráfica e de um Wizard para o Framework GRENJ”. 133p. Tese (Mestrado em Ciência da Computação) – Departamento de Computação, Universidade Federal de São Carlos, São Carlos-SP, 2009.

Viswanathan, V. (2008). “Rapid Web Application Development: A Ruby on Rails Tutorial”. Software, IEEE , vol.25, no.6, pp.98-106.

W3C. (2012) “World Wide Web Consortium”. Disponível em: http://www.w3c.org. Acessado em: 01/08/2012.

WebML (2012a). “The Web Modeling Language”. Disponível em: http://www.webml.org. Acessado em: 03/07/2012.

WebML (2012b). “WebML Papers”. Disponível em: http://www.webml.org/webml/page93.do?UserCtxParam=0&GroupCtxParam=0&ctx1=EN. Acesso em: 03/07/2012.

WebRatio (2012). “WebRatio Model-Driven Development”. Disponível em: http://www.webratio.com/. Acessado em: 02/07/2012.

Weiss, D. M. e Lai, C. T. R. (1999). “Software Product Line Engineering: A Family-Based Software Development Process”. Addison-Wesley.

YAML (2012). “YAML Ain’t Markup Language”. Disponível em: http://www.yaml.org/. Acesso em: 01/08/2012.

Yang Xiao-mei; Gu Ping; Dai Heng. (2009). “Mapping Approach for Model Transformation of MDA Based on XMI/XML Platform”. Education Technology and Computer Science. ETCS '09. First International Workshop on , vol.2, no., pp.1016-1019.

Ziadi, T.; Jézéquel, J.; Fondement, F. (2003). “Product line derivation with UML”. Proceeding Software Variability Management Workshop, University Of Groningen Department Of Mathematics And Computing Science, p. 94-102.

Zhuang, G.; Du, J. (2009) “MDA-based Modeling and Implementation of E-Commerce Web Applications in WebML”, Second International Workshop on Computer Science and Engineering (WCSE).

Apêndice A

Questionário de Conhecimento do Participantes do Experimento

Questionário para o experimento de MDWA

Metodologias de Desenvolvimento de Sistemas
Prof. Dra. Rosângela Aparecida Dellosso Penteado

Nome: _____ RA _____

Quais linguagens de programação que você conhece e qual o grau de conhecimento (1 menor, 5 maior)?

JAVA ( )1 ( ) 2 ( ) 3 ( ) 4 ( ) 5 Ruby ( )1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

C++ ( )1 ( ) 2 ( ) 3 ( ) 4 ( ) 5 Outra _____ ( )1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

Utilizou alguma forma de reuso? Qual?

_____

Grau de Reuso ( )1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

Você já utilizou frameworks? Quais?

_____
_____
_____
_____

Qual seu conhecimento sobre Ruby? ( )1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

Qual seu conhecimento sobre Ruby on Rails? ( )1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

Qual foi a dificuldade encontrada na programação usando Ruby e Ruby on Rails apresentada em aula? ( )1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

Qual foi a dificuldade encontrada no exercício de programação? ( )1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

Qual o grau de dificuldade que você teve para entender a abordagem MDWA?

( )1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

Você teve dificuldades para acompanhar o desenvolvimento utilizando MDD e a abordagem MDWA em aula? ( )1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

Você teve dificuldades com os exercícios envolvendo MDWA aplicados em aula?

( )1 ( ) 2 ( ) 3 ( ) 4 ( ) 5

Apêndice B

Especificações das Aplicações Desenvolvidas nos Experimentos

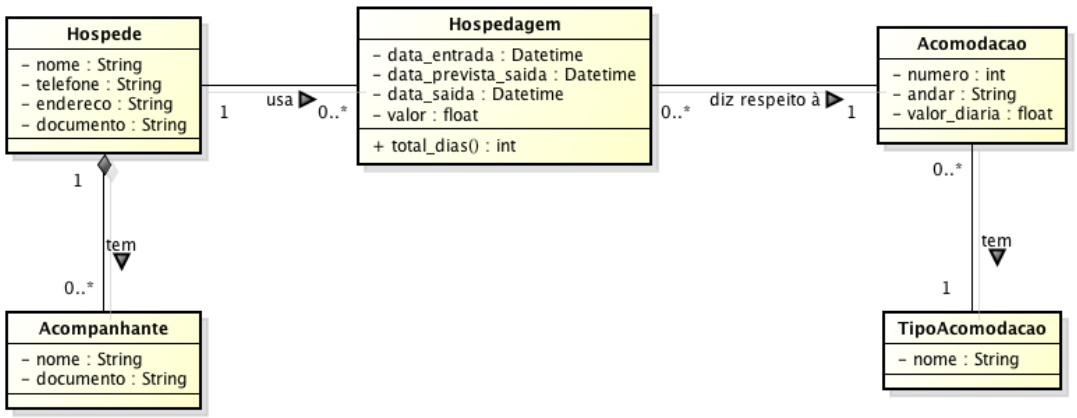
Experimento da abordagem MDWA

Aplicação 1 – Sistema de Gerenciamento de Hotel

Requisitos:

1. O sistema deve armazenar os dados do hóspede e de seus acompanhantes;
2. Cada hóspede pode se hospedar em uma acomodação. No caso do hóspede desejar mais acomodações, uma hospedagem deverá ser criada para cada uma das acomodações;
3. Cada acomodação deve ser categorizada de acordo com o seu tipo, como por exemplo, suíte presidencial ou quarto *deluxe*;
4. Uma diária se caracteriza por um ciclo de 24 horas de hospedagem. Se, por exemplo, o hóspede entrar na acomodação no dia 10/01 às 10:00, uma diária seria completa no dia 11/01 às 10:00. Se o hospede sair do quarto no dia 11/01 às 12:00, seriam caracterizadas duas diárias;
5. O valor da hospedagem deve ser calculado da seguinte forma: número de diárias x valor da diária.

Modelagem de classes UML:



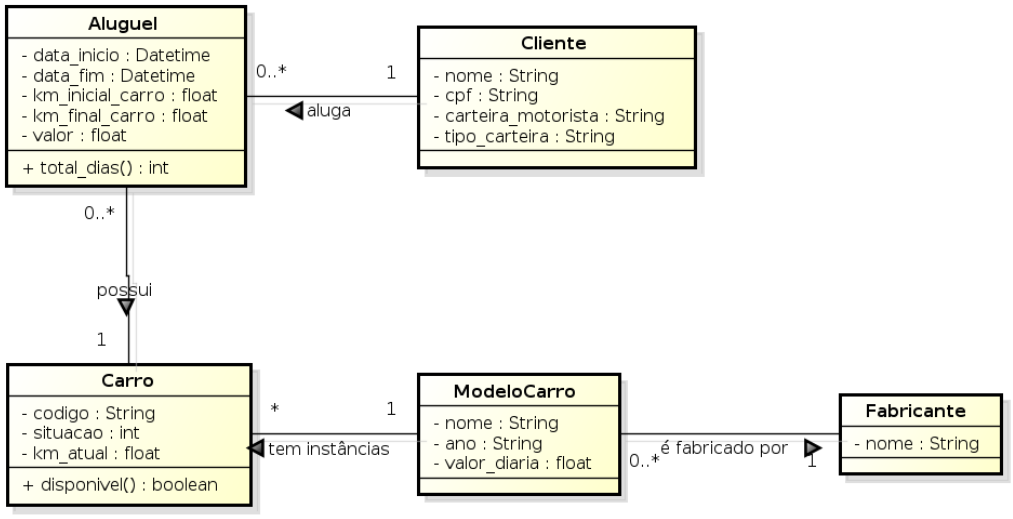
Experimento da abordagem MDWA

Aplicação 2 – Sistema de Locação de Carros

Requisitos:

1. O sistema deve armazenar os dados do cliente, principalmente informações sobre sua carteira de motorista;
2. Um cliente pode alugar somente um carro por vez;
3. Um aluguel possui horários de início e fim, além de armazenar as kilometragens inicial e final do veículo.
4. Um modelo de carro deve ser categorizado de acordo com seu fabricante;
5. Um modelo de carro pode gerar vários carros para aluguel. Todos os carros de um modelo possuem código, situação e kilometragem individuais;
6. Uma diária se caracteriza por um ciclo de 24 horas de aluguel do carro. Se, por exemplo, o cliente alugar um carro no dia 10/01 às 10:00, uma diária seria completa no dia 11/01 às 10:00. Se o hospede devolver o carro no dia 11/01 às 12:00, seriam caracterizadas duas diárias;
7. O valor da aluguel deve ser calculado da seguinte forma: número de diárias x valor da diária.

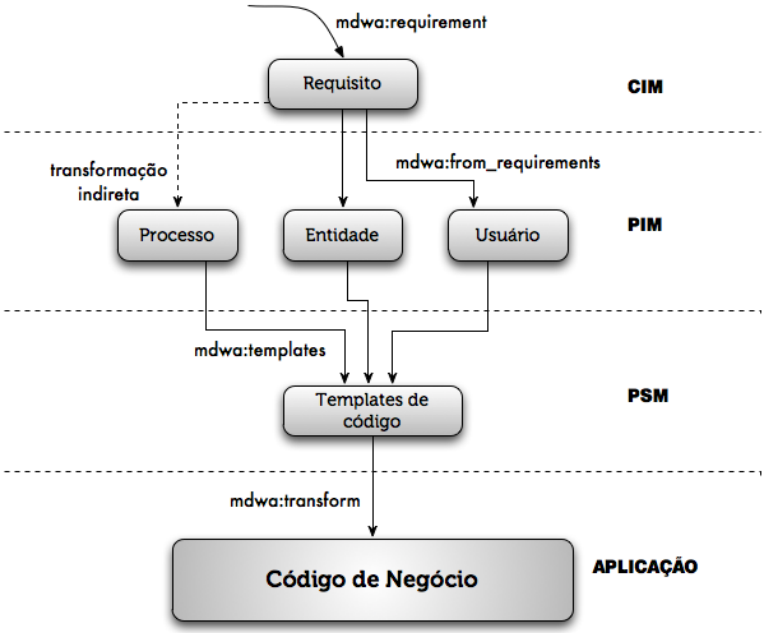
Modelagem de classes UML:



Apêndice C

Guia de Modelos e Transformações da Ferramenta Ruby MDWA

Guia de Referência da Ferramenta Ruby-MDWA



Comandos

`rails g mdwa:infrastructure`

Cria o ambiente de infra-estrutura, que possui separação de layouts, autenticação de usuários, login e estilização CSS.

`rails g mdwa:requirement "Resumo do requisito"`

Cria um requisito MDWA a partir de seu resumo.

`rails g mdwa:from_requirements [alias_req1 alias_req2 ...]`

Transforma os requisitos, passados como parâmetros, em entidades e usuários.

`rails g mdwa:templates [entidade1 entidade2 ...]`

Cria os templates das entidades passadas como parâmetros.

`rails g mdwa:transform [entidade1 entidade2 ...]`

Transforma os templates de código das entidades em código-fonte da aplicação. Recebe nomes de entidades como parâmetros, separados por espaço.

Apêndice D

Formulário de Registro de Tempo de Desenvolvimento

Formulário de Execução do Treinamento

Obs: este formulário deverá ser entregue preenchido ao final do experimento,

1. **Identificação do Participante**

Nome:			Data: ___/____/_____
	COM auxílio do MDWA		**SEM** auxílio MDWA

2. **Execução do Experimento**

Preenchido pelo Participante		Preenchido pelo Proponente do Experimento
Hora Início: ___ : ___	Hora Término: ___ : ___	**Tempo I (minutos):**

Interrupções

Preenchido pelo Participante		Preenchido pelo Proponente do Experimento
Hora Início: ___ : ___	Hora Término: ___ : ___	T1 (minutos):
Hora Início: ___ : ___	Hora Término: ___ : ___	T2 (minutos):
Hora Início: ___ : ___	Hora Término: ___ : ___	T3 (minutos):
Hora Início: ___ : ___	Hora Término: ___ : ___	T4 (minutos):
Hora Início: ___ : ___	Hora Término: ___ : ___	T5 (minutos):
TOTAL		**Tempo II (minutos):**

Tempo total do participante (Tempo I – Tempo II):	

3. **Observações**