Coleção UAB–UFSCar

Sistemas de Informação

Delano Medeiros Beder

Engenharia Web

uma abordagem sistemática para o desenvolvimento de aplicações web

Engenharia Web

uma abordagem sistemática para o desenvolvimento de aplicações web

Reitor
Targino de Araújo Filho
Vice-Reitor
Adilson J. A. de Oliveira
Pró-Reitora de Graduação
Claudia Raimundo Reyes

Secretária de Educação a Distância - SEaD
Aline M. de M. R. Reali
Coordenação SEaD-UFSCar
Daniel Mill
Denise Abreu-e-Lima
Glauber Lúcio Alves Santiago
Joice Otsuka
Marcia Rozenfeld G. de Oliveira
Sandra Abib
Vânia Paula de Almeida Neris
Coordenação UAB-UFSCar
Daniel Mill
Denise Abreu-e-Lima

Coordenadora do Curso de Sistemas de Informação
Vânia Neris

Conselho Editorial
José Eduardo dos Santos
José Renato Coury
Nivaldo Nale
Paulo Reali Nunes
Oswaldo Mário Serra Truzzi (Presidente)

Secretária Executiva
Fernanda do Nascimento

Diretor da EdUFSCar
Oswaldo Mário Serra Truzzi

UAB-UFSCar
Universidade Federal de São Carlos
Rodovia Washington Luís, km 235
13565-905 - São Carlos, SP, Brasil
Telefax (16) 3351-8420
www.uab.ufscar.br
uab@ufscar.br

EdUFSCar
Universidade Federal de São Carlos
Rodovia Washington Luís, km 235
13565-905 - São Carlos, SP, Brasil
Telefax (16) 3351-8137
www.editora.ufscar.br
edufscar@ufscar.br

Delano Medeiros Beder

Engenharia Web

uma abordagem sistemática para o desenvolvimento de aplicações web

São Carlos

EdUFSCar

2012

© 2012, Delano Medeiros Beder

Concepção Pedagógica
Daniel Mill

Supervisão
Douglas Henrique Perez Pino

Equipe de Revisão Linguística
Clarissa Galvão Bengtson
Daniel William Ferreira de Camargo
Daniela Silva Guanais Costa
Letícia Moreira Clares
Luciana Rugoni Sousa
Sara Naime Vidal Vital

Equipe de Editoração Eletrônica
Izis Cavalcanti

Equipe de Ilustração
Maria Julia Barbieri Mantoanelli

Capa e Projeto Gráfico
Luís Gustavo Sousa Sguissardi

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

B411e

Beder, Delano Medeiros.
Engenharia Web : uma abordagem sistemática para o desenvolvimento de aplicações web / Delano Medeiros Beder. -- São Carlos : EdUFSCar, 2012.
213 p. -- (Coleção UAB-UFSCar).

ISBN – 978-85-7600-290-1

1. Engenharia de software. 2. Web - desenvolvimento. I. Título.

CDD – 005.1 (20ª)
CDU – 681.32.06:62

Todos os direitos reservados. Nenhuma parte desta obra pode ser reproduzida ou transmitida por qualquer forma e/ou quaisquer meios (eletrônicos ou mecânicos, incluindo fotocópia e gravação) ou arquivada em qualquer sistema de banco de dados sem permissão escrita do titular do direito autoral.

SUMÁRIO

UNIDADE 3: Planejamento

UNIDADE 4: Modelagem

UNIDADE 6: Atividades guarda-chuva

UNIDADE 7: Metodologias ágeis x Engenharia Web

APRESENTAÇÃO

É indiscutível que a Web se tornou uma tecnologia essencial para negócios, comércio, educação, engenharia, entretenimento, finanças, governo, indústria, mídia, medicina, política, ciência e transporte, isso para citar apenas algumas áreas que têm impacto sobre nossa vida.

No entanto, à medida que aplicações web são integradas às estratégias de negócio, torna-se cada vez mais complexo o seu desenvolvimento. Apesar desse aumento de responsabilidade, o processo de Engenharia de Sistemas Web ainda não é tão disciplinado quanto a Engenharia de Software tradicional. Muitas aplicações Web continuam a ser construídas de uma maneira *ad hoc*, sem consideração com os princípios fundamentais da Engenharia de Software (ALTARAWNEH & SHIEK, 2008).

Dessa forma, é essencial que sistemas passem por um processo de engenharia (denominado Engenharia Web) de forma a construir e implantar uma solução eficaz e eficiente e que atenda às estratégias de negócio e às expectativas de seus usuários, pois como nos demais tipos de software, é necessário o completo entendimento do problema para se projetar uma solução eficaz que seja implementada e testada corretamente. É fundamental que erros de conteúdo, funcionalidade, usabilidade e segurança sejam identificados e corrigidos antes que o software entre em produção, pois o grande volume de aplicações Web equivalentes disponíveis faz com que o usuário possa não utilizar mais uma aplicação que já apresentou falhas/defeitos.

Nesse contexto, *Engenharia Web* foi escrito com o objetivo de lhe oferecer uma base sólida sobre a Engenharia Web, que consiste em uma abordagem sistemática e ágil para o desenvolvimento de aplicações Web. A agilidade implica em um enfoque de Engenharia de Software que incorpora ciclos de desenvolvimento rápidos. A abordagem sistemática implica a utilização de um **Arcabouço de Processo** que estabelece o alicerce para um processo de desenvolvimento completo, identificando um pequeno número de atividades que se aplicam a todos os sistemas, independentemente de tamanho e complexidade. No contexto deste livro, será apresentado um Arcabouço de Processo proposto por Pressman & Lowe (2009).

Finalizando a apresentação, o livro é composto de oito unidades, conforme descrito a seguir:

- A Unidade 1 apresenta as características inerentes a sistemas baseados na Web e os fundamentos da Engenharia Web, ressaltando dois conceitos muito importantes: agilidade e arcabouço de processo.

- As próximas 5 unidades (Unidade 2 à Unidade 6) discutem detalhadamente as atividades principais do Arcabouço de Processo proposto – comunicação, planejamento, modelagem, construção, implantação –, além das atividades guarda-chuva.
- A Unidade 7 discute em mais detalhes o relacionamento entre Engenharia Web e Agilidade de Processo.
- E, finalmente, a Unidade 8 apresenta o framework web Grails, o qual é inerente à filosofia ágil e poderia ser utilizado durante as atividades de construção e implantação do Arcabouço de Processo de Engenharia Web.

UNIDADE 1

Engenharia Web

1.1 Primeiras palavras

À medida que aplicações *web* são integradas às estratégias de negócio, torna-se cada vez mais complexo o seu desenvolvimento. Apesar desse aumento de responsabilidade, o processo de Engenharia de Sistemas Web ainda não é tão disciplinado quanto a Engenharia de Software tradicional. Muitas aplicações web continuam a ser construídas de uma maneira *ad hoc*, sem consideração com os princípios fundamentais da Engenharia de Software (ALTARAWNEH & SHIEK, 2008).

Esta unidade apresenta os fundamentos da Engenharia Web, que pode ser definida como a utilização de princípios, conceitos e métodos da Engenharia de Software de uma maneira que se adaptem às características inerentes de aplicações Web.

1.2 Problematizando o tema

Ao final desta unidade espera-se que o leitor seja capaz de reconhecer, distinguir e definir precisamente os conceitos básicos relacionados à Engenharia Web. Dessa forma, esta unidade pretende discutir as seguintes questões:

- Quais as principais características de sistemas baseados na Web?
- O que é Engenharia Web?
- Qual a correlação entre Engenharia de Software e Engenharia Web?
- Quais são as principais atividades do Arcabouço de Processo da Engenharia Web?

1.3 Aplicações Web

A Web tornou-se uma tecnologia indispensável para negócios, comércio, educação, engenharia, entretenimento, finanças, governo, indústria, mídia, medicina, política, ciência e transporte, isso para citar apenas algumas áreas que têm impacto sobre a nossa vida. No entanto, nem sempre foi assim. Os primeiros sítios Web eram bem simples e com impacto relativamente baixo em nossa sociedade.

Pressman (2006) descreve o histórico do desenvolvimento de aplicações Web da seguinte maneira:

> Nos primeiros dias da *World Wide Web* (entre 1990 e 1995), os sítios *web* eram formados de pouco mais do que um conjunto de arquivos de hipertexto

> ligados, que apresentavam informações usando texto e gráficos bem limitados. Com o passar do tempo, a Hypertext Markup Language (HTML) foi aumentada com ferramentas e tecnologias de desenvolvimento [por exemplo, Extensible Markup Language (XML), Java], que permitiram que engenheiros *Web* oferecessem capacidade de computação no lado do cliente e do servidor, juntamente com o conteúdo. Daí surgiu sistemas e aplicações baseados na *Web*. Hoje, as aplicações *Web* se tornaram ferramentas de computação sofisticadas, que não apenas oferecem funcionalidade isolada ao usuário final, mas também foram integradas a bancos de dados e aplicações corporativas e governamentais (PRESSMAN, 2006, p. 379).

Dessa forma, atualmente o desenvolvimento de aplicações Web é mais complexo do que nos primórdios da Internet, pois aplicações Web estão cada vez mais integradas às estratégias de negócio. No contexto deste livro, o termo *aplicação* Web abrange desde uma página Web simples (por exemplo, uma página pessoal em HTML) até um sítio Web abrangente (por exemplo, aplicações de comércio eletrônico, museus virtuais, *Internet Banking*, entre outros).

É importante ressaltar que uma aplicação Web é um software de computador no sentido que é uma coleção de instruções executáveis e dados que oferecem informações e funcionalidades para usuários finais. A implicação, portanto, é que é razoável esperar que o desenvolvimento de aplicações Web leve em consideração algumas ou todas as lições aprendidas durante as muitas décadas de desenvolvimento de software convencional. Esse tópico será discutido mais adiante nesta unidade.

Sendo uma aplicação Web um software de computador, é válido questionar se há diferenças entre sistemas baseados na Web e sistemas de software convencionais. Ou seja, se as características de uma aplicação Web são diferentes das características de um software convencional. Pressman & Lowe (2009) apresentam algumas características de aplicações Web que, quando consideradas em sua totalidade, diferenciam os sistemas baseados na Web dos sistemas mais convencionais, baseados em computador. As características (Figura 1) a seguir são encontradas na grande maioria das aplicações Web (PRESSMAN & LOWE, 2009):

- **Concentração na rede**. Cada aplicação Web está disponível no contexto de uma rede de computadores e deve servir às necessidades de uma comunidade diversificada de clientes. Esse contexto pode ser: a Internet (comunicação aberta ao mundo todo), a Intranet (comunicação restrita ao escopo de uma organização) ou a Extranet (comunicação entre redes).
- **Concorrência**. Um grande número de usuários poderá acessar a aplicação Web ao mesmo tempo. Em muitos casos, os padrões de acesso

variarão bastante. Em alguns casos, as ações de um usuário, ou um conjunto destes, podem ter um impacto sobre as ações ou informações apresentadas a outros usuários.

- **Carga imprevisível**. O número de usuários da aplicação Web poderá variar por ordens de grandeza de um dia para outro.
- **Desempenho**. Se um usuário da aplicação Web tiver que esperar muito tempo para acesso, para processamento no lado do servidor, para formatação e exibição no lado do cliente, ele poderá desistir de utilizar a aplicação.
- **Disponibilidade**. Embora uma disponibilidade de 100% seja pouco razoável, os usuários de aplicações Web populares normalmente exigem acesso 24 horas por dia, 7 dias por semana, 365 dias por ano ("24/7/365").
- **Orientada a dados**. A função principal de muitas aplicações Web é usar hipermídia para apresentar conteúdo de texto, gráficos, áudio e vídeo ao usuário final. Além disso, as aplicações Web normalmente são usadas para acessar informações que existem em bancos de dados legados (por exemplo, aplicações de comércio eletrônico ou financeiras).
- **Sensível ao conteúdo**. A qualidade e a natureza estética do conteúdo continuam sendo um determinante importante da qualidade de uma aplicação Web.
- **Evolução contínua**. Diferentemente do software de aplicação convencional, que evolui por uma série de versões planejadas, cronologicamente espaçadas, aplicações Web evoluem continuamente. Não é raro que algumas aplicações Web (especificamente, seu conteúdo) sejam atualizadas minuto a minuto ou por conteúdo a ser dinamicamente produzido a cada solicitação.
- **Urgência**. Embora a urgência, a necessidade imprescindível de colocar o software no mercado rapidamente, seja uma característica de muitos domínios de aplicação, as aplicações Web normalmente exibem um tempo para o mercado que pode ser uma questão de alguns dias ou semanas. Os engenheiros Web deverão usar métodos para planejamento, análise, projeto, implantação e testes que têm sido adaptados a cronogramas reduzidos, exigidos para o desenvolvimento da aplicação Web.
- **Segurança**. Como as aplicações Web estão disponíveis por acesso à rede, é difícil, se não impossível, limitar a população de usuários finais que podem acessar a aplicação. Para proteger conteúdo confidencial e

oferecer modos seguros de transmissão de dados, medidas de segurança fortes precisam ser implementadas por meio da infraestrutura que apoia a aplicação Web e na aplicação propriamente dita.

- **Estética**. Uma parte inegável do apelo de uma aplicação Web é sua aparência. Quando uma aplicação tiver sido criada para comercializar ou vender produtos ou ideias, ou fornecer serviços que geram receita, a estética pode estar relacionada tanto ao sucesso quanto ao projeto técnico.

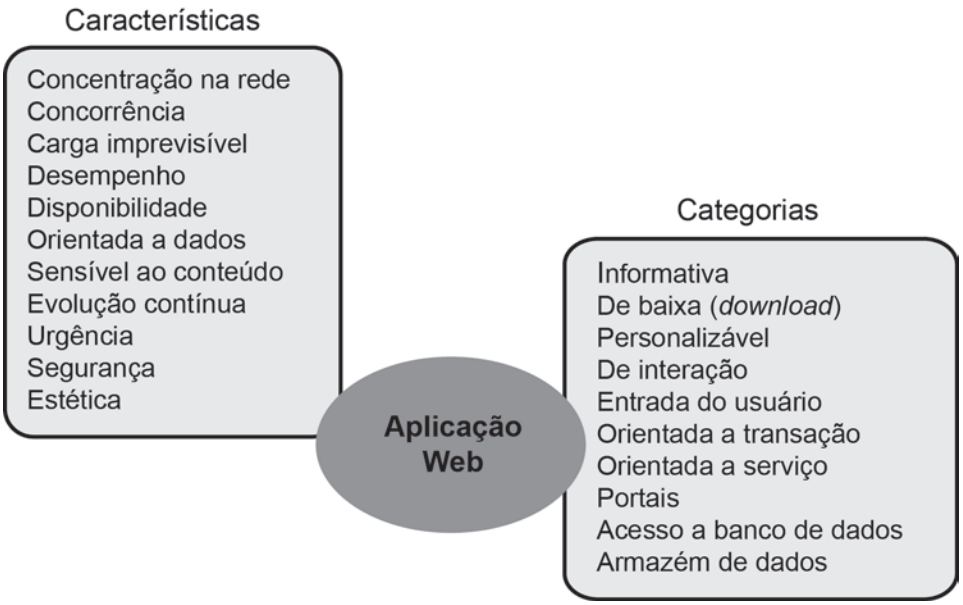


Figura 1 Características e categorias de aplicações Web.

Quanto aos problemas que aplicações Web tratam, as seguintes categorias (Figura 1) de aplicação Web são as mais comumente encontradas (PRESSMAN, 2006):

- **Aplicações informativas** – conteúdo somente de leitura é fornecido com navegação e ligações (links) simples.
- **Aplicações de baixa (*download)*** – um usuário baixa informação (faz *download*) de um servidor adequado.
- **Aplicações personalizáveis** – o usuário adapta o conteúdo a necessidades específicas.
- **Aplicações de interação** – a comunicação entre uma comunidade de usuários ocorre por intermédio de salas de bate-papo, quadros de aviso ou mensagens instantâneas.
- **Aplicações de entrada do usuário** – entrada baseada em formulários é o principal mecanismo para comunicar o propósito do usuário final.

- **Aplicações orientadas a transação** – o usuário faz uma solicitação (por exemplo, coloca um pedido) que é atendida pela aplicação Web.
- **Aplicações orientadas a serviços** – a aplicação fornece um serviço ao usuário (por exemplo, ajuda o usuário a calcular o valor da parcela do pagamento de um empréstimo).
- **Portais** – a aplicação direciona o usuário para outros conteúdos ou serviços da Web fora do domínio de aplicação do portal.
- **Acesso a banco de dados** – o usuário consulta uma grande base de dados e extrai informação.
- **Armazém de dados (*data warehouse*)** – o usuário consulta uma coleção de grandes bancos de dados e extrai informação.

É importante salientar que uma aplicação Web pode se enquadrar em mais de uma categoria, bem como trocar de categoria ao longo de seu tempo de vida. Por exemplo, considere uma organização multinacional que inicialmente decidiu realizar um baixo investimento no desenvolvimento de sistemas baseados na Web e apenas desenvolveu uma *aplicação informativa* (página principal[1] informativa da organização). No entanto, ao longo dos anos e à medida que estratégias de negócio foram integradas, essa aplicação Web evoluiu e tornou-se, por exemplo, um *armazém de dados*, em que as bases de dados dessa organização, dispostas em diferentes países, são acessadas.

1.4 Engenharia Web

Conforme discutido, atualmente o desenvolvimento de aplicações Web é mais complexo do que nos primórdios da Internet, pois elas estão cada vez mais integradas às estratégias de negócio das organizações. Apesar desse aumento de responsabilidade, o processo de Engenharia de Sistemas Web ainda não é tão disciplinado quanto a Engenharia de Software tradicional. Muitas aplicações Web continuam a ser construídas de uma maneira *ad hoc*, sem consideração com os princípios fundamentais da Engenharia de Software (ALTARAWNEH & SHIEKH, 2008).

Dessa forma, sistemas Web devem passar por um processo de engenharia (denominado Engenharia Web) de forma a construir e implantar uma solução eficaz e eficiente, que atenda às estratégias de negócio e às expectativas de seus usuários, pois como nos demais tipos de software, é necessário o completo entendimento do problema para se projetar uma solução eficaz que seja

1 Em inglês: *home page*.

implementada e testada corretamente. É fundamental que erros de conteúdo, funcionalidade, usabilidade e segurança sejam identificados e corrigidos antes que o software entre em produção, pois o grande volume de aplicações Web equivalentes disponíveis faz com que o usuário possa não utilizar mais uma aplicação que já apresentou falhas/defeitos.

Nesse sentido, Pressman (1998) discute o acoplamento da Engenharia de Software e Engenharia Web da seguinte maneira:

> Me parece que praticamente qualquer produto ou sistema importante merece engenharia. Antes que você comece a construí-lo, é melhor entender o problema, projetar uma solução funcional, implementá-la de uma forma sólida e testá-la totalmente. Você provavelmente também deve controlar as mudanças enquanto trabalha e ter algum mecanismo para garantir a qualidade do resultado final. Muitos desenvolvedores Web não discutem isso; eles simplesmente acham que seu mundo é realmente diferente e que as técnicas convencionais de engenharia de software simplesmente não se aplicam (PRESSMAN, 1998, p. 1).

O que se pode extrair dessa discussão é que os princípios, os conceitos e os métodos da Engenharia de Software podem ser aplicados ao desenvolvimento Web, mas sua aplicação requer uma técnica um tanto diferente do seu uso durante o desenvolvimento de sistemas de software convencionais.

Dessa forma, a Engenharia Web pode ser definida como a utilização de princípios, conceitos e métodos da Engenharia de Software, de uma maneira que se adaptem às características inerentes de aplicações Web, tais como Desempenho, Disponibilidade, Evolução Contínua, Urgência e Segurança (discutidas na Seção 1.3).

A Engenharia Web propõe um arcabouço de processo ágil, porém disciplinado, para a construção de sistemas Web de qualidade industrial (PRESSMAN & LOWE, 2009). Para melhor entender o que é Engenharia Web, dois conceitos-chaves precisam ser perfeitamente compreendidos: *Agilidade* e *Arcabouço de Processo*. Esses conceitos serão discutidos detalhadamente durante este livro.

1.4.1 Agilidade de processo

Agilidade de processo implica um enfoque de Engenharia de Software enxuto que incorpora ciclos de desenvolvimentos rápidos. E cada ciclo resulta na implantação de um incremento da aplicação Web. Isto é, o desenvolvimento da aplicação Web é realizado por meio da entrega de uma série de versões chamadas de incrementos, que fornecem progressivamente mais funcionalidade para os clientes à medida que cada incremento é entregue.

Metodologias ágeis (discutidas na Unidade 7) como *Extreme Programming* (XP) e *Scrum* são bons exemplos de modelos de processo que aplicam o conceito de agilidade de processo no desenvolvimento de sistemas de software.

1.4.2 Arcabouço de processo

O segundo conceito a ser discutido é arcabouço de processo, que estabelece o alicerce para um processo de desenvolvimento completo, identificando um pequeno número de atividades que se aplicam a todos os sistemas, independentemente de tamanho e complexidade.

Neste livro, um arcabouço genérico de processo de Engenharia Web, proposto por Pressman & Lowe (2009), será apresentado. Antes que o Arcabouço de Processo seja apresentado, é importante salientar alguns aspectos presentes na maioria dos projetos de desenvolvimento de aplicações Web, e que foram considerados na concepção do arcabouço proposto (PRESSMAN & LOWE, 2009):

- Requisitos evoluem com o tempo. Quando você inicia um projeto de desenvolvimento Web, pode haver incerteza sobre alguns elementos da estratégia de negócios, do conteúdo e da funcionalidade a ser entregue, questões de interoperabilidade e muitas outras facetas do problema.
- As mudanças ocorrem com frequência. Como a incerteza é uma parte inerente da maioria dos projetos de desenvolvimento Web, as mudanças nos requisitos são comuns. Além disso, o *feedback* do usuário (baseado na avaliação dos incrementos entregues) e alterações nas condições de negócios podem ocasionar mudanças.
- Linhas de tempo são curtas. Isso alivia a criação de documentação de engenharia volumosa, mas não impede a necessidade de documentar minimamente: (1) a análise do problema, (2) o projeto e (3) o teste.

Levando em consideração esses aspectos, o arcabouço proposto sugere o emprego de uma abordagem incremental. Ou seja, atividades de arcabouço ocorrerão repetidamente à medida que cada incremento de aplicação Web for desenvolvido e entregue.

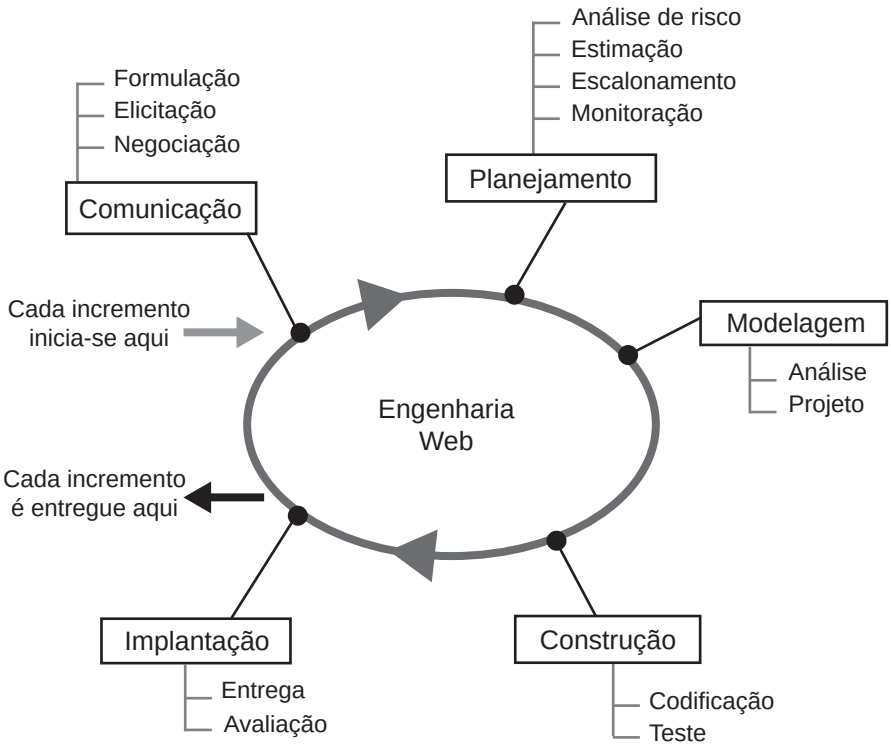


Figura 2 Arcabouço de Processo da Engenharia Web.

O arcabouço proposto (Figura 2) contém cinco atividades principais (PRESSMAN & LOWE, 2009):

- A atividade de **comunicação** (Unidade 2) envolve interação e colaboração intensas com o cliente (e outros interessados),[2] e abrange o levantamento de requisitos e outras atividades relacionadas.
- A atividade de **planejamento** (Unidade 3) estabelece um plano incremental para o trabalho de desenvolvimento.
- A atividade de **modelagem** (Unidade 4) abrange a criação de modelos (por exemplo, diagramas de classes) que auxiliam o processo de desenvolvimento.
- A atividade de **construção** (Unidade 5) combina a geração de código e o teste necessário para revelar erros no código, enquanto a atividade de **implantação** (Unidade 5) consiste na entrega de um incremento ao cliente, que o avalia e oferece *feedback* com base nessa avaliação.

Além dessas cinco atividades principais, será também discutida uma coleção de atividades guarda-chuva (Unidade 6), que ocorrem em segundo plano. Como essas atividades são igualmente importantes, uma equipe de desenvolvimento

2 Em inglês: *stakeholders*.

deverá considerá-las explicitamente. Entre atividades guarda-chuva pode-se citar: gerência de mudança, gerência de qualidade, gerência de riscos e gerência de projeto.

1.5 Considerações finais

Esta unidade apresentou as características inerentes a sistemas baseados na Web e os fundamentos da Engenharia Web, ressaltando dois conceitos muito importantes: agilidade e arcabouço de processo.

Como discutido, um arcabouço de processo estabelece o alicerce para um processo de desenvolvimento completo, identificando um pequeno número de atividades que se aplicam a todos os sistemas, independentemente de tamanho e complexidade. No contexto deste livro, será apresentado um Arcabouço de Processo proposto por Pressman & Lowe (2009).

As próximas unidades (Unidade 2 à Unidade 6) discutem mais detalhadamente as atividades principais do arcabouço de processo proposto – comunicação, planejamento, modelagem, construção, implantação –, além das atividades guarda-chuva.

O segundo conceito, agilidade de processo, implica um enfoque de Engenharia de Software que incorpora ciclos rápidos de desenvolvimento. A Unidade 7 discute com mais detalhes o relacionamento entre Engenharia Web e agilidade de processo.

1.6 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

PRESSMAN, R. S. Engenharia da Web. In: _____. *Engenharia de Software*. 6. ed. São Paulo: McGraw-Hill, 2006.

PRESSMAN, R. S.; LOWE, D. Sistemas Baseados na Web. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

_____. Engenharia Web. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

_____. Processo de Engenharia Web. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

UNIDADE 2

Comunicação

2.1 Primeiras palavras

Na unidade anterior foi apresentado o Arcabouço de Processo de Engenharia Web (Figura 2), que consiste em algumas atividades ocorridas repetidamente à medida que cada incremento de aplicação Web é desenvolvido e entregue.

Esta unidade apresenta a atividade de *comunicação*, que envolve interação e colaboração intensas com o cliente (e outros interessados), e abrange o levantamento de requisitos e outras atividades relacionadas.

2.2 Problematizando o tema

Ao final desta unidade espera-se que o leitor seja capaz de reconhecer e distinguir precisamente os conceitos básicos relacionados à atividade de comunicação do Arcabouço de Processo da Engenharia Web. Dessa forma, esta unidade pretende discutir as seguintes questões:

- Quais são os objetivos da atividade de comunicação do Arcabouço de Processo de Engenharia Web?
- Quais são as principais ações realizadas durante a atividade de comunicação?

2.3 Comunicação

Entre os princípios mais fundamentais da Engenharia de Software está: “entenda o problema antes de começar a resolvê-lo e certifique-se de que a solução que você concebe é aquela que as pessoas realmente desejam” (PRESSMAN, 2006, p. 389). Isto é, o completo entendimento do problema é fundamental para que se possa projetar uma solução eficaz que seja implementada e testada corretamente. É razoável afirmar que esse princípio também é um dos pilares da Engenharia Web.

A atividade de comunicação (Figura 3) oferece à equipe de Engenharia Web um modo organizado de levantar requisitos dos interessados e é caracterizada por três ações: formulação, elicitação e negociação (PRESSMAN & LOWE, 2009).

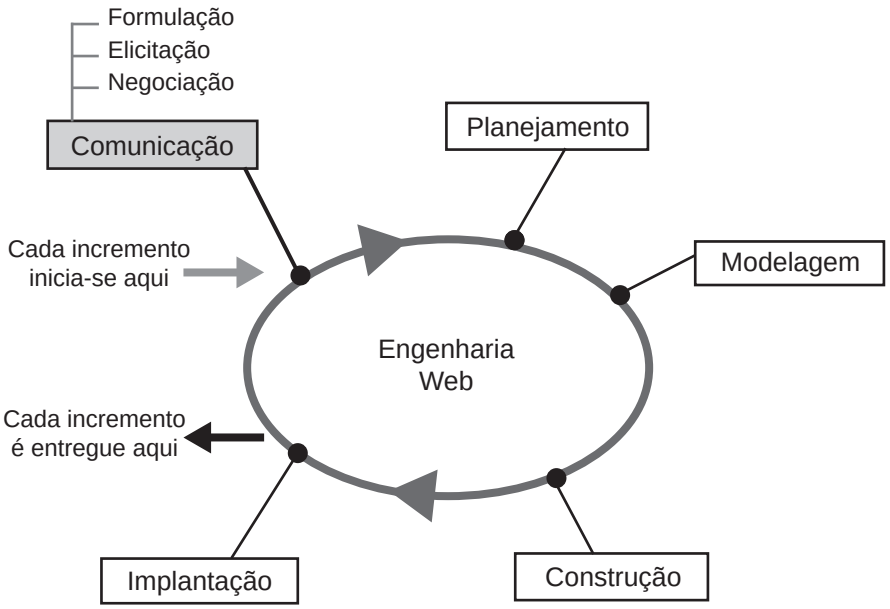


Figura 3 Atividade de comunicação no fluxo de processo de Engenharia Web.

A *formulação* define o contexto organizacional e de negócios para a aplicação Web. Além disso, os interessados são identificados; mudanças ou requisitos em potencial no ambiente da organização são previstos; e a integração entre a aplicação Web e outras aplicações de negócios, bancos de dados e funcionalidades é definida.

A *elicitação* é a atividade de coleta de requisitos, que inclui todos os interessados. A intenção é descrever o problema que a aplicação Web deverá resolver (junto com requisitos básicos para a aplicação Web) usando a melhor informação disponível. Além disso, tenta-se identificar áreas de incerteza e lugares em que ocorrerão mudanças em potencial.

E por fim, a *negociação* normalmente é exigida para conciliar diferenças entre os diversos interessados no projeto.

2.3.1 Formulação

Formulação é uma ação de Engenharia Web que começa com a identificação de negócio, passa por uma descrição dos objetivos da aplicação Web, define as principais características da aplicação Web e estabelece uma base para a ação de elicitação seguinte. A formulação permite que os interessados e a equipe de Engenharia Web estabeleçam um conjunto comum de metas e objetivos para a criação de cada incremento da aplicação Web. Ela também identifica o escopo de esforço de desenvolvimento e oferece um meio para determinar um resultado bem-sucedido (PRESSMAN & LOWE, 2009).

A menos que a aplicação Web seja muito simples, não é aconselhável usar informações colhidas de uma ou duas pessoas como base para a *formulação*. É necessária a identificação dos interessados. Um interessado pode ser definido como "qualquer um que se beneficia de uma maneira direta ou indireta do sistema que está sendo desenvolvido" (SOMMERVILLE, 2007, p. 98). Entre exemplos de interessados, pode-se citar: gerentes de negócios, desenvolvedores, engenheiros Web, engenheiros de suporte e usuários finais.

Powell, Jones & Cutts (1998) sugerem um conjunto de questões que devem ser criadas e respondidas no início da formulação:

- Qual é a principal motivação (necessidade de negócio) da aplicação Web?
- Quais são os objetivos que a aplicação Web deve preencher?
- Quem vai usar a aplicação Web?

A resposta a cada uma dessas simples perguntas deve ser dada tão sucintamente quanto possível. Por exemplo, considere que uma livraria decidiu estabelecer um sítio de comércio eletrônico na Web para vender seus produtos diretamente aos consumidores. Uma declaração descrevendo a motivação (1ª questão) da aplicação Web poderia ser:

LivrariaVirtual.com.br permitirá aos clientes adquirir nossos produtos (livros, CDs, DVDs etc.) e recebê-los em sua residência.

É importante notar que nessa declaração nenhum detalhe é fornecido. O objetivo é limitar a finalidade global da aplicação Web e colocá-la em um contexto comercial legítimo.

Depois de discussão adicional com vários interessados, é formulada a resposta à segunda questão:

LivrariaVirtual.com.br nos permitirá vender diretamente aos clientes, reduzindo os custos e aumentando nossas margens de lucro. Permitirá também aumentar as vendas em 10% projetados com base nas atuais vendas anuais e permitirá que sejam exploradas novas regiões nas quais atualmente não temos lojas.

Finalmente, a empresa define a demografia (3ª questão) da aplicação Web:

> Os usuários previstos de **LivrariaVirtual.com.br** são clientes interessados na compra de nossos produtos.

2.3.2 Elicitação

A elicitação é a atividade de coleta de requisitos. O objetivo é identificar um conjunto de requisitos que tenha como foco: (1) o conteúdo da aplicação Web, (2) a interação do usuário e (3) a interoperabilidade com os sistemas de negócios e banco de dados existentes.

Métodos de coleta de requisitos estão presentes na literatura da área de Engenharia de Software (PRESSMAN, 2006; SOMMERVILLE, 2007). Embora a atividade de coleta de requisitos para a Engenharia Web possa ser abreviada, os objetivos dessa coleta permanecem inalterados. Adaptados para aplicações Web, esses objetivos pretendem (PRESSMAN & LOWE, 2009):

- identificar os requisitos de conteúdo;
- identificar os requisitos funcionais;
- definir os cenários de interação para as diferentes classes de usuários.

Pressman (2006) sugere que os seguintes passos sejam conduzidos para atingir esses objetivos:

- peça aos interessados para definir as categorias de usuários e desenvolver descrições de cada categoria;
- comunique-se com os interessados para definir os requisitos básicos da aplicação Web;
- analise a informação coletada e use-a para conferi-la com os interessados;
- defina os casos de uso que descrevem os cenários de interação para cada classe de usuário.

Para definir uma categoria de usuário, Pressman & Lowe (2009) sugerem um conjunto de perguntas fundamentais a serem abordadas. Usando as respostas a essas questões, deverá ser definido o menor conjunto razoável de classes de usuários.

- *Qual é o objetivo geral do usuário ao usar a aplicação Web?* Por exemplo, um usuário da **LivrariaVirtual.com.br** poderia estar interessado em comprar livros de Informática. Outro usuário poderia estar interessado em comprar CDs de música erudita. O terceiro usuário poderia apenas querer realizar uma comparação de preços. Cada um pode representar uma classe ou categoria de usuário diferente, e cada uma terá diferentes necessidades e navegará pela aplicação Web de formas diferentes.
- *Qual é a base e o nível de sofisticação do usuário em relação ao conteúdo e à funcionalidade da aplicação Web?* Para um usuário experiente, conteúdo ou funcionalidade elementar oferecem pouco benefício. Por outro lado, para usuários iniciantes é importante a presença de conteúdo e de funcionalidades elementares.
- *Como o usuário chegará à aplicação Web?* A chegada ocorrerá por meio de uma ligação (link) de outro sítio *web* ou de uma maneira mais controlada, pela página principal da organização?
- *Que características genéricas da aplicação Web o usuário gosta e não gosta?* Diferentes tipos de usuários podem ter gostos distintos. É importante tentar determinar as suas preferências (o que gostam e o que não gostam).

O próximo passo na elicitação é a identificação do conteúdo e dos requisitos funcionais. A cada interessado é solicitado rever a descrição da aplicação Web e fazer uma lista de *objetos de conteúdo* que são: (1) parte do ambiente que cerca o sistema, (2) produzidos pelo sistema e (3) usados pelo sistema para realizar suas funções. Além disso, cada interessado deverá fazer uma lista de funções que manipulam ou interagem com os objetos de conteúdo. Finalmente, listas de restrições também são desenvolvidas.

Os objetos de conteúdo descritos para a aplicação Web **LivrariaVirtual.com.br** poderiam incluir:

- Perfil da empresa.
- Especificações de produtos (livros, CDs, DVDs).
- Banco de dados de produtos (inclui preços, itens em estoque, custos de entrega etc.).
- Banco de dados de usuários (inclui identificação do usuário e outras informações relacionadas ao cliente).

Cada um desses objetos de conteúdo representa um requisito de conteúdo preliminar e também implica um conjunto de funções que serão exigidas para adquirir, manipular ou produzir esses objetos de conteúdo. A lista de funções para a aplicação Web **LivrariaVirtual.com.br** poderia incluir:

- processar dados do usuário;
- processar pedido de compra;
- processar solicitações do cliente;
- adicionar produto ao estoque.

Quanto às restrições, elas podem ser *internas* ou *externas*. *Restrições internas* estão relacionadas ao ambiente técnico (banco de dados, navegador Web etc.) em que a aplicação Web será implantada e ao ambiente de projeto em que a aplicação Web será construída (ferramentas disponíveis, hardware de desenvolvimento, padrões de software etc.). *Restrições externas* estão relacionadas ao ambiente de negócios e de uso da aplicação Web. Regras de negócio, demandas de segurança, desempenho, interoperabilidade são alguns exemplos de *restrições externas* possíveis.

Após os interessados realizarem a tarefa de produção das listas individuais, é necessário que essas listas sejam combinadas em uma lista coletiva, que elimine entradas redundantes e acrescente novas ideias que aparecerem durante a discussão. Depois que as listas coletivas para todas as áreas de assunto (objetos, funções e restrições) tiverem sido criadas, a discussão, coordenada pelo facilitador, vem em seguida. A lista combinada é encurtada, ampliada ou editada para refletir corretamente a aplicação Web a ser desenvolvida. O objetivo é desenvolver uma lista de consenso em cada área de assunto. As listas são então separadas para ação posterior (PRESSMAN & LOWE, 2009).

Quando as listas de consenso identificando os objetos de conteúdo, funções e restrições tiverem sido completadas, os interessados deverão desempenhar o papel de um usuário (de uma categoria de usuário específica) e desenvolver cenários de uso para uma ou mais entradas em cada uma das listas. Cada cenário de uso normalmente é composto de um ou dois parágrafos narrativos que descrevem como um usuário final aplicaria ou criaria um objeto de conteúdo ou interagiria com uma função da aplicação Web.

Casos de uso (COCKBURN, 2001) é uma técnica bastante utilizada para a criação de cenários do usuário. Também é uma técnica para captar os requisitos funcionais de um sistema, servindo para descrever as interações típicas entre

usuários e o próprio sistema e, fornecendo uma narrativa sobre como o sistema é utilizado (FOWLER, 2005).

No contexto da Engenharia Web, casos de uso descrevem como uma categoria de usuário específica (chamada de ator) interagirá com a aplicação Web para realizar uma ação específica. O caso de uso descreve a interação do ponto de vista do usuário.

Em geral, os casos de uso são desenvolvidos iterativamente. Somente aqueles necessários para o incremento a ser construído são desenvolvidos durante a atividade de comunicação para o incremento. Esses casos de uso podem ser refinados durante a atividade de modelagem da análise (Seção 4.4) para o incremento.

Em geral, os casos de uso são escritos no primeiro momento em um padrão de narrativa informal de um cenário de uso. Se for preciso mais formalidade, o mesmo caso de uso é reescrito usando um formato mais estruturado.

Para ilustrar, considere uma funcionalidade da **LivrariaVirtual.com.br** chamada de *Comprar Produtos*. O interessado que assume o papel de *Cliente* poderia escrever a seguinte narrativa:

Caso de uso: Comprar Produtos
Ator: Cliente

Narrativa: eu acesso o sítio web **LivrariaVirtual.com.br**, entro com meu ID de usuário e uma senha e, quando estiver validado, navego pelo catálogo e seleciono produtos – livros, CDs ou DVDs. Então, eu poderei obter informação descritiva e de preço de cada produto. Após eu inserir os produtos em meu carrinho de compras, seleciono “Pagamento” e o sistema solicita que eu preencha o formulário de remessa (endereço de entrega; opção de entrega imediata ou em três dias) e informações sobre cartão de crédito. Após o sistema validar as informações, a compra é confirmada e uma confirmação desta é enviada para mim via e-mail.

Outra maneira de escrever o caso de uso é apresentar a interação como uma sequência ordenada de ações do usuário.

Caso de uso: Comprar Produtos
Ator Principal: Cliente

Precondições: o sistema deve estar totalmente configurado; ID de usuário e senha devem ser obtidos.

Cenário principal de sucesso:

1. O cliente acessa o sítio **LivrariaVirtual.com.br**.
2. O cliente entra com seu ID de usuário.
3. O cliente entra com sua senha.
4. O sistema apresenta o catálogo de produtos.
5. O cliente navega pelo catálogo e seleciona produtos para consultar.
6. O sistema apresenta informação descritiva e de preço de cada produto.
7. O cliente insere os produtos desejados em seu carrinho de compras.
8. O cliente seleciona “Pagamento”.
9. O cliente preenche o formulário de remessa (endereço de entrega; opção de entrega imediata ou em três dias).
10. O sistema apresenta a informação completa do faturamento, incluindo a remessa.
11. O cliente preenche a informação de cartão de crédito.
12. O sistema autoriza a compra.
13. O sistema confirma imediatamente a venda.
14. O sistema envia uma confirmação para o cliente via e-mail.

Extensões:

3a. ID e senhas são incorretas ou não reconhecidas.
.1 Ver caso de uso Validar ID e senha.

10a. O sistema falha na autorização do pagamento com cartão de crédito.
.1 O cliente pode inserir novamente a informação do cartão de crédito ou cancelar a operação.

Por fim, antes do planejamento (Unidade 3) do primeiro incremento começar, é necessário desenvolver uma lista ordenada de todos os incrementos em potencial a serem implantados. Ou seja, é necessário priorizar os conteúdos e as funcionalidades (indicados por cenários de uso) a serem desenvolvidos em cada incremento. As ações de negociação são atividades-chave nessa tarefa de priorização.

2.3.3 Negociação

Cada interessado possui uma visão diferente da aplicação Web. Por exemplo, os gerentes de negócio estão interessados no conjunto de funcionalidades que resultará em aumento de receita para a empresa enquanto os usuários finais podem estar interessados em funcionalidades que já são familiares a eles e sejam fáceis de aprender e usar. Dessa forma, a negociação normalmente é exigida para conciliar diferenças entre os diversos interessados no projeto.

Durante o processo de negociação, pode-se pedir que um interessado equilibre funcionalidade, desempenho e outras características do produto ou sistema contra custo e tempo para entrega. A intenção nessa negociação é estabelecer requisitos do incremento que atendem às necessidades do cliente enquanto, ao mesmo tempo, refletem as restrições do mundo real (por exemplo, tempo, pessoal, orçamento) que foram impostas sobre a equipe de Engenharia Web.

Pressman & Lowe (2009) apresentam algumas orientações que deveriam ser consideradas durante a negociação:

- Reconheça que isso não é uma competição. Para ter sucesso, ambas as partes precisam sentir que ganharam ou que conseguiram algo. Ambas terão que se comprometer.
- Mapeie uma estratégia. Decida o que você gostaria de conseguir, o que a outra parte deseja conseguir e como você fará para que isso aconteça.
- Ouça ativamente. Não trabalhe formulando sua resposta enquanto a outra parte estiver falando. Ouça. É provável que você obtenha conhecimento que o ajudará a negociar melhor sua posição.
- Focalize os interesses da outra parte. Não assuma posições rígidas se você quiser evitar conflitos.
- Não leve para o lado pessoal. Concentre seu foco no problema que precisa ser solucionado.
- Seja criativo. Não tenha medo de pensar criativamente se estiver em um impasse.
- Esteja pronto para se comprometer. Quando um acordo tiver sido alcançado, não hesite, comprometa-se com ele e siga adiante.

2.4 Considerações finais

Esta unidade apresentou os fundamentos da atividade de comunicação do Arcabouço de Processo de Engenharia Web. Como discutido, a atividade de comunicação oferece à equipe de Engenharia Web uma estratégia organizada de levantar requisitos dos interessados e é caracterizada por três ações:

- a *formulação*, que define o contexto organizacional e de negócios para a aplicação Web;
- a *elicitação*, que consiste na atividade de coleta de requisitos, que inclui todos os interessados;
- e, por fim, a *negociação*, que normalmente é exigida para conciliar diferenças entre os diversos interessados no projeto.

2.5 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

PRESSMAN, R. S. Formulação e Planejamento para Engenharia da Web. In: _____. *Engenharia de Software*. 6. ed. São Paulo: McGraw-Hill, 2006.

PRESSMAN, R. S.; LOWE, D. Comunicação. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

UNIDADE 3

Planejamento

3.1 Primeiras palavras

Dando continuidade à discussão das atividades do Arcabouço de Processo de Engenharia Web, esta unidade discute a atividade de *planejamento* cujo objetivo é estabelecer um plano incremental para o trabalho de desenvolvimento. O plano geralmente consiste na definição de tarefas e num cronograma para o período de tempo projetado para o desenvolvimento do incremento da aplicação Web.

3.2 Problematizando o tema

Ao final desta unidade espera-se que o leitor seja capaz de reconhecer e distinguir precisamente os conceitos básicos relacionados à atividade de planejamento do Arcabouço de Processo da Engenharia Web. Dessa forma, esta unidade pretende discutir as seguintes questões:

- Quais são os objetivos da atividade de planejamento do Arcabouço de Processo de Engenharia Web?
- Quais são as principais ações realizadas durante a atividade de planejamento?

3.3 Planejamento

Outro princípio fundamental da Engenharia de Software é: "planeje o trabalho antes de começar a realizá-lo" (PRESSMAN, 2006, p. 389). A atividade de planejamento (Figura 4) oferece à equipe de Engenharia Web um modo organizado de planejar o desenvolvimento dos incrementos da aplicação Web e é caracterizado por quatro ações: análise de risco, estimação, escalonamento e monitoração.

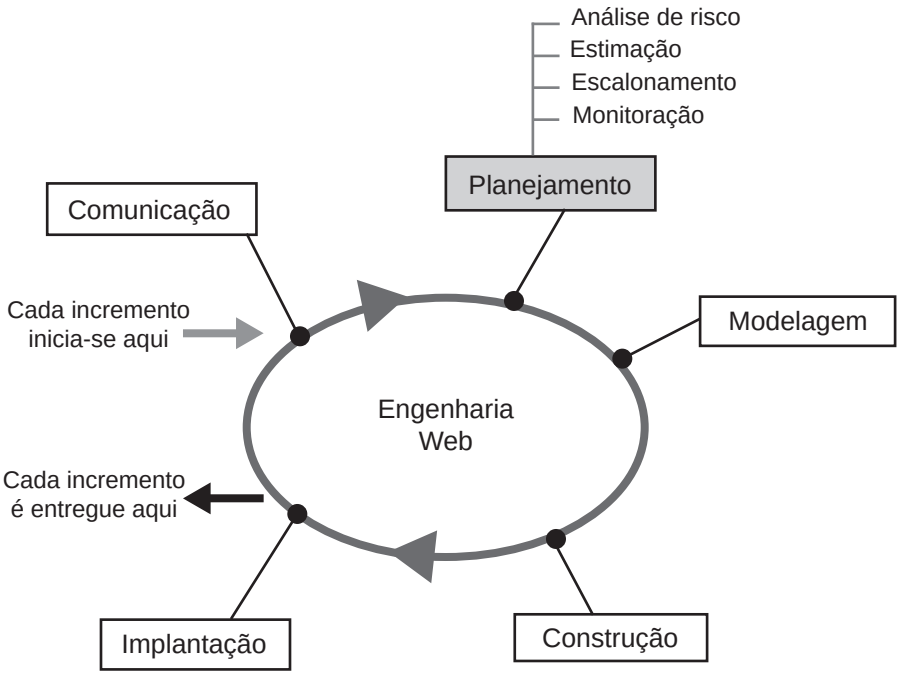


Figura 4 Planejamento no fluxo de processo de Engenharia Web.

O planejamento é realizado por todos os membros da equipe de Engenharia Web e é coordenado pelo líder de produto. Conforme foi discutido na Unidade 1, a Engenharia Web enfatiza a agilidade, e a filosofia ágil (Unidade 7) reforça a importância de equipes auto-organizadas em que a competência individual é agrupada à colaboração em grupo. No contexto da Engenharia Web, a auto-organização implica em três características:

1. a equipe de Engenharia Web se organiza para o trabalho a ser feito;
2. a equipe organiza o Arcabouço de Processo para acomodar melhor seu ambiente local;
3. a equipe organiza o cronograma de trabalho para conseguir entregar o incremento da aplicação Web.

A auto-organização tem diversos benefícios técnicos, porém o mais importante é que ela serve para melhorar a colaboração e impulsionar a moral da equipe (PRESSMAN & LOWE, 2009).

3.3.1 Análise de riscos

O gerenciamento de risco compreende uma série de tarefas que ajudam uma equipe de Engenharia Web a entender e gerenciar os muitos problemas que podem assolar o desenvolvimento de uma aplicação Web. Um risco é um

problema em potencial: ele pode acontecer, ou não. Mas, independentemente do resultado, é uma ideia realmente boa identificá-lo, avaliar sua probabilidade de ocorrência, estimar seu impacto e estabelecer um plano de contingência caso o problema realmente ocorra (PRESSMAN & LOWE, 2009).

Uma equipe de Engenharia Web considera o risco em dois níveis de granularidade diferentes:

- o impacto do risco sobre o desenvolvimento da aplicação Web como um todo (visão macroscópica);
- o impacto do risco sobre a entrega bem-sucedida do incremento da aplicação Web atualmente sendo desenvolvido (visão microscópica).

No nível macroscópico, algumas questões relacionadas ao risco precisam ser feitas e respondidas. Como exemplos pode-se citar: Os incrementos da aplicação Web planejados podem ser entregues dentro do prazo definido? Como as solicitações de mudança impactarão os cronogramas de entrega? A equipe entende os métodos, as tecnologias e as ferramentas exigidos?

No nível microscópico, as preocupações são mais básicas. Como exemplos de questões relacionadas ao risco que precisam ser feitas e respondidas pode-se citar: A atividade de comunicação desenvolveu informações suficientes para as atividades de modelagem, construção e implantação? A composição da equipe proporciona uma combinação adequada de habilidades para construir esse incremento? O conteúdo e as funcionalidades estão definidos adequadamente?

Identificação dos riscos. Quanto à identificação dos riscos, é solicitado a cada membro da equipe de Engenharia Web fazer uma lista de riscos, que pode ser organizada em uma das seguintes categorias: (1) riscos de pessoas, (2) riscos de produto e (3) riscos de processo (PRESSMAN & LOWE, 2009).

Riscos de pessoas são problemas que podem ser identificados em alguma ação ou falha humana. Por exemplo, é provável que o incremento da aplicação Web necessite a utilização de uma tecnologia específica, mas a equipe não possui nenhum membro com experiência nessa tecnologia. Um risco de pessoas relacionado à tecnologia foi identificado. Por outro lado, talvez durante as atividades de comunicação, um interessado em específico não tenha sido suficientemente cooperativo e informações cruciais para o desenvolvimento da aplicação Web não tenham sido coletadas. Um risco de pessoas relacionado à comunicação foi identificado.

Riscos de produto normalmente estão relacionados a problemas em potencial associados ao conteúdo, funcionalidades, restrições ou desempenho da aplicação Web. Por exemplo, um objeto de conteúdo importante (descrição dos produtos à venda no **LivrariaVirtual.com.br**) está desatualizado e exige considerável modificação antes que possa ser implementado.

Riscos de processo são problemas que estão ligados às ações e tarefas do arcabouço que foram escolhidas pela equipe. Em alguns casos, muito processo pode ser risco em potencial. Isto é, um risco de processo poderia ser o seguinte: o trabalho associado ao desenvolvimento de um artefato (exemplo: modelo de análise, modelo de projeto etc.) completo pode ser muito demorado e causará um atraso nas atividades do arcabouço subsequentes.

Avaliação dos riscos. Ao final, as listas de riscos são coletadas a partir dos membros da equipe e consolidadas por categoria. A equipe de Engenharia Web, então, faz uma reunião para avaliá-las. Cada risco é discutido e avaliado de duas maneiras: (1) a probabilidade que o risco aconteça e (2) as consequências (impacto) dos problemas associados ao risco, caso ele ocorra. O objetivo é avaliar os riscos de tal forma que leve à priorização.

Pressman & Lowe (2009) sugerem que seja feita uma tabela de riscos ordenada, em que cada risco é calculado por meio do seguinte produto: *probabilidade* x *impacto*. Riscos de alta probabilidade e alto impacto aparecem no alto da tabela e riscos de baixa prioridade e baixo impacto aparecem no final. Em seguida, a equipe estuda a tabela de riscos e define a linha de corte estabelecendo que apenas os riscos que se encontram acima desta receberão consideração adicional.

Plano de contingência. Como o espaço de tempo para o desenvolvimento de um incremento é curto, planos de contingência são desenvolvidos. Os membros da equipe consideram cada risco que esteja acima da linha de corte da tabela de riscos (ver discussão anterior) e respondem a três perguntas que são registradas para uso posterior (PRESSMAN & LOWE, 2009):

- Como podemos evitar o risco completamente?
- Que fatores podemos monitorar para determinar se o risco está se tornando mais ou menos provável?
- Se o risco se tornar realidade, o que vamos fazer a seu respeito?

Finalizando essa discussão sobre gerenciamento de riscos, é importante salientar que apesar de essa atividade ser iniciada durante a atividade de planejamento, na realidade ela é uma atividade guarda-chuva (outras atividades guarda-chuva serão discutidas na Seção 6) que é retornada durante todo o

fluxo de processo. O desafio para uma equipe de Engenharia Web é realizar o suficiente para que ela seja proativa sobre o risco, mas não tanto que atrase outras atividades do desenvolvimento (PRESSMAN & LOWE, 2009).

3.3.2 Estimação

O foco da estimativa está em avaliar se um incremento de aplicação Web planejado pode ser desenvolvido com os recursos disponíveis de acordo com as restrições de tempo definidas. Essa tarefa é realizada levando em consideração o conteúdo e as funcionalidades de cada incremento. Isto é, durante a atividade de estimação, os membros da equipe de Engenharia Web tentam responder uma pergunta similar a essa: "É possível implementar o terceiro incremento da **LivrariaVirtual.com.br** com três pessoas trabalhando por cinco semanas, dado nosso conhecimento atual, os riscos que identificamos e a lista de tarefas e funcionalidades que definimos para esse incremento?".

Se a resposta unânime da equipe é "sim", nenhuma atividade adicional de estimação é necessária. No entanto, se a resposta for negativa, a equipe tem duas alternativas possíveis:

1. entrar em contato com a gerência do projeto, externar suas preocupações, mas mesmo assim prosseguir o desenvolvimento do incremento, ou;
2. realizar alguma estimativa detalhada com o objetivo de ajudar a equipe e os interessados a entenderem melhor os recursos e o tempo necessários.

Duas técnicas para a estimativa detalhada de aplicações Web são apresentadas por Pressman & Lowe (2009). A primeira, *estimativa baseada em cenário*, consiste em examinar os cenários de uso definidos para o incremento a ser construído. Examinando o histórico da equipe, pode ser estabelecido um valor E_{med}, que é o esforço médio (em homens-dias) que foi exigido para implementar um cenário de uso. Para estimar o incremento atual, é necessário contar o número de cenários de uso e multiplicar pelo esforço médio. Esse número pode ser ajustado com base na complexidade percebida dos cenários de uso. Quando o esforço for determinado, ele pode ser distribuído pelas ações e tarefas de Engenharia Web ao longo da linha de tempo do projeto.

A Tabela 1 ilustra essa técnica para um possível incremento da **LivrariaVirtual.com.br**, em que o histórico da equipe indica que E_{med} = 6 homens-dias. O

primeiro cenário de uso a ser implementado é consideravelmente mais complexo do que a média, com multiplicador de complexidade de 1,5. Logo, o esforço geral exigido para implementar esse incremento é estimado como sendo 27 homens-dias.

Tabela 1 Estimativa baseada em cenários.

Cenário de uso	E_{med}	Complexidade	Esforço
Comprar produtos	6	1,5	9
Gerar boleto bancário (pagamento)	6	1,0	6
Confirmar pagamento via boleto	6	1,0	6
Autorizar pagamento via cartão de crédito	6	1,0	6
Total			27

A segunda técnica chama-se *estimativa baseada em tabela produto-processo*. Essa técnica de estimativa usa uma tabela de produto-processo e todas as principais tarefas[3] de Engenharia Web são listadas na primeira coluna da tabela. Todos os principais objetos de conteúdo e funcionalidades para um incremento são listados na primeira linha. Os membros da equipe estimam a quantidade de esforço (em homens-dias) necessária para desenvolver para cada objeto de conteúdo e funcionalidade. A Tabela 2 ilustra essa técnica de estimativa para o mesmo incremento da **LivrariaVirtual.com.br**. Levando em consideração essa técnica, o esforço geral exigido para implementar esse incremento é estimado como sendo 28 homens-dias.

3 Essas tarefas serão discutidas nas próximas seções.

Tabela 2 Estimativa baseada em tabela produto-processo.

Conteúdo e funcionalidades	(1)	(2)	(3)	(4)	(5)	(6)	Total
Especificações de produto	1	1	1	1	0,5	0,25	4,75
Especificações de boleto bancário	0,5	1	0,5	1	0,5	0,25	3,75
Especificações de cartão de crédito	0,5	0,25	0,25	0,5	0,5	0,25	2,25
Processar pedido de compra	1	1,5	1,5	2	0,5	0,25	6,75
Pagar via boleto bancário	1	1	1	2	0,5	0,25	5,75
Pagar via cartão de crédito	1	1	1	1	0,5	0,25	4,75
Totais	5	5,75	5,25	7,5	7,5	1,5	28

(1) Análise
(2) Projeto
(3) Codificação
(4) Teste
(5) Entrega
(6) *Feedback*

Pode-se observar que as duas técnicas produzem estimativas de esforço próximas. Pressman & Lowe (2009) argumentam que as duas técnicas discutidas são complementares e sugerem que estimativas sejam produzidas utilizando ambas e que os resultados sejam harmonizados para produzir uma única estimativa.

3.3.3 Escalonamento

Qualquer projeto técnico (por exemplo, o desenvolvimento de um software) consiste de centenas de pequenas tarefas que precisam ocorrer para realizar uma meta maior. Algumas dessas tarefas não se encontram no fluxo principal, e podem ser completadas sem preocupação com o impacto sobre a data de término. Outras tarefas se encontram no *caminho crítico*. Se essas tarefas críticas

atrasarem, a data de término do incremento da aplicação Web será colocada em risco.

O objetivo da equipe de Engenharia Web é listar todas as ações e tarefas para um incremento, montar uma rede que represente suas interdependências, identificar as tarefas que são críticas dentro da rede e depois rastrear seu progresso para garantir que qualquer atraso seja reconhecido. Para conseguir isso, é fundamental que um cronograma seja definido em um grau de resolução que permita que o progresso seja monitorado e o projeto seja controlado (PRESSMAN & LOWE, 2009).

O cronograma de projeto da Engenharia Web pode ser visto de dois pontos de vista:

- No nível macroscópico, o cronograma lista todos os incrementos da aplicação Web planejados e projeta datas em que cada um deles será implantado. Essas datas são preliminares, porém é uma indicação de quando o conteúdo e a funcionalidade estarão disponíveis e quando o projeto inteiro estará concluído.
- No nível microscópico, o cronograma aloca o esforço estimado para tarefas específicas durante o período planejado para o desenvolvimento de um incremento.

Na construção do cronograma de um incremento, uma lista de tarefas é criada usando-se, como ponto de partida, as tarefas genéricas derivadas como parte do arcabouço e depois as adaptando levando em consideração o conteúdo e as funcionalidades a serem desenvolvidos para um incremento de aplicação Web específico. Pressman & Lowe (2009) sugerem que cada ação de arcabouço (e suas tarefas relacionadas) possa ser adaptada em uma dentre quatro maneiras: (1) uma tarefa é aplicada como se encontra, (2) uma tarefa é eliminada porque não é necessária para o incremento, (3) uma nova tarefa (adaptada) é acrescentada e (4) uma tarefa é refinada (elaborada) para uma série de subtarefas nomeadas que se torna, cada uma, parte do cronograma.

Por exemplo, suponha que durante o desenvolvimento da primeira interação da aplicação **LivrariaVirtual.com.br**, a equipe considere que o conteúdo e as funcionalidades previstos são muito simples. Esse incremento poderia ser simplesmente a implantação da página principal da organização com informações sobre a empresa (aplicação informativa – ver Unidade 1). Nesse caso, as tarefas relacionadas à modelagem (Seção 4) poderiam ser eliminadas para esse incremento. Por outro lado, em um incremento posterior, o conteúdo e as funcionalidades previstos não são considerados elementares e, portanto,

a equipe decide que as tarefas relacionadas à modelagem necessitam ser realizadas. Nesse caso, essas tarefas são consideradas e são alocadas sobre o cronograma de desenvolvimento do incremento.

3.3.4 Monitoração

Para um projeto de Engenharia Web, a implantação de um incremento normalmente é a principal medida do progresso geral. No entanto, antes que esse incremento esteja implantado, uma indagação recorrente da equipe poderia ser: "onde nos encontramos durante esse incremento?".

Para oferecer uma resposta a essa indagação, é preciso acompanhar o progresso enquanto um incremento da aplicação Web está sendo desenvolvido. Para projetos de Engenharia Web pequenos, um incremento pode ser desenvolvido durante um período de poucas semanas (duas ou três). No melhor caso, marcos intermediários são definidos informalmente, e o cronograma do projeto pode não ter sido desenvolvido em uma granularidade que ajudará no acompanhamento.

No caso de projetos maiores, Pressman & Lowe (2009) apresentam três técnicas que podem ser usadas no acompanhamento do progresso de um incremento:

- A primeira técnica consiste em realizar uma enquete com a equipe de Engenharia Web para determinar quais atividades do Arcabouço de Processo foram finalizadas.
- A segunda técnica consiste em determinar quantos cenários do usuário foram implementados e quantos (para determinado incremento) ainda precisam ser implementados. Isso oferece uma indicação bruta do grau relativo de "completude" do incremento do projeto.
- Por fim, se a equipe de Engenharia Web tiver gasto tempo para construir um cronograma de trabalho detalhado para o incremento, o progresso pode ser acompanhado determinando-se quantas tarefas de trabalho foram finalizadas, quantos produtos de trabalho foram produzidos e revisados, e quão confiantes os membros da equipe estão na entrega na data de término do incremento.

Em um nível macroscópico, existem sinais indicando que um incremento (ou o projeto inteiro) tem problemas. Reel (1999) apresenta dez sinais que indicam que um projeto de sistemas de informação está em perigo. Esses sinais podem ser adaptados para adoção em um projeto de desenvolvimento Web.

1. A equipe de Engenharia Web não entende as necessidades de seu cliente.
2. O escopo da aplicação Web está mal definido.
3. As mudanças são mal gerenciadas.
4. A tecnologia escolhida muda.
5. As necessidades de negócios parecem estar mudando (ou estão mal definidas).
6. Os prazos não são realistas.
7. Os usuários não estão realmente interessados na aplicação Web.
8. O patrocínio foi perdido (ou nunca foi obtido de modo apropriado).
9. A equipe de Engenharia Web não possui pessoas com habilidades apropriadas.
10. Os profissionais evitam as melhores práticas e as lições aprendidas.

3.4 Considerações finais

Esta unidade apresentou os fundamentos da atividade de planejamento do Arcabouço de Processo de Engenharia Web.

O planejamento é realizado por todos os membros da equipe de Engenharia Web e é coordenado pelo líder de equipe. O planejamento considera uma série de fatores, tais como escopo de projeto e conhecimento do contexto de negócios – informação derivada dos artefatos produzidos durante a atividade de comunicação (Unidade 2).

Quando a atividade de planejamento é finalizada, o trabalho técnico começa. Na próxima unidade são discutidos muitos aspectos diferentes da modelagem no contexto de aplicações Web.

3.5 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

PRESSMAN, R. S. Formulação e Planejamento para Engenharia da Web. In: _____. *Engenharia de Software*. 6. ed. São Paulo: McGraw-Hill, 2006.

PRESSMAN, R. S.; LOWE, D. Planejamento. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

UNIDADE 4

Modelagem

4.1 Primeiras palavras

Prosseguindo a discussão sobre as atividades do Arcabouço de Processo de Engenharia Web, esta unidade discute a atividade de **modelagem**.

O objetivo da atividade de modelagem é criar modelos (por exemplo, diagramas de classe e de atividade) que auxiliam o processo de desenvolvimento da aplicação Web. Ou seja, a atividade de modelagem na Engenharia Web cria diferentes representações (modelos) de algum aspecto (conceito, funcionalidade ou conteúdo) da aplicação Web a ser construída.

4.2 Problematizando o tema

Ao final desta unidade espera-se que o leitor seja capaz de reconhecer e distinguir precisamente os conceitos básicos relacionados à atividade de modelagem do Arcabouço de Processo da Engenharia Web. Dessa forma, esta unidade pretende discutir as seguintes questões:

- Quais são os objetivos da atividade de modelagem do Arcabouço de Processo de Engenharia Web?
- Quais são as principais ações realizadas durante a atividade de modelagem?

4.3 Modelagem

No contexto do Arcabouço de Processo de Engenharia Web, a atividade de *modelagem* (Figura 5) cria uma ou mais representações conceituais de algum aspecto da aplicação Web a ser construída. Uma *representação conceitual* abrange um ou mais dos seguintes artefatos: documentos escritos, esboços, diagramas esquemáticos, modelos gráficos, cenários escritos ou protótipos em papel ou executáveis (PRESSMAN & LOWE, 2009). A intenção da atividade de modelagem é desenvolver modelos de análise e projeto que definam requisitos e, concomitantemente, representem a aplicação Web que os implementem.

Duas ações de Engenharia Web ocorrem durante a modelagem: *análise* e *projeto*. Ambas as ações têm como objetivo a produção de modelos, apesar de formas muito diferentes (PRESSMAN & LOWE, 2009):

- A modelagem de análise o ajuda a entender a natureza do problema a ser resolvido e a moldura da aplicação Web que lhe permitirá resolver esse problema.

- De modo oposto, a modelagem de projeto trata de compreender a estrutura interna da aplicação Web sendo desenvolvida e como isso cria a moldura da aplicação Web que foi identificada pelo modelo de análise.

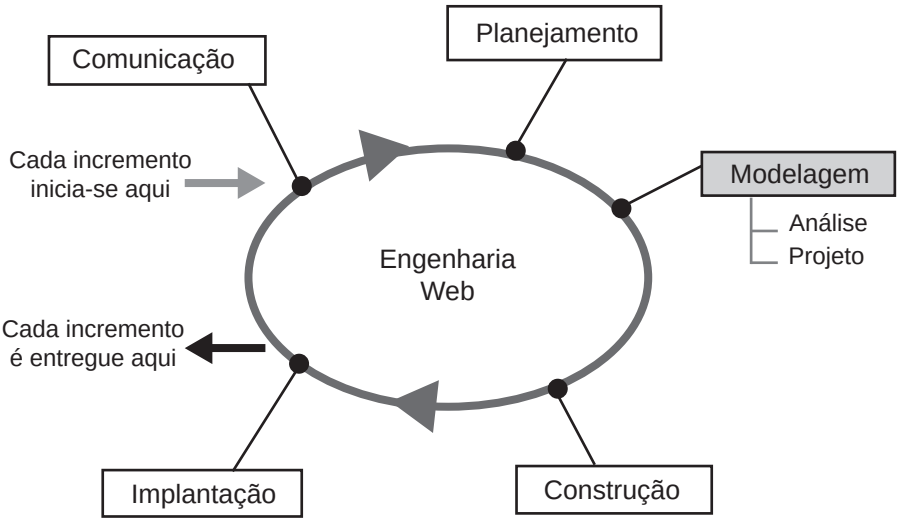


Figura 5 Modelagem.

4.4 Modelagem de análise

Conforme discutido na Unidade 2, a atividade de comunicação oferece à equipe de Engenharia Web um modo organizado de levantar requisitos dos interessados. O resultado é um conjunto de requisitos para cada incremento da aplicação Web a ser construída.

No entanto, como determinar se os requisitos coletados são suficientes? Ou completos? Ou ambíguos? Ou até mesmo contraditórios? Essa é a finalidade da modelagem de análise para aplicações Web. Ela garante que a equipe Web tenha clareza no entendimento do que está construindo.

Por outro lado, alguns desenvolvedores poderiam questionar a necessidade da modelagem de análise em um Arcabouço de Processo ágil (conforme discutido nas unidades anteriores). Eles poderiam argumentar: “análise consome recursos e tempo. Ela nos fará perder tempo quando precisamos projetar e codificar a aplicação o mais rápido possível”.

A modelagem de análise consome tempo, mas ela garante que o princípio “entenda o problema antes de começar a resolvê-lo” seja respeitado.

Nesse sentido, Pressman & Lowe (2009) argumentam a favor da análise de modelagem:

> A análise do problema exige tempo, mas resolver o problema errado leva ainda mais tempo. A questão para cada desenvolvedor de aplicação Web é

> simples: você está seguro de que compreende os requisitos do problema? Se a resposta for um enfático 'sim', então pode ser possível pular a modelagem de análise, mas se a resposta for 'não', então a análise de requisitos deverá ser realizada (PRESSMAN & LOWE, 2009, p. 123).

Dessa forma, os engenheiros Web devem entender o papel da análise consideravelmente bem para compreender perfeitamente o quanto de análise precisa ser realizada. Pressman (2006) argumenta que as seguintes condições devem ser consideradas no momento da adoção ou não da modelagem de análise:

- tamanho e complexidade da aplicação Web;
- número de interessados (a análise pode ajudar a identificar requisitos conflitantes – originários de diferentes fontes);
- tamanho da equipe de Engenharia Web;
- as metas e os objetivos (determinados durante a formulação) da aplicação Web vão afetar os fundamentos básicos do negócio;
- escala aos quais os membros da equipe de Engenharia Web trabalharam juntos antes (a análise pode ajudar a desenvolver um entendimento comum do projeto);
- escala ao qual o sucesso da organização é diretamente dependente do sucesso da aplicação Web.

4.4.1 Modelos de análise

A modelagem de análise tem como objetivo fornecer um mecanismo disciplinado para representar e avaliar o conteúdo e a funcionalidade da aplicação Web, os modos de interação que os usuários encontrarão e o ambiente e a infraestrutura em que a aplicação Web está hospedada.

Embora os modelos específicos dependam da natureza da aplicação Web, podemos identificar quatro classes principais de modelos de análise:

- **Modelo de interação**. Descreve a maneira como os usuários interagem com a aplicação Web.
- **Modelo de conteúdo**. Identifica o espectro completo do conteúdo a ser fornecido pela aplicação Web. O conteúdo inclui dados de texto, gráficos e imagens, e áudio/vídeo.
- **Modelo funcional**. Define as operações que serão aplicadas ao conteúdo da aplicação Web e descreve outras funções de processamento que são independentes do conteúdo, mas necessárias para o usuário final.

- **Modelo de configuração**. Descreve o ambiente e a infraestrutura em que a aplicação Web está hospedada.

Cada um desses modelos pode ser desenvolvido usando um esquema de representação[4] que permite que sua intenção e estrutura sejam comunicadas e avaliadas com facilidade entre os membros da equipe de engenharia Web e outros interessados.

Pressman (2006) ressalta a importância de casos de uso para a modelagem de análise:

> Um modelo de análise é guiado por informação contida nos casos de uso desenvolvidos para a aplicação. Descrições de casos de uso são analisadas para identificar potenciais classes de análise, operações e seus atributos. O conteúdo a ser apresentado pela aplicação Web é identificado e funcionalidades a ser realizadas são extraídas das descrições de casos de uso. Finalmente, os requisitos específicos da implementação devem ser desenvolvidos de modo que o ambiente e a infraestrutura que apoiam a aplicação web possam ser construídos (PRESSMAN, 2006, p. 414).

O modelo em si contempla elementos estruturais e dinâmicos. *Elementos estruturais* identificam as classes de análise e os objetos de conteúdo necessários para criar uma aplicação Web que atenda as necessidades dos interessados. *Elementos dinâmicos* do modelo de análise descrevem como os elementos estruturais interagem entre si e com os usuários finais (PRESSMAN, 2006).

4.4.1.1 Hierarquia de usuários

As pessoas que usam uma aplicação Web são as que julgarão seu sucesso. Por essa razão, é importante que a equipe de engenharia Web entenda quem serão seus usuários. Durante a atividade de comunicação (Unidade 2), cada categoria de usuário é definida à medida que os requisitos são levantados. As categorias, em muitos casos, são relativamente simples e permanecem inalteradas enquanto a análise é realizada.

No entanto, no início da análise, é importante reexaminar essas categorias com um pouco mais de cuidado para garantir que a compreensão seja completa. Isso poderia incluir a definição de uma hierarquia de usuários e depois o refinamento dos cenários de uso que já foram desenvolvidos. Essa hierarquia pode ajudar uma equipe de Engenharia Web a compreender melhor os relacionamentos entre as diferentes categorias de usuários e a própria aplicação Web.

4 Por exemplo, UML (BOOCH, RUMBAUGH & JACOBSON, 2005), que fornece uma gama de diagramas que pode ser usada para análise e projeto.

Os usuários da **LivrariaVirtual.com.br** são refinados e uma hierarquia de usuários mais detalhada é produzida. As categorias de usuários (frequentemente chamadas de atores) apresentadas na Figura 6 fornecem uma indicação da funcionalidade a ser fornecida pela aplicação Web. Referindo-se à figura, o *Usuário* do **LivrariaVirtual.com.br** no topo da hierarquia representa a classe mais geral de usuário (categoria) e é refinada em níveis mais baixos. Um *Visitante* é um usuário que visita a aplicação Web, mas não se registra. Tais usuários estão frequentemente procurando informação geral (perfil da empresa etc.), comparando preços de produtos e/ou estão interessados no conteúdo ou na funcionalidade "livre". Um *Cliente Registrado* já forneceu informação de contato (além de outros dados). Subcategorias de *Cliente Registrado* incluem:

- *Novo Cliente* – cliente registrado que deseja personalizar – incluir categorias de produtos favoritos etc. – e depois comprar produtos (e, portanto, precisa interagir com a funcionalidade de *comércio eletrônico* da aplicação Web).
- *Cliente Existente* – um usuário que já adquiriu produtos do **LivrariaVirtual.com.br** e está usando a aplicação Web para (1) comprar produtos adicionais ou (2) contatar o suporte ao cliente.

Membros da equipe de *Suporte ao Cliente* são usuários especiais que podem também interagir com o conteúdo e a funcionalidade do **LivrariaVirtual.com.br** à medida que eles ajudam os clientes que contataram esse suporte.

E por fim, um *Administrador* é um usuário especial que interage com o conteúdo e a funcionalidade do **LivrariaVirtual.com.br** para realizar as atividades administrativas, tais como cadastro de produtos, atualização de estoque, entre outros.

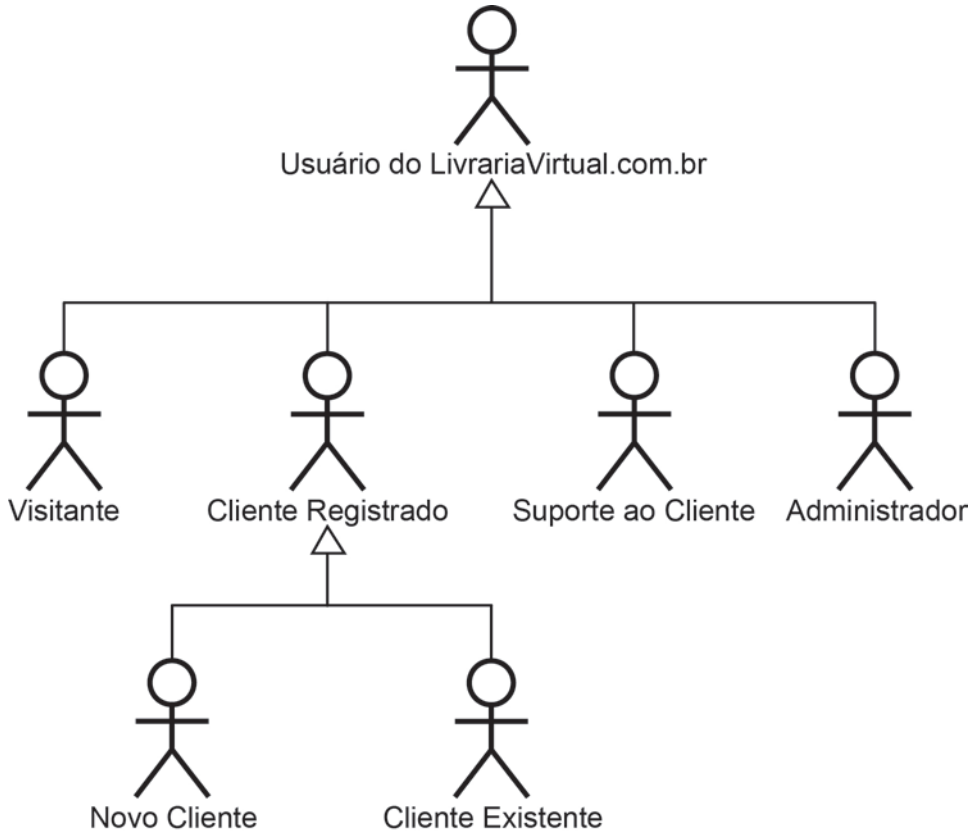


Figura 6 Hierarquia de usuários para LivrariaVirtual.com.br.

4.4.1.2 Casos de uso

Após a definição da hierarquia de usuários, talvez seja necessário refinar os casos de uso que descrevem os cenários de interação para cada classe de usuário. Relembrando que no contexto da Engenharia Web, casos de uso descrevem como uma categoria de usuário específica (chamada de ator) interagirá com a aplicação Web para realizar uma ação específica. O caso de uso descreve a interação do ponto de vista do usuário.

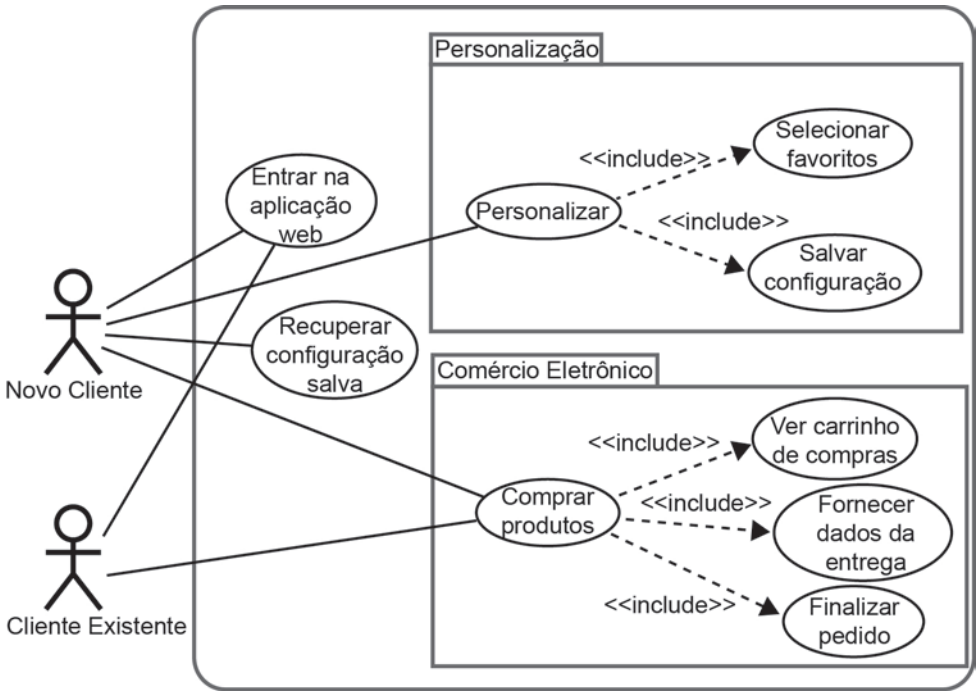


Figura 7 Diagrama de casos de uso para novo cliente.

A Figura 7 representa um diagrama de casos de uso UML para as categorias de usuários *Novo Cliente* e *Cliente Existente*. Cada elipse no diagrama representa um caso de uso que descreve uma interação específica entre os usuários e a aplicação Web. É importante salientar que as funcionalidades podem ser particionadas em pacotes UML. Dois pacotes são observados na Figura 7: *Personalização* e *Comércio Eletrônico*.

Como discutido na Unidade 2, os casos de uso são escritos no primeiro momento em um padrão de narrativa informal de um cenário de uso. Ou seja, geralmente não mais do que um parágrafo simples seria necessário para descrever um caso de uso. Se for preciso mais formalidade, o mesmo caso de uso é rescrito usando um formato mais estruturado.

4.4.1.3 Classes de análise

As classes de análise são derivadas pelo exame de cada caso de uso. Por exemplo, considere o caso de uso *Comprar Produtos* apresentado na Unidade 2.

Caso de uso: Comprar Produtos
Ator: Cliente

Narrativa: eu acesso o sítio web **LivrariaVirtual.com.br**, entro com meu ID de usuário e uma senha e, quando estiver validado, navego pelo catálogo e seleciono produtos – livros, CDs ou DVDs. Então, eu poderei obter informação descritiva e de preço de cada produto. Após eu inserir os produtos em meu carrinho de compras, eu seleciono "Pagamento" e o sistema solicita que eu preencha o formulário de remessa (endereço de entrega; opção de entrega imediata ou em três dias) e informações sobre cartão de crédito. Após o sistema validar as informações, a compra é confirmada e uma confirmação desta é enviada para mim via e-mail.

Uma rápida análise gramatical do caso de uso identifica três classes candidatas (sublinhadas): *Produto, Compra e Pagamento*. Uma descrição preliminar de cada classe é apresentada na Figura 8.

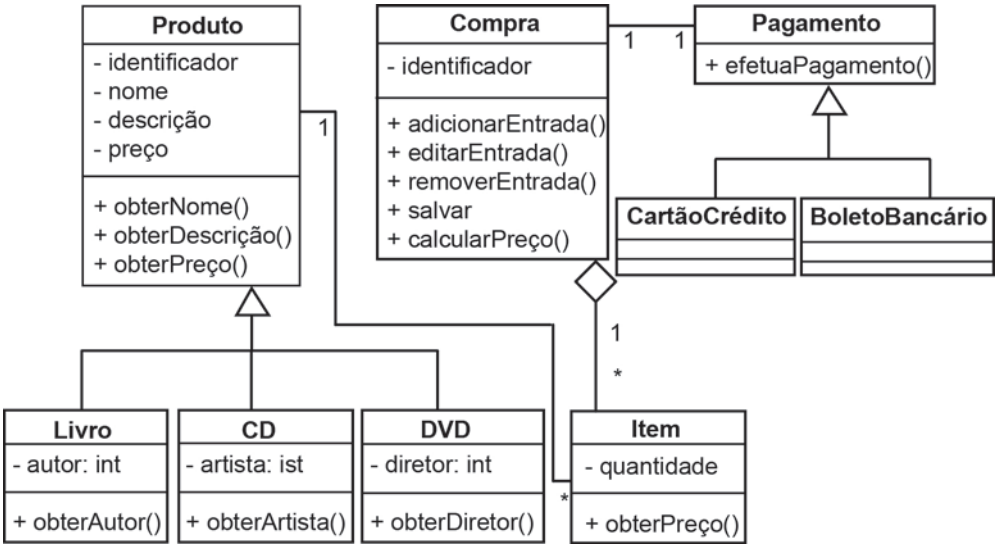


Figura 8 Classes de análise.

A classe *Produto* engloba todos os produtos que podem ser comprados. É uma representação generalizada de *Livro*, *CD* e *DVD*. Cada objeto Produto contém informação correspondente à árvore de dados para a classe (discutido na Seção 4.4.1.4). Alguns desses atributos da classe são itens de dados simples ou compostos e outros são objetos de conteúdo (veja Figura 9). Operações relevantes à classe também são apresentadas. Por questões de simplicidade, as subclasses de *Produto* apenas possuem um atributo: (1) a classe *Cd* possui o atributo *artista* que representa o nome do artista (ou conjunto musical) do CD, (2) a classe *Dvd* possui o atributo *diretor* que representa o nome do diretor do filme gravado no DVD e (3) a classe *Livro* possui o atributo *autor* que representa o nome do autor do livro.

A classe *Compra* engloba uma lista de produtos que um *Cliente Registrado* (*Novo Cliente* ou *Cliente Existente*) selecionou. *Compra* é uma abstração do *carrinho de compras* e consiste em uma agregação de *Item* – uma classe que constrói uma lista composta de produtos a serem comprados. E por fim, a classe *Pagamento* é uma representação generalizada das diferentes formas de pagamento: pagamento por cartão de crédito (*CartãoCrédito*) ou pagamento por boleto bancário (*BoletoBancário*).

Cada caso de uso coletado para a aplicação **LivrariaVirtual.com.br** é analisado para procura de classes de análise e, consequentemente, o *modelo de classes de análise* é refinado para refletir as novas classes de análise identificadas.

4.4.1.4 Modelo de conteúdo

O *modelo de conteúdo* contém elementos estruturais que fornecem uma visão importante dos requisitos de conteúdo para uma aplicação Web. Esses elementos estruturais englobam objetos de conteúdo (por exemplo, texto, imagens gráficas, fotografias, áudio etc.) apresentados como partes da aplicação Web. Além disso, o modelo de conteúdo inclui todas as classes de análise (Seção 4.4.1.3).

Analogamente a outros elementos do modelo de análise, o modelo de conteúdo é derivado de um exame cuidadoso dos casos de uso. Objetos de conteúdo são extraídos de casos de uso pelo exame da descrição de cenário à procura de referências diretas ou indiretas ao conteúdo. Por exemplo, no caso de uso *Comprar Produtos*, a seguinte sentença está presente: “então, eu poderei obter informação descritiva e de preço de cada produto”.

Nessa sentença a referência ao conteúdo é implícita. Nesse caso, o engenheiro Web pode solicitar ao autor do caso de uso melhor detalhamento sobre o

que “informação descritiva e de preço de cada produto” significa. Então, o autor do caso de uso poderia indicar que “informação descritiva” inclui (1) uma fotografia/imagem do produto e (2) uma descrição técnica (autor, artista etc.) longa do produto. Geralmente são utilizadas *árvores de dados* (SRIDHAR & MANDYAM, 2012) que mostram o relacionamento entre os objetos de conteúdo e/ou a hierarquia de conteúdo mantida por uma aplicação Web.

Considere a árvore de dados criada para os produtos do **LivrariaVirtual.com.br** apresentada na Figura 9. A árvore representa uma hierarquia de informações usada para descrever o produto (classe de análise discutida na Seção 4.4.1.3). Itens de dados simples ou compostos (um ou mais valores) são representados por retângulos não sombreados. Objetos de conteúdo são representados por retângulos sombreados. Na figura, *descrição* é definida por quatro objetos de conteúdo (os objetos sombreados).

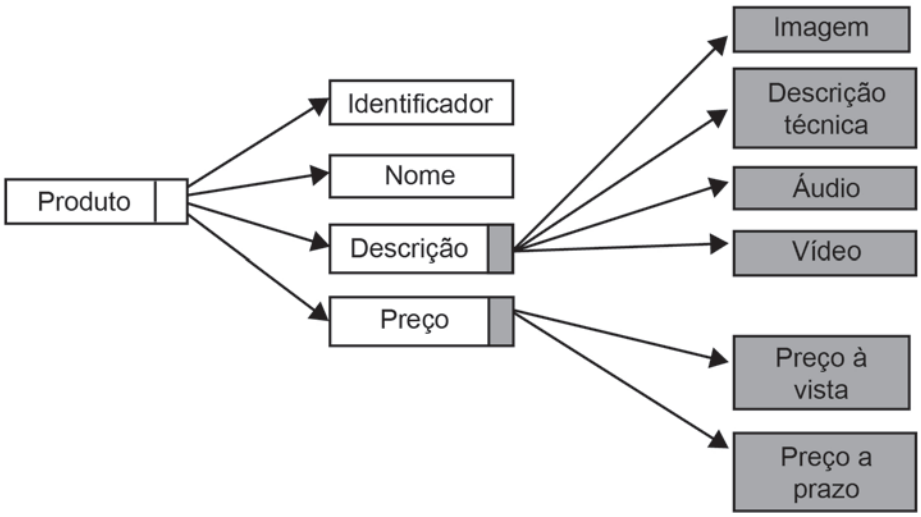


Figura 9 Árvore de dados.

4.4.1.5 Modelo de interação

A grande maioria de aplicações Web possibilita uma interação (“conversa”) entre um usuário final e a funcionalidade, o conteúdo e o comportamento da aplicação. Esse modelo de *interação* é composto de quatro elementos: (1) casos de uso, (2) diagrama de sequência e (3) diagramas de estado (PRESSMAN, 2006). Casos de uso já foram discutidos nas seções anteriores. Nesta seção serão apresentados os outros dois elementos.

Diagramas de sequência. Diagramas de sequência UML (BOOCH, RUMBAUGH & JACOBSON, 2005) fornecem uma representação abreviada da forma pela qual ações de usuário (elementos dinâmicos de um sistema)

colaboram com as classes de análise (elementos estruturais). Desde que as classes de análise são extraídas a partir dos casos de uso, há a necessidade de garantir a rastreabilidade entre as classes de análise (elementos estruturais) e os casos de uso (elementos dinâmicos). Um diagrama de sequência para o caso de uso *Comprar Produtos* é apresentado na Figura 10. Nesse diagrama de sequência é apresentada a interação de um *Cliente Registrado* (*Novo Cliente* ou *Cliente Existente*) com a aplicação Web que ocorre durante a compra de produtos. É assumido que o cliente já está autenticado no sistema (caso de uso *Entrar na Aplicação Web* foi realizado com sucesso).

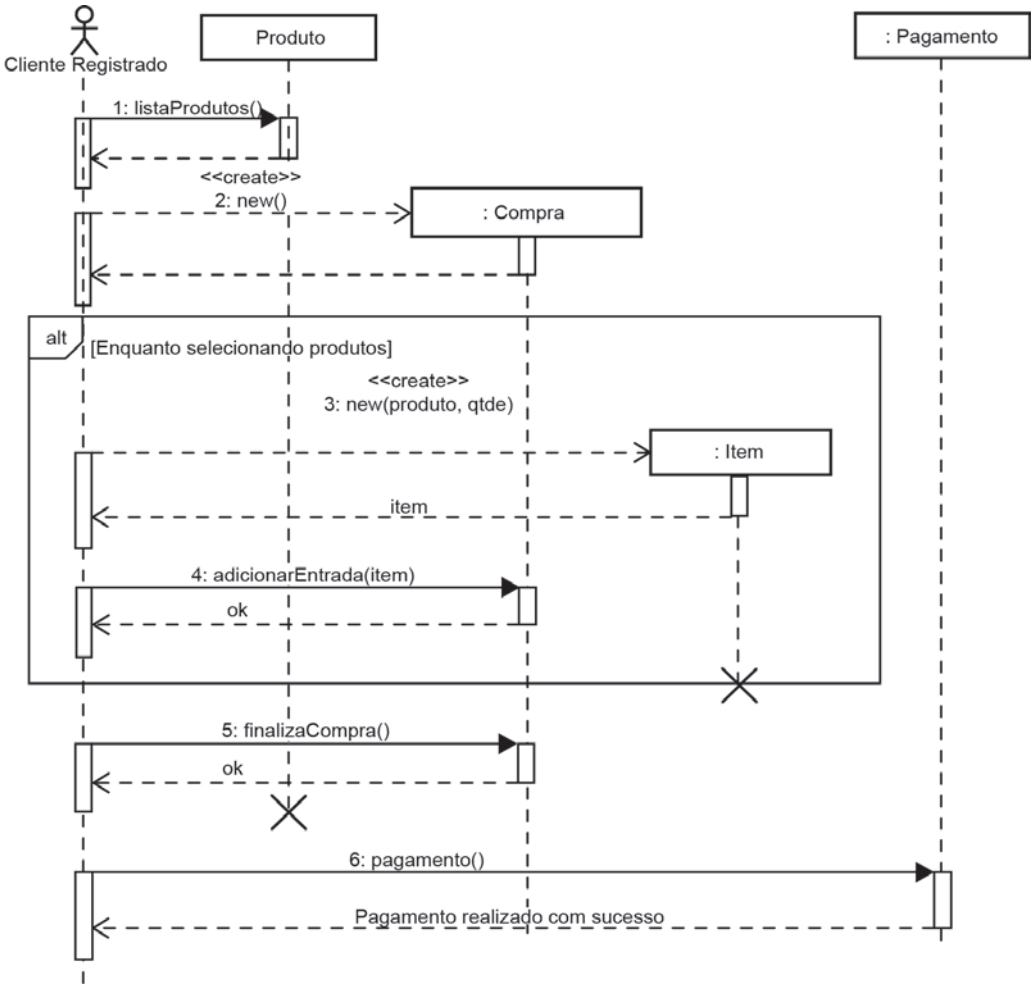


Figura 10 Diagrama de sequência.

Diagramas de estado. Diagramas de estado UML (BOOCH, RUMBAUGH & JACOBSON, 2005) fornecem uma outra representação do comportamento dinâmico da aplicação *Web* à medida que uma interação ocorre. A Figura 11 ilustra os estados externamente observáveis do objeto *Compra* (caso de uso *Compra de Produtos*). Esse diagrama indica os eventos necessários para deslocar de um estado para outro. É importante observar que as operações (eventos

no diagrama de estado) estão particionadas nas classes *Compra*, *Item* e *Pagamento* presentes no modelo de classes de análise.

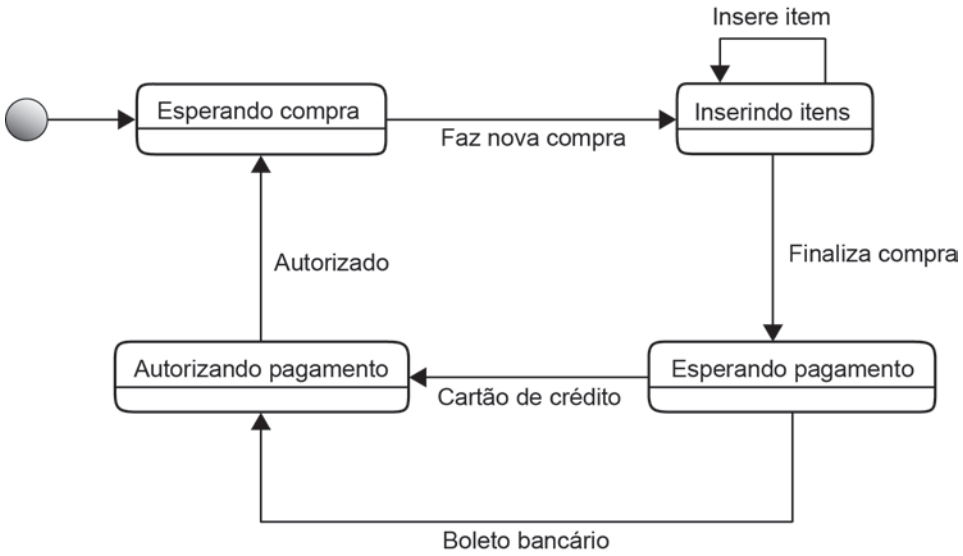


Figura 11 Diagrama de estado.

4.4.1.6 Modelo funcional

Segundo Pressman (2006), o modelo funcional atende a dois elementos do processamento da aplicação Web, cada um representando um nível diferente de abstração procedimental:

- Funcionalidade que é disponibilizada pela aplicação Web aos usuários finais. Isso engloba qualquer funcionalidade iniciada pelo usuário final.
- Operações contidas nas classes de análise que implementam comportamentos (manipulando atributos) associados à classe.

Independentemente do nível de abstração, diagramas de atividade UML (BOOCH, RUMBAUGH & JACOBSON, 2005) podem ser utilizados para representar detalhes de processamento. A Figura 12 apresenta um diagrama de atividade para a operação *calcularPreço* que é parte da classe de análise *Compra*.

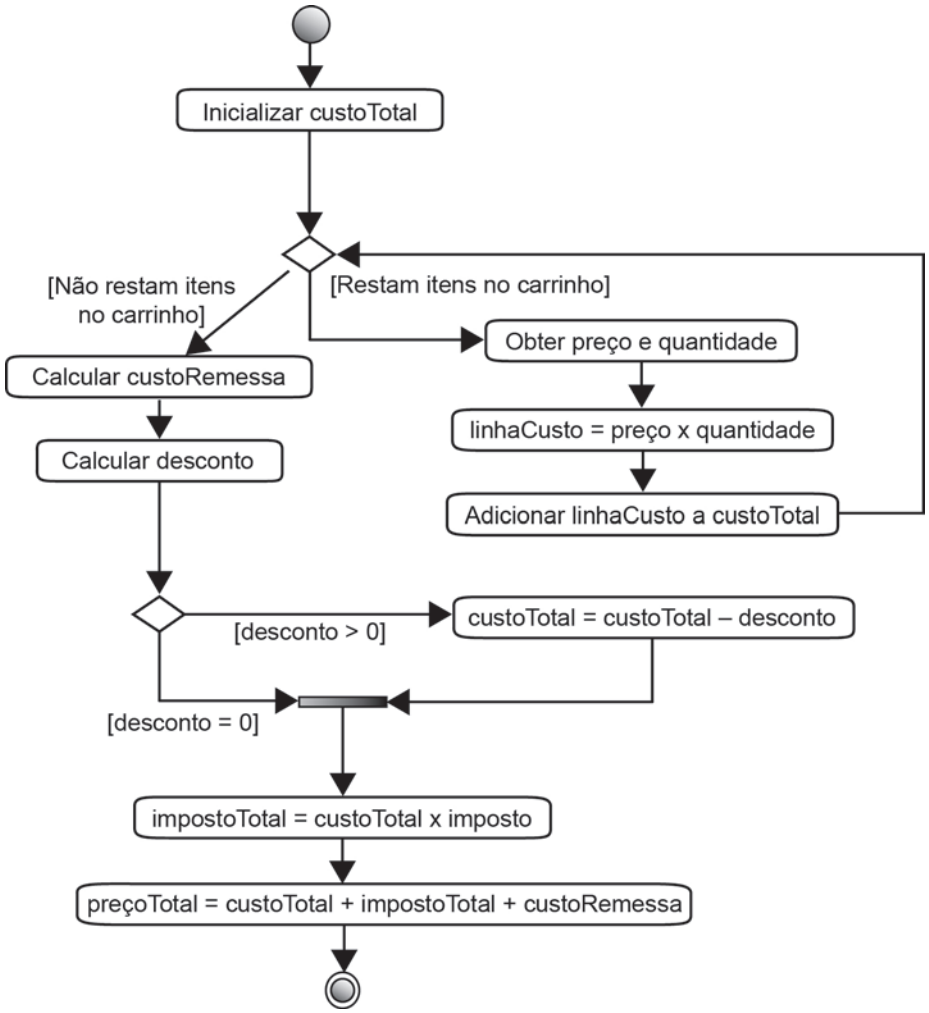


Figura 12 Diagrama de atividade.

4.4.1.7 Modelo de configuração

Aplicações Web devem ser projetadas e implementadas de tal forma que acomodem uma variedade de ambientes tanto do lado do servidor quanto do lado do cliente. Por exemplo, uma aplicação Web pode ser hospedada em um servidor que fornece acesso via Internet, Intranet ou Extranet. Dessa forma, o hardware do servidor e o ambiente do sistema operacional devem ser especificados. Por outro lado, apesar da existência de padrões, é notório que cada navegador Web (*browser*) tem suas peculiaridades.

Segundo Pressman (2006), o *modelo de configuração*, na maioria dos casos, é uma lista de atributos no lado do servidor e no lado do cliente. No entanto, para aplicações Web mais sofisticadas, diversas complexidades de configuração (por exemplo, distribuição de carga entre múltiplos servidores, banco de dados remotos) podem ter impacto sobre a análise e o projeto. Nesses casos, diagramas de implantação UML (BOOCH, RUMBAUGH & JACOBSON, 2005)

podem ser utilizados para representar situações em que arquiteturas de configuração complexas são necessárias.

4.4.1.8 Análise relacionamento-navegação

No contexto de aplicações Web, cada elemento (classe de análise, objeto de conteúdo, funcionalidade) tem o potencial de estar ligado a todos os outros elementos. Na realidade, mecanismos de ligação (link), e a navegação que eles proporcionam fornecem uma ferramenta de acesso fundamental dentro das aplicações Web. No entanto, quando o volume de conteúdo e a complexidade da interação com o usuário aumentam, o número de ligações aumenta e, portanto, a complexidade navegacional em toda a aplicação Web também aumenta.

A questão fundamental, portanto, é como estabelecer as ligações apropriadas entre objetos de conteúdo e entre as funções que oferecem funcionalidades exigidas pelo usuário (PRESSMAN, 2006).

A Análise Relacionamento-Navegação (ARN) oferece uma série de etapas de análise que se esforçam em identificar relacionamentos entre os elementos descobertos como parte da criação do modelo de análise (YOO & BIEBER, 2001). A técnica ARN é organizada em cinco etapas:

1. Análise dos interessados. Identifica as diversas categorias de usuários e estabelece uma hierarquia apropriada de interessados.
2. Análise de elemento. Identifica os objetos de conteúdo e os elementos funcionais que são de interesse para os usuários finais.
3. Análise de relacionamento. Descreve os relacionamentos que existem entre os elementos da aplicação Web.
4. Análise de navegação. Examina como os usuários poderiam acessar elementos individuais ou grupos de elementos.
5. Análise de avaliação. Considera questões pragmáticas (por exemplo, custo-benefício) associadas à implementação dos relacionamentos definidos anteriormente.

A descrição detalhada da técnica ARN está além do escopo deste livro. O leitor interessado deve consultar Yoo & Bieber (2001) para mais detalhes.

4.5 Modelagem de projeto

Enquanto o foco da *modelagem de análise* está no “o que”, o foco da *modelagem de projeto* está no “como”. A modelagem de projeto engloba atividades técnicas e não técnicas. A aparência do conteúdo é desenvolvida como parte do projeto gráfico, o leiaute de estética da interface com o usuário é criado como parte do projeto da interface, e a estrutura técnica da aplicação Web é modelada como parte do projeto arquitetural e navegacional (PRESSMAN, 2006).

Em geral, um bom projeto deve guiar a construção da aplicação Web e garantir que as soluções adotadas maximizem alguns requisitos de qualidade, tais como: simplicidade, robustez e usabilidade. Para que o objetivo de ser um guia eficaz para a construção seja alcançado, o projeto deve ser inteligível e incorporar um nível apropriado de detalhe. No entanto, o leitor poderia indagar: “que nível de detalhe é o apropriado?”.

Pressman & Lowe (2009), respondendo precisamente essa indagação, argumentam que “no caso do projeto de aplicações Web, precisamos de informações suficientes para tornar o projeto útil, mas não tanto que o processo de projeto se torne arrastado” (PRESSMAN & LOWE, 2009, p. 157). O que se pode abstrair dessa discussão é que é fundamental que a equipe Web entenda consideravelmente bem o papel do projeto para decidir o quanto que desta fase precisa ser realizada em cada incremento do desenvolvimento da aplicação Web.

Para realizar o projeto de um modo que alcance alta qualidade, a equipe Web deve entender e aplicar as principais tarefas de projeto. Desde que a mistura adequada de estética, conteúdo e tecnologia, é dependente da natureza da aplicação, as atividades de projeto que são enfatizadas variam no desenvolvimento de cada aplicação.

Pressman & Lowe (2009) apresentam uma pirâmide de projeto (Figura 13) que contém as ações de projeto (níveis) que ocorrem quando um modelo de projeto completo é criado.

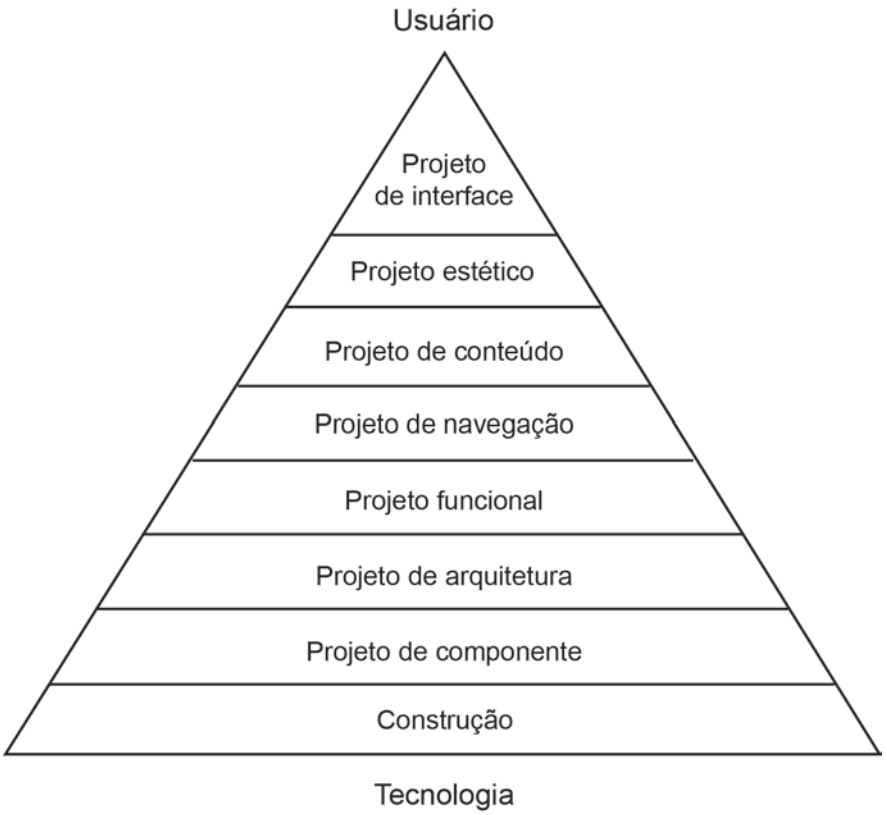


Figura 13 Pirâmide de projeto de Aplicação Web.

- *Projeto de interface* – descreve a estrutura e a organização da interface com o usuário. Ele inclui uma representação do leiaute da tela, uma definição dos modos de interação e uma descrição dos mecanismos de navegação.
- *Projeto estético* – também chamado de *projeto gráfico*, descreve a aparência e o estilo da aplicação Web. Ele inclui esquemas de cor, leiaute geométrico, tamanho de texto, fonte e posicionamento, o uso de figuras gráficas e decisões estéticas relacionadas.
- *Projeto de conteúdo* – define o leiaute, a estrutura e o esboço para todo o conteúdo que é apresentado como parte da aplicação Web. Ele estabelece os relacionamentos entre os objetos de conteúdo.
- *Projeto de navegação* – representa o fluxo de navegação entre os objetos de conteúdo e para todas as funcionalidades da aplicação Web.
- *Projeto funcional* – identifica o comportamento geral e a funcionalidade que é provida pela aplicação Web. Na maioria das aplicações, as funcionalidades seguem padrões bem estabelecidos, tais como: carrinhos de compras e comércio eletrônico. Nesses casos, o projeto funcional não é necessário. Por outro, quando o projeto foca explicitamente funcionalidades altamente especializadas, o projeto funcional é fundamental.

- *Projeto de arquitetura* – identifica a estrutura geral para a aplicação Web.
- *Projeto de componente* – desenvolve a lógica de processamento detalhada exigida para implementar componentes funcionais.

4.5.1 Projeto de interação

Os dois níveis superiores da pirâmide – projeto de interface e projeto estético – formam a base da interação do usuário com o sistema e, portanto, podem ser agrupados como *projeto de interação*.

4.5.1.1 Projeto de interface

A interface do usuário é a *primeira impressão* de uma aplicação Web. Não importa o valor do seu conteúdo, da sofisticação de suas capacidades de processamento e serviços, e do benefício geral da aplicação Web, se uma interface mal projetada desaponta o usuário em potencial. Devido à grande quantidade de aplicações Web concorrentes, a interface precisa atrair imediatamente um usuário em potencial. Caso contrário, o usuário vai para outro lugar.

Pressman & Lowe (2009) salientam que

> toda interface de usuário deverá apresentar as seguintes características: ser fácil de usar, ser fácil de aprender, ser fácil de navegar, ser intuitiva, ser consistente, eficaz, livre de erros e funcional. Ela deve oferecer ao usuário final uma experiência satisfatória e recompensadora (PRESSMAN & LOWE, 2009, p. 181).

O projeto de interação para as aplicações Web começa não com uma consideração da tecnologia ou das ferramentas, mas com um exame cuidadoso do usuário final. Durante a modelagem de análise (Seção 4.4), foi desenvolvida uma hierarquia de usuários. Cada categoria de usuário pode ter necessidades sutilmente diferentes, querer interagir com a aplicação Web de maneiras diferentes e exigir funcionalidade e conteúdo exclusivos.

Ou seja, a informação contida no modelo de análise forma a base para a criação de um leiaute de tela que representa o projeto gráfico e o posicionamento de ícones, a definição de textos, a especificação e nomeação das janelas e a especificação dos itens de menus. Ferramentas são então usadas para criar o protótipo e, por fim, para implementar o modelo de projeto da interface (PRESSMAN & LOWE, 2009).

A discussão detalhada de técnicas de projeto de interfaces Web está além do escopo deste livro. No entanto, é fundamental salientar que a maioria das

diretrizes encontradas na literatura de *projeto de interface com o usuário* (DIX et al., 2004; SHARP, ROGERS & PREECE, 2007) aplica-se igualmente ao projeto de interfaces de aplicações Web. Além disso, existem alguns trabalhos específicos (NIELSEN, 2000; PRESSMAN & LOWE, 2009) para o projeto de interfaces de aplicações Web. O leitor interessado deve consultar esses trabalhos para mais detalhes.

4.5.1.2 Projeto estético

O *projeto estético* é um esforço artístico que complementa os aspectos técnicos da Engenharia Web. Na ausência desse projeto, uma aplicação Web pode ser funcional, porém não atraente. Geralmente o projeto estético envolve duas atividades: *projeto do leiaute* e *projeto gráfico*.

Pressman & Lowe (2009) argumentam que para realizar um projeto estético efetivo, é necessário retomar a hierarquia de usuários refinada durante a modelagem de análise e realizar as seguintes indagações: *quem são os usuários da aplicação Web* e *que aparência eles desejam?*

Além disso, é importante salientar que toda página Web tem uma quantidade limitada de espaço que pode ser usado para apoiar recursos de navegação, conteúdo informativo e funcionalidade controlada pelos usuários. O planejamento desse espaço é realizado durante o projeto estético.

Apesar de não existirem regras absolutas para o *projeto do leiaute*, Pressman & Lowe (2009) sugerem algumas orientações que merecem ser consideradas:

- *Não tenha medo do espaço em branco*. É desaconselhável encher cada espaço de uma página Web com informações. Além de criar um caos visual desagradável aos olhos, também dificulta ao usuário identificar a informação ou os recursos necessários.
- *Enfatize o conteúdo*. Essa é a razão para o usuário estar lá. Como exemplo, Nielsen (2000) sugere que a página Web típica deverá ter a seguinte divisão: 80% dedicado ao conteúdo, e 20% à navegação e outros recursos.
- *Organize elementos do leiaute do canto superior esquerdo ao inferior direito*. A grande maioria dos usuários verá uma página Web da mesma maneira como eles folheiam a página de um livro – do canto superior esquerdo para o canto inferior direito. Dessa forma, devem-se colocar os elementos de alta prioridade na parte superior esquerda do espaço da página.
- *Agrupe navegação, conteúdo e funcionalidade geograficamente dentro da página*. Usuários procuram padrões em praticamente todas as coisas

e, portanto, se não houver padrões discerníveis dentro de uma página Web, a frustração do usuário provavelmente aumentará.

- *Não estenda o espaço da sua página com a barra de rolagem*. Embora a rolagem frequentemente seja necessária, a maioria dos estudos indica que os usuários prefeririam não rolar. É preferível reduzir o conteúdo da página ou apresentar o conteúdo necessário em múltiplas páginas.
- *Considere a resolução e o tamanho da janela do navegador ao projetar o leiaute*. Nielsen (2000) sugere que o projeto deverá especificar todos os itens de leiaute como uma porcentagem do espaço disponível, ao invés de definir tamanhos fixos.
- *Projete o leiaute visando à liberdade de navegação*. Um leiaute genérico de todas as páginas da aplicação Web deve assumir que o usuário navegará para a página de maneiras não esperadas (por exemplo, por meio de um link direto de um mecanismo de busca). O leiaute deve ser projetado para acomodar a chegada imprevisível sem confusão (da parte do usuário). Também é importante garantir que um usuário não terá permissão para navegar para uma página protegida sem primeiro passar pela validação de segurança.
- *Não assuma que o leiaute será consistente em todos os diferentes monitores e navegadores*. O leiaute deve ser projetado de tal modo que seja eficaz em monitores grandes e pequenos, usando recursos que são traduzidos adequadamente na maioria dos navegadores populares.
- *Se você usa fotos, mantenha-as em formato pequeno com a opção de ampliar*. Arquivos .jpeg grandes podem levar tempo para serem baixados. A maioria dos usuários estará satisfeita com uma foto em miniatura, desde que tenha a opção de ver a versão maior.
- *Utilize folhas de estilo em cascata*[5] *(caso deseje leiaute, aparência e estilo coesos por todas as páginas da aplicação)*. Uma folha de estilo permite que os desenvolvedores especifiquem a aparência e o estilo (por exemplo, fonte, tamanho e estilo) para todas as páginas Web.

Por fim, o *projeto gráfico* considera cada aspecto da aparência e estilo de uma aplicação Web. O processo de projeto gráfico começa com o leiaute (*projeto de leiaute*) e prossegue para uma consideração dos esquemas de cor globais, tipos de texto, tamanhos e estilos, o uso de mídia suplementar (por exemplo, áudio, vídeo, animação etc.); e todos os outros elementos estéticos de uma aplicação.

5 Em inglês: Cascading Style Sheets (CSS).

4.5.2 Projeto da informação

Os dois níveis seguintes da pirâmide – classificados como *projeto da informação* – focalizam a informação: o conteúdo disponibilizado pela aplicação Web, o modo como esse conteúdo é organizado e a maneira como o usuário trafega pela aplicação Web para acessar o conteúdo ou a funcionalidade.

O conteúdo é a "essência" da maioria das aplicações Web. Nesse sentido, pode-se afirmar que apesar da reconhecida importância de interfaces eficazes de usuários, navegação clara e funcionalidade rica, essas características não são suficientes para garantir o sucesso de uma aplicação Web. Ou seja, uma aplicação Web está fadada ao fracasso, mesmo que exiba as características citadas, caso seu conteúdo não seja significativo ou não possa ser localizado.

Nesta seção, o foco será dado a três questões relacionadas ao projeto da informação (PRESSMAN & LOWE, 2009):

- *Conteúdo*. Que conteúdo está disponível?
- *Composição*. Que visões nesse conteúdo estarão disponíveis aos usuários?
- *Navegação*. Como os usuários acessam a essas visões?

4.5.2.1 Projeto de conteúdo

O relacionamento entre os objetos de conteúdo definidos como parte do modelo de análise (Figura 9) e os objetos de projeto que representam conteúdo é análogo ao relacionamento de classes de análise e classes de projeto do paradigma orientado a objetos. No contexto da Engenharia Web, um *objeto de conteúdo* está mais próximo de um objeto de dados convencional. Um objeto de conteúdo tem atributos que incluem informação específica de conteúdo (normalmente definida durante a modelagem de análise) e atributos específicos de implementação, que são especificados como parte da modelagem de projeto.

Como exemplo, considere a classe de análise desenvolvida para a aplicação **LivrariaVirtual.com.br** denominada *Produto* desenvolvida na Seção 4.4.1.3 e apresentada na Figura 14. Na Seção 4.4.1.3, foi mencionado um atributo *descrição* representado aqui como uma classe de projeto denominada *DescriçãoProduto* composta de quatro objetos de conteúdo: *DescriçãoTécnica*, *Imagem*, *Áudio* e *Vídeo*, mostrados como objetos sombreados na figura. A informação contida nos objetos de conteúdo é indicada como atributos. Por exemplo, *Imagem* (uma imagem.jpg) tem os atributos *dimH* (dimensão horizontal), *dimV* (dimensão vertical) e *estilo* (estilo de borda).

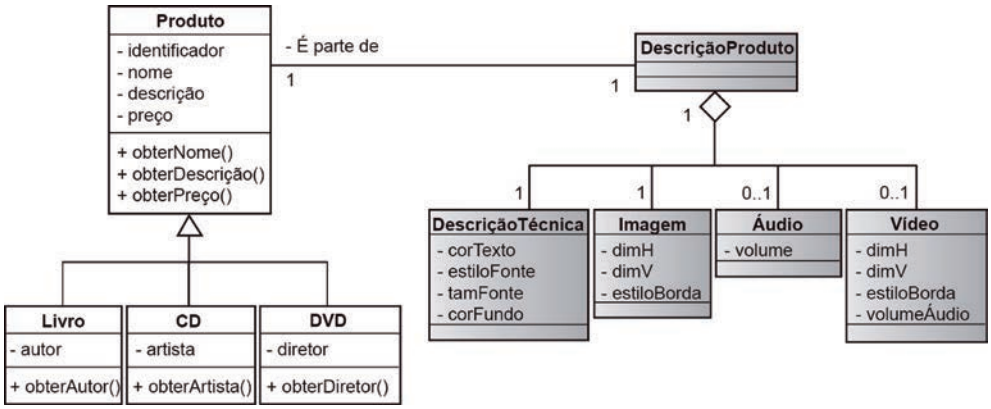


Figura 14 Projeto dos objetos de conteúdo.

Associação e agregação UML (BOOCH et al., 2005) podem ser usadas para representar relacionamentos entre objetos de conteúdo. Por exemplo, a associação UML mostrada na Figura 14 indica que uma *DescriçãoProduto* é usada para cada instância da classe *Produto*. *DescriçãoProduto* é composta dos quatro objetos de conteúdo apresentados. No entanto, a notação de multiplicidade indica que *Áudio* e *Vídeo* são opcionais (nenhuma ocorrência é possível) e uma *DescriçãoTécnica* e uma *Imagem* são obrigatórias.

Composição. Uma vez modelados todos os objetos de conteúdo, a informação que cada um deles deve disponibilizar precisa ser criada e depois formatada para melhor atender às necessidades do cliente. À medida que os objetos são formatados, eles são "unidos" para compor as páginas da aplicação Web. Pressman (2006) argumenta que a quantidade de objetos de conteúdo incorporados em uma única página é dependente das necessidades dos usuários, das restrições impostas pela velocidade de *download* das conexões de Internet e das restrições impostas pela quantidade de rolagem que o usuário tolera.

4.5.2.2 Projeto de navegação

Assim como muitas atividades de Engenharia Web, o projeto de navegação inicia-se com o exame da hierarquia de usuários e dos casos de uso relacionados que foram desenvolvidos para cada categoria de usuário (ator). Cada ator pode ter necessidades de navegação sutilmente diferentes. Em adição, os casos de uso, desenvolvidos para cada ator, vão definir um conjunto de classes que englobam um ou mais objetos de conteúdo ou de funcionalidades da aplicação Web.

À proporção que cada usuário interage com a aplicação Web, encontra uma série de *unidades de semânticas de navegação*[6] (USN), que pode ser definida como "um conjunto de informação e estruturas de navegação relacionadas

6 Em inglês: Navigation Semantic Unit (NSU).

que colaboram para o atendimento de um subconjunto de requisitos de usuário" (CACHERO et al., 2002, p. 3).

Para ilustrar o desenvolvimento de uma USN, considere o caso de uso *Comprar Produtos* apresentado na Unidade 2 e reproduzido aqui.

Caso de uso: Comprar Produtos
Ator: Cliente

Narrativa: Eu acesso o sítio web **LivrariaVirtual.com.br**, entro com meu ID de usuário e uma senha e, quando estiver validado, navego pelo catálogo e seleciono produtos – livros, CDs ou DVDs. Então, eu poderei obter informação descritiva e de preço de cada produto. Após eu inserir os produtos em meu carrinho de compras, eu seleciono "Pagamento" e o sistema solicita que eu preencha o formulário de remessa (endereço de entrega; opção de entrega imediata ou em três dias) e informações sobre cartão de crédito. Após o sistema validar as informações, a compra é confirmada e uma confirmação desta é enviada para mim via e-mail.

Os itens sublinhados na descrição do caso de uso representam classes e objetos de conteúdo que serão incorporados em uma ou mais USNs que vão permitir que clientes realizem o cenário descrito no caso de uso *Comprar Produtos*.

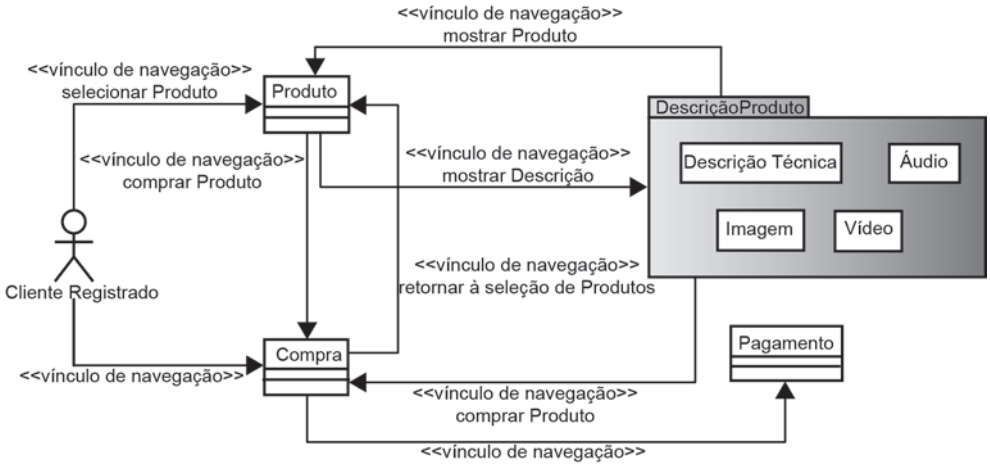


Figura 15 Criação de uma unidade de semântica de navegação.

A Figura 15 ilustra uma análise semântica parcial da navegação implícita no caso de uso *Comprar Produto*. Importantes classes do domínio do problema são apresentadas com objetos de conteúdo selecionados. Esses itens são nós de navegação – cada uma das setas representa um vínculo de navegação e é rotulada com a ação iniciada pelo usuário que ocasiona a ocorrência do vínculo. O projetista cria uma USN para cada caso de uso associado a cada papel de usuário.

À medida que o projeto prossegue, os mecanismos de navegação devem ser definidos. Pressman (2006) lista algumas opções possíveis:

- *Link individual de navegação* – vínculos baseados em texto, botões e metáforas gráficas.
- *Barra de navegação horizontal* – relaciona as principais categorias de conteúdo ou funcionais em uma barra que contém os links apropriados.
- *Coluna vertical de navegação* – disponibiliza as principais categorias de conteúdo ou funcionais ou exibe todos os principais objetos de conteúdo.
- *Mapas* – fornecem um sumário para a navegação para todos os objetos de conteúdo e de funcionalidade contidos na aplicação Web.

Por fim, é importante salientar que para escolher os mecanismos de navegação, o projetista deve estabelecer convenções e ajudas de navegação apropriadas. O objetivo é tornar a navegação mais amigável ao usuário. Por exemplo, para navegação em texto, a cor deve ser usada para indicar os links de navegação e fornecer indicação dos links já percorridos.

4.5.3 Projeto funcional

As aplicações Web contemporâneas são radicalmente diferentes daquelas que existiam durante os primórdios da Internet, e possivelmente a maior mudança tem sido na incorporação de funcionalidade cada vez mais sofisticada e complexa. Ou seja, nos últimos anos foi notório o surgimento de conceitos e tecnologias – tais como Web 2.0, Ajax e serviços Web – que enfatizam o suporte para a funcionalidade rica e aplicações Web altamente interativas tais como *blogs*, *wikis* e aplicações de conteúdo dinâmico (exemplo: placar ao vivo de partidas de futebol).

Pressman & Lowe (2009) argumentam que devido às primeiras aplicações Web focarem principalmente o gerenciamento e acesso de informações, o foco do processo de projeto era naturalmente sobre projeto da informação. Quando o foco das aplicações Web passou para uma funcionalidade cada vez mais complexa, o projeto das funcionalidades tornou-se necessário. O desafio das

equipes de Engenharia Web atuais é conduzir o projeto funcional de tal forma que complemente outras atividades de projeto (por exemplo, projeto de interação e projeto de informação) de maneira imperceptível. O projeto funcional é quase sempre baseado em componentes (ver discussão sobre desenvolvimento baseado em componentes na Seção 4.5.4.2).

As funcionalidades disponibilizadas por uma aplicação Web podem ser divididas em dois conjuntos: (1) *funcionalidades em nível de usuário* expressam as funcionalidades que apoiam os usuários na obtenção de seus objetivos e (2) *funcionalidades em nível de aplicação* representam um projeto em nível mais baixo da funcionalidade interna, que pode não ser diretamente visível aos usuários. Dadas essas definições, é razoável afirmar que as funcionalidades em nível de usuário estão mais ligadas aos requisitos centrais da aplicação Web enquanto as funcionalidades em nível de aplicação estão profundamente embutidas na estrutura da aplicação Web e geralmente surgem do projeto detalhado e incremental das funcionalidades em nível de usuário.

O projeto funcional começa com o projeto da funcionalidade do usuário, derivada de uma definição dos objetivos do usuário (geralmente documentados por casos de uso durante a atividade de comunicação). Basicamente, começa-se considerando o modelo de análise e a arquitetura de informação inicial e depois examinando como a funcionalidade afeta: (1) a interação do usuário com a aplicação, (2) a informação que é apresentada e (3) as tarefas do usuário que são realizadas (PRESSMAN & LOWE, 2009).

O projeto de funcionalidade em nível de usuário será combinado com o projeto de informação para criar uma arquitetura funcional. Uma *arquitetura funcional* é uma representação do domínio funcional da aplicação Web e descreve os principais componentes funcionais e como esses componentes interagem uns com os outros. O projeto de funcionalidade em nível de aplicação define a funcionalidade de suporte interna à aplicação Web. Normalmente, ela vem após a funcionalidade do usuário e também afetará a arquitetura funcional. A Figura 16 ilustra a arquitetura funcional preliminar da aplicação **LivrariaVirtual.com.br**.

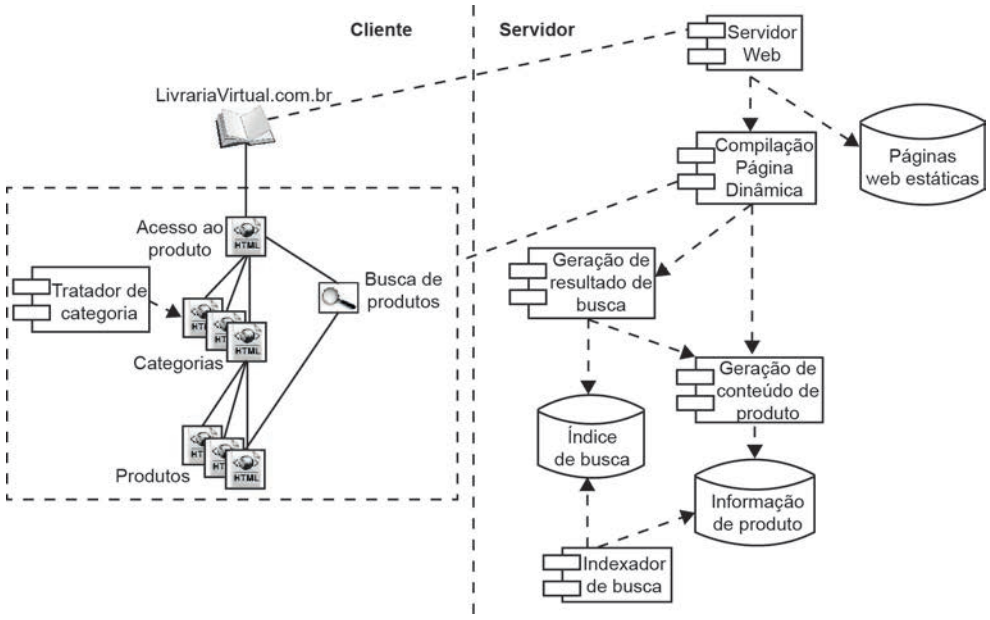


Figura 16 Arquitetura funcional preliminar da **LivrariaVirtual.com.br**.

Nessa arquitetura funcional, decidiu-se gerar dinamicamente as páginas de informação de produto. Nessa arquitetura em particular, a geração das páginas de acesso de produto, categoria (CD, DVD ou Livro) e informação de produto (que poderiam ter sido projetadas como parte da arquitetura de informação) são criadas dinamicamente pelo *componente de compilação de página dinâmica*. Este, por sua vez, usará o componente de *geração do resultado de busca* ou *geração de conteúdo de produto* para criar o conteúdo da página. Um componente (funcionalidade em nível de aplicação), o indexador de busca, será responsável pela geração dos índices de busca.

Categorias de funcionalidades. A escala, a complexidade e o tipo de funcionalidade incorporada em uma aplicação Web podem variar significativamente. Como consequência, a técnica da equipe de Engenharia Web para o projeto funcional deve ser ajustada aos tipos de funcionalidades que foram solicitados pelos interessados e identificados como parte da atividade de comunicação.

Pressman & Lowe (2009) apresentam alguns exemplos típicos de diferentes categorias de funcionalidades:

1. *Suporte à interação no lado do cliente*. Aplicações Web normalmente incorporam funcionalidades no lado do cliente voltadas a fornecer apoio à navegação do usuário e acesso à informação. Os exemplos mais comuns destes são menus *drop-down* e pré-carga de imagem. A Figura 17(a) apresenta um exemplo de menu *drop-down* – sítio Google Agenda (<https://www.google.com/calendar>).

2. *Suporte ao gerenciamento de informações no lado do cliente*. Aplicações Web em algumas situações implementam a manipulação complexa de informações no lado do cliente. Alguns exemplos incluem aplicações envolvendo manipulação complexa de informações, incluindo a pré-carga de dados e aplicações AJAX.[7] A Figura 17(b) apresenta um exemplo de manipulação de informações no lado do cliente – sítio Google Maps (<http://maps.google.com.br/maps>).

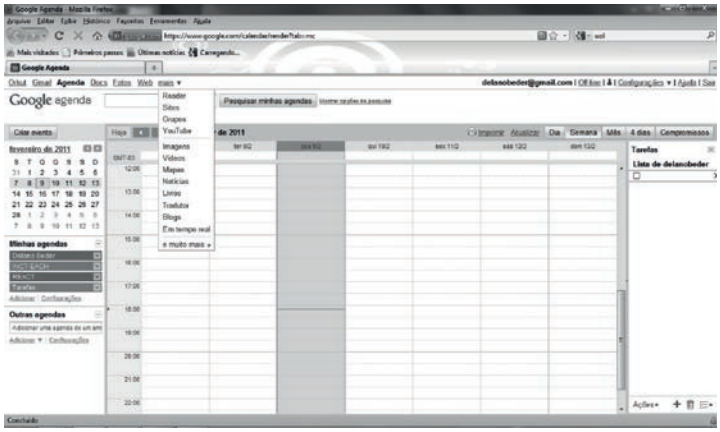
3. *Tratamento de conteúdo no lado do servidor*. Aplicações Web podem incorporar funcionalidades no lado do servidor para gerenciar conteúdo dinâmico e rapidamente modificável. Alguns exemplos aqui seriam quadros de discussão, *wikis* e sítios apresentando conteúdo de eventos ao vivo (por exemplo, sites de notícias, sites de esportes). A Figura 17(c) apresenta um exemplo de funcionalidade para lidar com conteúdo dinâmico – sítio UOL Esporte Placar (<http://placar.esporte.uol.com.br>).

4. *Gerenciamento de grandes conjuntos de dados no lado do servidor*. Aplicações Web podem usar funcionalidades no lado do servidor para gerenciar grandes conjuntos de dados complexos. Alguns exemplos incluem catálogos de produto, repositórios de documento, bibliotecas e listas pessoais. A Figura 17(d) apresenta um exemplo de busca em um repositório de produtos – sítio Livraria Cultura (<http://www.livrariacultura.com.br>).

5. *Suporte a processo e/ou fluxo de trabalho*. Aplicações Web normalmente implementam um processo computacional e transformacional específico, que apoia a entrada de dados procedimental, gerenciamento de transação ou fluxos de trabalho de negócios. Alguns exemplos são compras on-line, processos de cadastramento e pesquisas on-line. A Figura 17(e) apresenta um exemplo de processo de fluxo de trabalho – sítio Livraria Cultura (<http://www.livrariacultura.com.br>).

7 Acrônimo para *Asynchronous Javascript And* XML.

a)


Google agenda
Pesquisar minhas agendas
de 2011
Tarefas
Lista de delanobeder

b)


Google maps
Universidade Federal de São Carlos
Pesquisar no Mapa
Universidade Federal de São Carlos
Livros Universitários
Fundação Universidade Federal de São Carlos
Universidade Federal de São Carlos
Universidade Federal de São Carlos
Fundação Universidade Federal de São Carlos
Fundação Universidade Federal de São Carlos

c)


UOL Esporte - Placar
Futebol > Amistosos
França 0 x 0 Brasil
AO VIVO
Comente
ENTRAR

d)


livraria cultura
Engenharia Web (em Portugues) (2009)
COMPRAR
Engenharia De Software Orientado Para A Web (em Portugues) (2002)
COMPRAR
Refinamento

e)

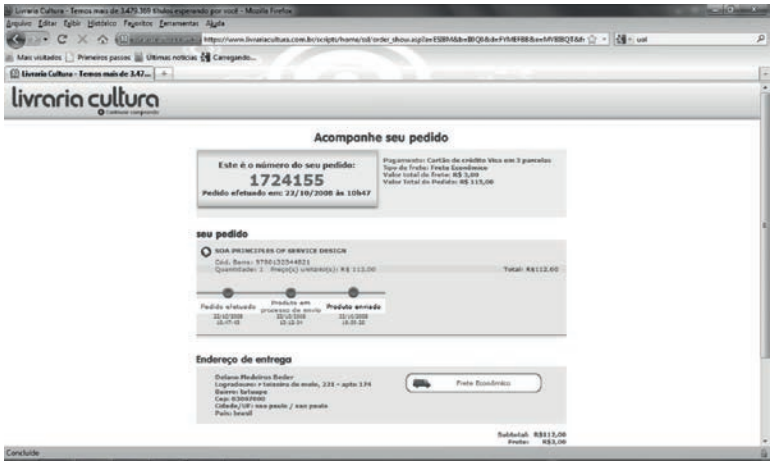


Figura 17 a) Menus *drop-down* como exemplo de suporte ao acesso a informações no lado do cliente; b) *Zoom* e rolagem de imagem como exemplo de manipulação de dados no lado do cliente; c) Atualizações do placar ao vivo como exemplo de funcionalidade para lidar com conteúdo dinâmico; d) Busca em um repositório de produtos como exemplo de gerenciamento de grandes conjuntos de dados; e) Um processo de fluxo de trabalho como exemplo de suporte ao processo e/ou fluxo de trabalho.

4.5.4 Projeto técnico

Os níveis mais inferiores da pirâmide de projeto – classificados como *projeto técnico* – consideram como a interface, a navegação, o conteúdo e as funções serão integrados e implementados. O projeto técnico oferece uma base para a atividade de construção que se segue. Ele trata tanto do projeto da arquitetura quanto do projeto dos componentes da aplicação Web.

4.5.4.1 Projeto de arquitetura

Projeto de arquitetura está relacionado aos objetivos estabelecidos para uma aplicação Web, o conteúdo a ser apresentado, os usuários visitantes, e a filosofia de navegação estabelecida. O projetista arquitetural deve identificar a arquitetura de conteúdo e a arquitetura da aplicação Web. *Arquitetura de conteúdo* concentra-se na maneira pela qual objetos de conteúdo (Seção 4.5.2.1), ou a composição destes, são estruturados para a apresentação e navegação. *Arquitetura da aplicação* trata o modo pelo qual a aplicação é estruturada para gerir a interação com o usuário, para prover funcionalidades, efetuar navegação e apresentar conteúdo (PRESSMAN, 2006).

Arquitetura de conteúdo. O projeto da *arquitetura de conteúdo* focaliza a definição da estrutura global de hipermídia da aplicação Web. A Figura 18 ilustra

quatro diferentes estruturas de informação que podem ser utilizadas no projeto da arquitetura de conteúdo (POWELL, 2000).

Estruturas lineares. Estruturas lineares são utilizadas quando uma sequência previsível de interações (com alguma variação ou desvio) é comum. Por exemplo, uma apresentação tutorial em que as páginas de informação, junto com gráficos relacionados, vídeos curtos ou áudios são exibidos apenas depois que a informação de pré-requisito tiver sido apresentada.

Estruturas de matriz (ou grade). Essa estrutura de informação pode ser aplicada quando o conteúdo da aplicação Web é organizado categoricamente em duas (ou mais) dimensões. Como exemplo da utilização dessa estrutura, considere uma aplicação Web[8] que vende calçados esportivos. A dimensão horizontal da matriz representa o tipo de calçado a ser vendido – específico para cada esporte. A dimensão vertical representa as ofertas fornecidas por vários fabricantes de calçados esportivos. Assim, um usuário pode navegar horizontalmente na matriz para encontrar o esporte desejado e, depois, verticalmente para examinar as ofertas dos fabricantes de calçados esportivos.

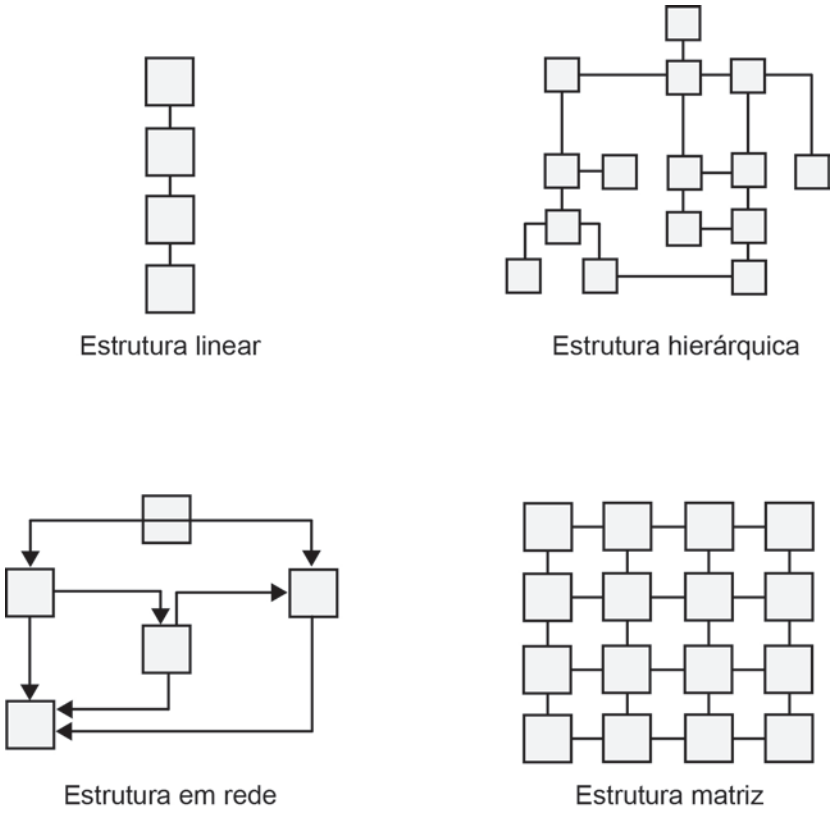


Figura 18 Estruturas de informação.

8 Exemplo adaptado de Pressman (2006).

Estruturas hierárquicas. Essa estrutura, sem dúvida, é a arquitetura de conteúdo mais comum, e é usada para refletir as taxonomias de informação naturais. A seleção da taxonomia correta é um desafio substancial, particularmente quando a informação não é homogênea. Além disso, uma estrutura hierárquica pode ser projetada de maneira que permita um fluxo horizontal, ao longo dos ramos verticais da estrutura. Deve-se notar que esse fluxo permite uma navegação rápida ao longo do conteúdo da aplicação Web. No entanto, pode causar confusão para os usuários.

Estruturas em rede. Esses elementos arquiteturais são compostos de links associativos que vinculam conceitos comuns ou relacionados, unidos dentro do espaço da informação. A estrutura em rede pode ser muito eficaz em aspectos relacionados à interconexão dentro de uma aplicação Web e permite aos usuários uma flexibilidade navegacional substancial. Se usada de forma criteriosa, uma estrutura em rede pode ajudar os usuários a navegarem produtivamente.

Pressman (2006) argumenta que as estruturas de informação discutidas nos parágrafos anteriores podem ser combinadas para formar estruturas compostas. A arquitetura de conteúdo de uma aplicação Web pode, por exemplo, ser hierárquica, mas parte da estrutura pode apresentar características lineares, enquanto outra parte da arquitetura pode estar em rede. O objetivo do projetista arquitetural é combinar a estrutura da aplicação Web com o conteúdo a ser apresentado e o processamento a ser realizado.

Arquitetura da aplicação. Arquiteturas de software fornecem uma descrição abstrata de um sistema por focar em sua estrutura e abstrair os detalhes de implementação.

Bass, Clements & Kazman (2003) ressaltam três vantagens de projetar explicitamente uma arquitetura de software:

1. *Comunicação entre os interessados*. A arquitetura é uma apresentação em alto nível do sistema que pode ser usada para enfocar a discussão entre os diferentes interessados.
2. *Análise do sistema*. Tornar a arquitetura da aplicação explícita requer alguma análise. Decisões de projeto de arquitetura têm profundo efeito sobre se o sistema atenderá os requisitos críticos, como desempenho, confiabilidade e facilidade de manutenção.
3. *Reúso em larga escala*. Um modelo de arquitetura de sistema é uma descrição completa e administrável de como um sistema está organizado e de como os componentes operam entre si. A arquitetura de sistema é muitas vezes a mesma para sistemas com requisitos similares e, assim, pode apoiar o reúso de software em larga escala (ver discussão sobre estilos e padrões arquiteturais adiante nesta seção).

Em geral, a descrição de uma arquitetura de software é baseada nos seguintes conceitos (SHAW & GARLAN, 1996):

- componentes lógicos que representam as unidades de computação e têm pontos de interface (portas de componentes com outros elementos arquiteturais);
- conectores que representam o padrão de composição entre os componentes e têm pontos de interface (papéis de conectores) com outros elementos arquiteturais. Um conector prescreve o protocolo de interação que ocorre entre os componentes que são compostos por meio dele;
- configuração que define a estrutura do sistema por compor uma coleção de instâncias de componentes conectados por meio de instâncias de conectores. Uma arquitetura de software é então definida como uma configuração que instancia componentes e conectores.

Isto é, uma *arquitetura de software* define o sistema em função de seus componentes computacionais e da interação entre esses componentes.

Muitos dos métodos tradicionais de desenvolvimento de software não incorporam um aspecto muito importante, que é a reutilização sistemática de conhecimento prévio sobre sistemas relacionados na definição da arquitetura de software de um sistema. Existem duas abordagens complementares que podem ser utilizadas na definição de arquiteturas de software: *estilos arquiteturais* e *padrões arquiteturais*.

A noção de *estilos arquiteturais* (SHAW & GARLAN, 1996) provê meios de explorar as semelhanças entre sistemas. Um estilo arquitetural define um conjunto de propriedades compartilhadas pelas arquiteturas de software que são membros deste estilo arquitetural.

Entre os exemplos de estilos arquiteturais encontrados na literatura pode-se citar estes:

- **Cliente & Servidor**. Nesse estilo, existem dois tipos de componentes: *clientes* e *servidores*. Chamadas de procedimentos remotos é o protocolo geralmente utilizado para prover comunicação entre esses componentes. Um sistema organizado de acordo com esse estilo é constituído de um conjunto de clientes que requisitam serviços (usando chamadas de procedimentos remotos ou outra forma de comunicação a servidores). A arquitetura Web é fortemente baseada nesse estilo arquitetural: clientes (navegadores) requisitam, utilizando o protocolo HTTP, páginas ou funcionalidades presentes em servidores web.

- **Camada**. Este estilo arquitetural define uma estrutura hierárquica em que cada camada provê serviços para a camada superior e usa os serviços da camada inferior na hierarquia. Um exemplo de arquitetura em camadas é o modelo OSI de protocolos de rede (ZIMMERMANN, 1980).

Segundo Buschmann et al. (1996), os *padrões arquiteturais* proveem um conjunto de subsistemas ou componentes predefinidos, especificando as suas responsabilidades, restrições topológicas, restrições sobre tipos de componentes e conectores, organizando, dessa forma, os relacionamentos e a comunicação entre eles.

Como exemplo de um padrão arquitetural bastante conhecido pela comunidade de desenvolvimento Web, pode-se citar o modelo-visão-controlador (MVC) (KRASNER & POPE, 1988), que desacopla a interface com o usuário da funcionalidade e do conteúdo da aplicação Web. Uma representação esquemática da arquitetura MVC aparece na Figura 19. Esse padrão arquitetural define três componentes (*Modelo*, *Visão* e *Controlador*) com características bem delineadas (PRESSMAN, 2006):

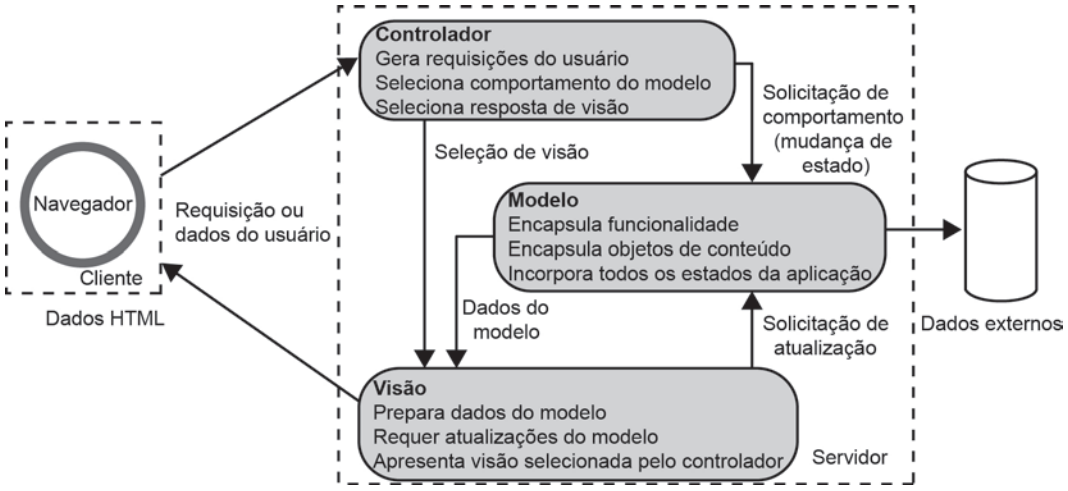


Figura 19 Padrão arquitetural MVC.

O *modelo* contém todo o conteúdo específico da aplicação e da lógica de processamento, incluindo todos os objetos de conteúdo, acesso a dados externos e fontes de informação, e todas as funcionalidades de processamento que são específicas da aplicação. No caso de sistemas que utilizam bases de dados, o modelo mantém o estado persistente do negócio e somente ele pode acessar as bases de dados.

A *visão* contém todas as funcionalidades específicas da interface que habilita a apresentação de conteúdo e lógica de processamento.

O *controlador* gerencia o acesso e a manipulação do modelo e da visão, bem como coordena o fluxo de dados entre eles. Em uma aplicação Web, o controlador monitora a interação com o usuário e, baseando-se nisso, recupera os dados do modelo e utiliza-os para atualizar ou construir a visão.

Referindo-se à figura, as solicitações ou dados do usuário são tratados pelo controlador que seleciona a visão apropriada com base na solicitação do usuário. Quando o tipo de solicitação é determinado, uma solicitação de comportamento é transmitida ao modelo, que implementa a funcionalidade ou recupera o conteúdo exigido para satisfazer a solicitação. O modelo pode acessar dados armazenados em um banco de dados corporativo, como parte de um armazenamento de dados local ou como uma coleção de arquivos independentes. Os dados devolvidos pelo modelo devem ser formatados e organizados pela visão apropriada e depois transmitidos do servidor de aplicação de volta ao navegador para exibição no navegador presente na máquina do cliente (PRESSMAN, 2006).

4.5.4.2 Projeto de componente

Aplicações Web contemporâneas disponibilizam uma série de funcionalidades sofisticadas, tais como: (1) processamento responsável por proporcionar que os conteúdos e as navegações sejam dinâmicos, (2) processamento de dados apropriados ao domínio da aplicação e (3) consulta e acesso a base de dados sofisticados.

Para proporcionar essas (e muitas) outras funcionalidades, a equipe Web deve projetar e construir componentes de programas análogos em estrutura aos componentes de software convencional.

O desenvolvimento de software baseado em componentes está além do escopo deste livro. Dessa forma, é apresentado no quadro a seguir apenas um breve sumário das principais características desse paradigma de desenvolvimento de software. Para mais detalhes, o leitor deve consultar Heineman & Council (2001) e Szyperski (2002).

O desenvolvimento baseado em componentes (DBC) pode ser visto como uma abordagem de desenvolvimento que tem como meta a construção e a reutilização de componentes de software. Ele permite a construção de sistemas por meio da reutilização de componentes de software que já foram bem especificados e testados, diminuindo os custos do processo de desenvolvimento e aumentando a produtividade e a qualidade do software (HEINEMAN & COUNCIL, 2001).

Os princípios fundamentais sobre componentes de software são os mesmos princípios que formam a base do modelo de objetos (WIRFS-BROCK, WILKERSON & WEINER, 1990): (1) unificação de dados e operações, (2) encapsulamento, em que um cliente de um componente não tem conhecimento acerca de como as operações são implementadas, (3) identidade única, pois cada componente possui uma única identidade que o representa independentemente de seu estado, (4) separação entre a especificação e a implementação de um componente de software e (5) divisão da especificação em várias interfaces.

A especificação dos componentes é geralmente dividida em duas partes: os serviços que um componente oferece são definidos em suas *interfaces providas* enquanto os serviços requeridos, serviços que o componente depende, são definidos em suas *interfaces requeridas*. A separação da especificação em diferentes interfaces aumenta a flexibilidade e adaptabilidade do componente, desde que um dado componente apenas conhece as interfaces requeridas, sem nenhum conhecimento dos componentes que implementam estas. Essa adaptação é realizada por meio de conectores que reconhecem as interfaces dos diferentes componentes e realiza a adaptação de uma interface para outra. Com essa separação, diminui-se o impacto e as dificuldades de modificações no sistema, visto que um componente de software pode ser trocado por outro, contanto que esse último implemente pelo menos as mesmas interfaces do componente original. Essas características permitem que um componente de software seja atualizado ou trocado, sem impacto sobre os clientes desse componente.

O projeto de componentes deve guiar a fase de construção/codificação (Unidade 5). No entanto, na realidade, não existem linguagens baseadas em componentes – o que mais se aproxima são as linguagens orientadas a objetos. Dessa forma, é necessário que cada componente seja elaborado com mais detalhes para guiar sua codificação em uma linguagem de programação existente.

Pressman & Lowe (2009) argumentam que no contexto de Engenharia de Software orientado a objetos, um componente contém um conjunto de classes colaborativas. Cada classe no escopo de um componente é totalmente elaborada para incluir todos os atributos e operações que são relevantes à sua

implementação. Como parte da colaboração do projeto, todas as interfaces que permitem que as classes se comuniquem e colaborem com outras classes do projeto também precisam ser definidas.

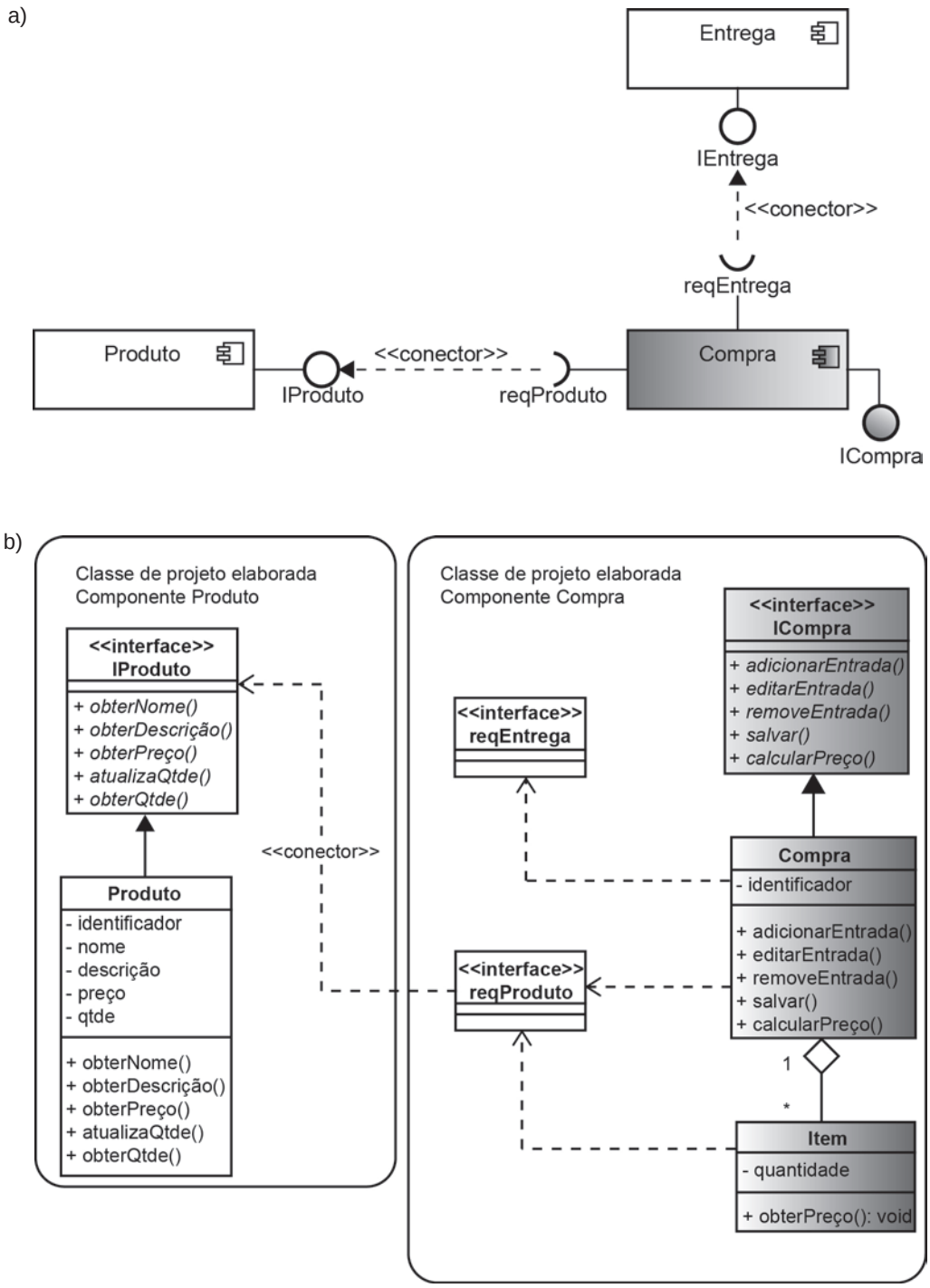


Figura 20 a) Projeto preliminar de componentes da **LivrariaVirtual.com.br**; b) Projeto detalhado (orientado a objetos) dos componentes.

A Figura 20(a) apresenta o projeto preliminar de componentes da aplicação **LivrariaVirtual.com.br**. Nessa figura, encontram-se três componentes:

- O componente *Compra* é uma abstração do carrinho de compras da aplicação. Ele disponibiliza suas funcionalidades por meio da interface provida *ICompra*. Para prover tais funcionalidades ele requer algumas funcionalidades providas pelos componentes *Produto* (representada pela interface requerida *reqProduto*) e *Entrega* (representada pela interface requerida *reqEntrega*). As interfaces providas e requeridas são conectadas (a conexão é representada pela dependência UML <<conector>>).
- O componente *Produto* é uma abstração dos produtos (livros, CDs e DVDs) vendidos pela aplicação. Ele disponibiliza suas funcionalidades por meio da interface provida *IProduto*.
- O componente *Entrega* é responsável pela entrega (nos respectivos endereços) dos produtos adquiridos por clientes. Ele disponibiliza suas funcionalidades por meio da interface provida *IEntrega*.

A Figura 20(b) apresenta o projeto detalhado preliminar de alguns componentes da aplicação **LivrariaVirtual.com.br**. Ou seja, essa figura apresenta um projeto orientado a objetos que fornece informação suficiente para guiar a construção (Unidade 5) dos componentes. A classe de projeto *Compra* contém informação detalhada necessária para implementar o componente *Compra*. As interfaces *reqProduto* e *reqEntrega* implicam comunicação e colaboração com outros componentes (o projeto detalhado do componente *Entrega* não é apresentado na figura).

É importante salientar que as classes de análise (Seção 4.4.1.3) e de projeto geralmente diferem. Ou seja, as classes de projeto são produzidas a partir das classes de análises originais, que são refinadas e possivelmente expandidas com informações necessárias para guiar a construção (Unidade 5) dos componentes. Por exemplo, a classe de projeto Produto (Figura 20(b)) consiste do refinamento da classe de análise Produto (Figura 8). Pode-se notar que a classe de projeto Produto possui um novo atributo *qtde* – abstração da quantidade de produtos em estoque – e novos métodos que acessam e atualizam esse atributo.

4.5.4.3 Padrões de projeto

Uma das vantagens do paradigma de orientação a objetos é o seu suporte à reusabilidade, que é a prática de incorporar componentes/módulos de

software já existentes em sistemas de software para os quais eles não foram originalmente desenvolvidos. No passado, a ideia de reusabilidade estava associada apenas à reutilização de código ou à invocação de rotinas de bibliotecas. Portanto, a reutilização de software era realizada na fase de codificação do sistema. Entretanto, a reutilização de código-fonte é uma forma muito limitada de reusabilidade. Mais benefícios podem ser obtidos quando a reusabilidade é aplicada também às fases de análise e projeto do desenvolvimento de software.

Por um lado, como discutido anteriormente, a noção de estilos e padrões arquiteturais provê meios de reutilizar sistematicamente o conhecimento prévio sobre sistemas relacionados na definição da arquitetura de software de um sistema. Por outro lado, padrões de projeto (GAMMA et al., 2000) procuram documentar conhecimentos e experiências de projetos existentes no intuito de ajudar na busca de soluções apropriadas para problemas de *projeto de componentes*.

Um padrão de projeto documenta uma alternativa de solução para um problema específico, recorrente em diversas aplicações. Ele tem uma estrutura e formato particular, e descreve um problema que ocorre em um domínio em particular e como resolver esse problema. Geralmente a escolha de um padrão de projeto é influenciada pelos estilos e/ou padrões arquiteturais escolhidos anteriormente.

Padrões de projeto refinam as decisões de projeto definidas na fase do *projeto de arquitetura*. O principal benefício decorrente do uso de padrões de projeto é facilitar a comunicação entre desenvolvedores de software, de uma mesma equipe ou independentes, ao permitir o emprego de estruturas de um nível de abstração maior do que linguagens de programação, porém com o mesmo grau de formalismo destas, contribuindo assim, positivamente, para a reutilização de software.

Uma descrição detalhada dos padrões de projeto existentes está além do escopo deste livro. Para mais detalhes, o leitor interessado deve consultar (GAMMA et al., 2000).

4.6 Considerações finais

Esta unidade apresentou os fundamentos da atividade de modelagem do Arcabouço de Processo de Engenharia Web. Como discutido, a atividade de modelagem abrange a criação de modelos que auxiliam o processo de desenvolvimento e é caracterizada por duas ações: modelagem de análise e modelagem de projeto.

O foco da modelagem de análise está no “o quê?”. Ou seja, entender o problema antes de tentar resolvê-lo. Já a modelagem de projeto foca no “como resolver o problema?”.

4.7 Estudos complementares

Esta unidade apresentou uma lista abrangente de tópicos. Dessa forma, não foi possível abordar detalhadamente todos estes. Portanto, nesta seção são apresentadas algumas referências para estudos complementares.

Atividade de modelagem. Para mais detalhes, o leitor interessado deve consultar:

PRESSMAN, R. S. Modelagem de análise para Aplicações da Web. In: _____. *Engenharia de Software*. 6. ed. São Paulo: McGraw-Hill, 2006.

_____. Modelagem de projeto para Aplicações da Web. In: _____. *Engenharia de Software*. 6. ed. São Paulo: McGraw-Hill, 2006.

PRESSMAN, R. S.; LOWE, D. A Atividade de Modelagem. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

_____. Modelagem de Análise para WebApps. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

_____. Projeto da WebApp. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

_____. Projeto da Interação. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

_____. Projeto da Informação. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

_____. Projeto Funcional. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

_____. Padrões de Projeto. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

Desenvolvimento baseado em componentes. Para mais detalhes, o leitor interessado deve consultar:

HEINEMAN, G. T.; COUNCIL, W. T. *Component-Based Software Engineering*: putting the pieces together. 1. ed. Boston: Addison-Wesley, 2001.

SZYPERSKI, C. *Component Software*: beyond object-oriented programming. 2. ed. Boston: Addison-Wesley, 2002.

Estilos e padrões arquiteturais. Para mais detalhes, o leitor interessado deve consultar:

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. *Pattern-Oriented Software Architecture*: a system of patterns. 1. ed. Hoboken: Wiley & Sons, 1996.

SHAW, M.; GARLAN, D. *Software Architecture*: perspectives on an emerging discipline. New Jersey: Prentice-Hall, 1996.

Padrões de projeto. Para mais detalhes, o leitor interessado deve consultar:

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Padrões de Projeto*: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.

Ou os repositórios de padrões listados a seguir:

Portland Pattern Repository. Disponível em: <http://c2.com/cgi/wiki?PortlandPatternRepository>. Acesso em: 23 jan. 2012.

Core J2EE patterns. Disponível em:<http://www.corej2eepatterns.com/Patterns2ndEd/index.htm>. Acesso em: 23 jan. 2012.

Interaction Design Pattern Library. Disponível em: <http://welie.com/patterns/>. Acesso em: 23 jan. 2012.

UNIDADE 5

Construção e implantação

5.1 Primeiras palavras

Finalizando a discussão sobre as principais atividades do Arcabouço de Processo de Engenharia Web, esta unidade discute duas atividades fortemente relacionadas: construção e implantação.

A atividade de *construção* combina a geração de código e o teste necessário para revelar erros no código enquanto a atividade de *implantação* consiste na entrega de um incremento ao cliente, o qual o analisa e oferece *feedback* com base na avaliação.

5.2 Problematizando o tema

Ao final desta unidade, espera-se que o leitor seja capaz de reconhecer e distinguir precisamente os conceitos básicos relacionados às atividades de construção e implantação do Arcabouço de Processo da Engenharia Web. Dessa forma, esta unidade pretende discutir as seguintes questões:

- Quais são os objetivos da atividade de construção do Arcabouço de Processo de Engenharia Web?
- Quais são as principais ações realizadas durante a atividade de construção?
- Quais são os objetivos da atividade de implantação do Arcabouço de Processo de Engenharia Web?
- Quais são as principais ações realizadas durante a atividade de implantação?

5.3 Construção

Ao fim de uma atividade de comunicação minuciosa, um planejamento atencioso e uma modelagem detalhada, um incremento da aplicação Web é construído e implantado. O resultado é um incremento operacional que está disponível aos usuários finais. É importante observar que grande parte da construção e implantação da aplicação Web está profundamente entrelaçada com as tecnologias, as ferramentas e as linguagens em particular que são usadas durante a construção. E cada uma dessas tecnologias de Engenharia Web é distinta e evolui rapidamente. Embora as atividades de construção e implantação sejam conceitualmente diferentes, elas estão profundamente interligadas. Dessa forma, justifica-se o estudo dessas duas atividades em uma mesma unidade.

Figura 21 Construção.

A atividade de construção compreende seleção, codificação, autoria de páginas, integração, refatoração e ações de teste. Quando completadas, essas ações levam a uma aplicação Web operacional que está pronta para entrega aos usuários finais (PRESSMAN & LOWE, 2009):

- a *seleção* envolve a identificação de componentes preexistentes, que podem ser reutilizados dentro do projeto proposto;
- a *codificação* envolve a adaptação de componentes existentes ou a criação de novos componentes, e pode envolver a criação de código HTML ou código-fonte em uma linguagem de programação;
- a *autoria de páginas* envolve a integração do conteúdo com o projeto gráfico (leiaute) e o mapeamento do conteúdo nas páginas. Também pode incluir a implantação de folhas de estilo;
- a *integração* compreende o vínculo do código, conteúdo e apresentação nos componentes finais liberados para uso;
- a *refatoração* é o ato de alterar o código sem afetar a funcionalidade que ele implementa. É utilizado para melhorar sua estrutura, esclarecer e remover o código redundante;
- finalmente, o *teste* envolve a verificação de que os diversos componentes estão corretos. Devido sua importância para o processo de Engenharia Web, o teste é considerado separadamente na Seção 5.4.

Por fim, Pressman & Lowe (2009) argumentam que a *construção* é uma atividade baseada em componentes. O desenvolvimento de software baseado em componentes está além do escopo deste livro. Na Unidade 4 foi apresentado um breve sumário das principais características desse paradigma de desenvolvimento de software. O leitor interessado deve consultar Heineman & Council (2001) e Szyperski (2002) para mais detalhes.

5.4 Testes de sistemas Web

Nesta seção pretende-se dar uma visão das atividades de teste de aplicações Web. Essas atividades levam em consideração os tipos de teste e as técnicas que podem ser usadas, as estratégias mais adequadas e as características inerentes (segurança, desempenho etc.) às aplicações Web. Por fim, as ferramentas que podem auxiliar no teste de aplicações Web são apresentadas. O texto presente nesta seção é baseado no trabalho de Delamaro, Chaim & Vincenzi (2010).

5.4.1 Teste de requisitos funcionais

Analogamente à maioria dos sistemas de software, é necessário que sejam testados os requisitos funcionais e não funcionais das aplicações Web.

As técnicas comumente utilizadas no teste de requisitos funcionais de aplicações Web são testes caixas brancas e caixas-pretas. Testes caixas brancas são baseados em um exame rigoroso do detalhe procedimental. Caminhos lógicos internos ao software e colaborações entre componentes são testados, definindo-se casos de testes que exercitam conjuntos específicos de condições e/ou ciclos (PRESSMAN, 2006). Testes caixas-pretas referem-se a testes que são conduzidos na interface do software. Um teste caixa-preta examina algum aspecto fundamental do sistema, pouco se preocupando com a estrutura interna do software (PRESSMAN, 2006). Apesar de poder ser utilizado em qualquer nível – teste unitário, integração ou de sistema – geralmente testes caixas brancas são utilizados para os testes unitários. Por outro lado, testes caixas-pretas são normalmente utilizados para testes de sistema e integração (DELAMARO, CHAIM & VINCENZI, 2010).

Uma breve conceituação dos termos – teste unitário, de integração e de sistema – é apresentada neste quadro:

- O teste unitário foca um componente individual de software exercitando a interface, as estruturas de dados e a funcionalidade do componente, em um esforço para descobrir erros que são locais ao componente.
- O teste de integração foca no projeto e na integração dos componentes do software. O objetivo é assegurar que a estrutura do software, quando suas unidades estão integradas, está adequada e as interações entre essas unidades funcionam corretamente.
- Durante o teste de sistema, procura-se garantir que todos os requisitos especificados para o software estão corretamente implementados. Isso inclui requisitos funcionais e não funcionais (desempenho, segurança etc.).

5.4.2 Teste de camadas

Conforme foi discutido na Unidade 4, a arquitetura típica de uma aplicação Web pode ser dividida em camadas: apresentação, negócios e acesso aos dados. Em uma arquitetura MVC, o acesso aos dados poderia ser o **M**odelo, a apresentação poderia ser a **V**isão e a camada de negócios poderia ser o **C**ontrolador. Devido à divisão em camadas, os testes podem ser concentrados em cada uma das camadas.

Durante o teste de uma camada, os objetos da camada inferior são substituídos por *dublês de teste* (MESZAROS, 2007). Os dublês de teste podem ser de vários tipos, a saber (DELAMARO, CHAIM & VINCENZI, 2010):

- objetos burros (*dummy*): utilizados somente para preencher uma lista de parâmetros e nunca são usados;
- objetos falsos (*fake*): possuem uma implementação simplificada, por exemplo, com valores fixos;
- objetos substitutos (*stub*): são também objetos falsos com implementação simplificada, mas que possuem funcionalidade adicional, como registrar as chamadas de seus métodos;
- objetos imitadores (*mock*): são os dublês mais complexos, pois podem ser programados para receber chamadas específicas, tratar chamadas inesperadas e também verificar se todas as chamadas previstas foram recebidas.

É importante mencionar que a estratégia discutida anteriormente não é exclusiva das aplicações Web, podendo ser aplicada em outros tipos de sistemas que utilizam uma arquitetura em camadas. A seguir, são discutidas estratégias específicas aplicadas ao teste de cada uma das camadas de uma aplicação Web.

Teste do acesso aos dados. A camada de acesso aos dados geralmente é construída utilizando DAOs (Data Access Objects). Existem duas estratégias para testar DAOs (DELAMARO, CHAIM & VINCENZI, 2010):

- A primeira utiliza a base de dados real como se fosse a própria aplicação. A base de dados pode ser a base de dados local do desenvolvedor ou uma base de dados dedicada aos testes.
- A segunda é baseada em objetos imitadores. A vantagem é que não precisa utilizar a base de dados real. No entanto, é necessária a implementação de objetos imitadores, o que implica em um custo adicional de programação.

Teste da lógica de negócios. Delamaro, Chaim & Vincenzi (2010) considera que esse teste é similar ao teste de outros tipos de software. Na maioria dos casos, são utilizados objetos imitadores que fazem o papel da camada de acesso aos dados, ou seja, são criados imitadores DAO. No entanto, é possível também utilizar DAOs reais que acessam a camada de dados. Nesse caso, Delamaro, Chaim & Vincenzi (2010) argumenta que é realizado o teste de integração (das camadas de negócio e de acesso aos dados), bem como o teste das regras de negócio.

Teste da camada de apresentação. A camada de apresentação geralmente é realizada utilizando a linguagem de marcação HTML. Páginas nesse formato são enviadas ao cliente por meio do protocolo HTTP. Navegadores apresentam as páginas enviadas ao usuário final que interage por meio delas. Ferramentas como HTTPUnit, Selenium e HTMLUnit (ver discussão sobre ferramentas na Seção 5.4.4) aproveitam o conhecimento da linguagem HTML para simular as ações do usuário como, por exemplo, preencher um campo ou apertar um botão. Dessa maneira é possível desenvolver programas ou *scripts* que executem e validem os casos de testes da camada de apresentação (DELAMARO, CHAIM & VINCENZI, 2010).

5.4.3 Teste de requisitos não funcionais

Testes que verificam requisitos não funcionais como o teste de desempenho, de carga e de estresse são fundamentais em aplicações Web.

- O teste de desempenho tem como objetivo produzir dados que permitam prever o desempenho da aplicação quando diferentes níveis de carga (e.g., número de requisições) são submetidos. Em especial, espera-se identificar os níveis de carga que provocam a exaustão dos recursos do sistema.
- O teste de carga é semelhante ao teste de desempenho, porém visa verificar como a aplicação se comporta quando submetida a um grande volume de dados, cálculos e processamento intensivo.
- O teste de estresse força a aplicação a operar em condições de recursos reduzidos. O objetivo é verificar o comportamento da aplicação em situações acima dos limites estabelecidos para garantir que a aplicação funciona adequadamente ou que falhas sejam tratadas adequadamente sem perda ou corrupção dos dados.

Delamaro, Chaim & Vincenzi (2010) sugerem que a técnica de teste mais indicada para os testes de desempenho, carga e estresse é a técnica de teste caixa cinza. Analogamente ao teste caixa-preta, o foco desse teste é verificar se a saída obtida corresponde à saída esperada. O diferencial é que, nesse caso, o testador utiliza seu conhecimento sobre as estruturas de dados, os algoritmos utilizados, o ambiente de execução e arquitetura da aplicação para desenvolver os testes. No entanto, é importante salientar que o testador não necessita conhecer o código-fonte da aplicação Web.

Outros requisitos não funcionais como segurança, usabilidade e compatibilidade não podem ser negligenciados no desenvolvimento de aplicações Web. No entanto, conforme argumenta Delamaro, Chaim & Vincenzi (2010), para o teste desses requisitos não existem técnicas de teste em particular. Há estratégias como inspeção do código para detecção de falhas de segurança, avaliação do uso da aplicação para identificação de falhas de usabilidade e ergonomia e avaliação da compatibilidade da aplicação com diferentes navegadores.

5.4.4 Ferramentas

Nesta seção são apresentadas algumas ferramentas que auxiliam o teste de aplicações Web. Todas as ferramentas apresentadas são de código-fonte aberto (*open-source*).

As ferramentas para teste unitário são genericamente referidas como ferramentas *xUnit*. A letra *x* é uma referência à linguagem alvo, que pode ser Java, C, PHP etc. A mais famosa dessas é a que apoia o teste de programas escritos em Java, conhecida com JUnit;[9] as demais ferramentas funcionam de maneira semelhante. O principal objetivo dessas ferramentas é a automatização de testes unitários que consiste na execução automática de um conjunto de testes unitários e à avaliação dos resultados gerados.

HttpUnit[10] é uma ferramenta Java que empregada em conjunto com JUnit é utilizada para executar casos de testes que verificam a funcionalidade de uma página Web. Operações como preencher um formulário e clicar em um botão são fornecidas. Além disso, HttpUnit permite que as páginas retornadas sejam comparadas com as saídas esperadas, possibilitando a validação automática dos testes. Os testes utilizando essa ferramenta são programas Java que invocam as operações fornecidas pela ferramenta. HtmlUnit[11] é uma ferramenta Java similar a HttpUnit em relação ao conjunto de funcionalidades.

Selenium[12] corresponde a um conjunto de ferramentas voltadas à automatização do teste de aplicações Web. Seu conjunto de ferramentas é mais completo que HttpUnit e HtmlUnit, visto que, além de fornecer uma biblioteca para programação dos casos de testes, permite também que eles sejam registrados na forma de *scripts* com uma ferramenta de captura e reexecução. Os *scripts* podem ser gravados em várias linguagens como Java, Groovy, entre outras. As principais ferramentas de Selenium são as descritas a seguir (DELAMARO, CHAIM & VINCENZI, 2010):

- **Selenium Remote Control (SRC)**: ferramenta responsável pela criação e execução de testes para aplicações Web. Os *scripts* descrevem ações que devem ser realizadas na aplicação sob teste.
- **Selenium IDE**: extensão do navegador Mozilla Firefox para gravação e reprodução de teste. Essa ferramenta corresponde a um ambiente gráfico no qual é gravada a interação usuário-navegador, gerando assim um *script* executável em SRC. A ferramenta também possui a funcionalidade de executar o *script* gerado.

9 Disponível em: <http://www.junit.org>. Acesso em: 23 jan. 2012.
10 Disponível em: <http://httpunit.sourceforge.net>. Acesso em: 23 jan. 2012.
11 Disponível em: <http://htmlunit.sourceforge.net>. Acesso em: 23 jan. 2012.
12 Disponível em: <http://seleniumhq.org>. Acesso em: 23 jan. 2012.

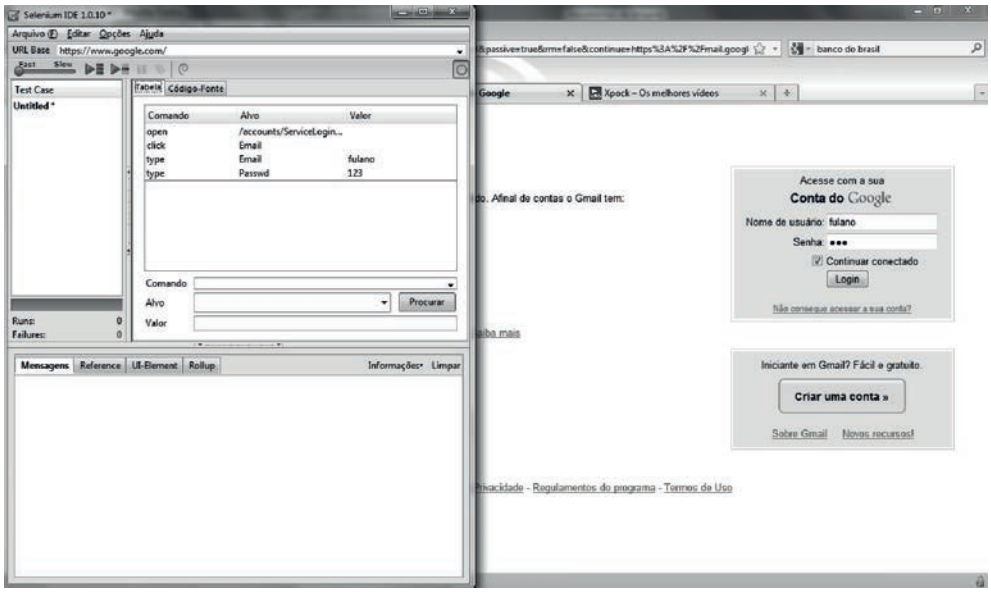


Figura 22 *SeleniumIDE* – Exemplo de captura e de reexecução.

Como exemplo[13] da utilização do *SeleniumIDE*, a Figura 22 apresenta o sítio *Google Gmail* para o qual foi fornecida a cadeia de caracteres “fulano” no campo *Nome de usuário* e “123” no campo *Senha*. No canto esquerdo, podemos observar que a ferramenta *SeleniumIDE* registra todos os comandos emitidos pelo testador. A Figura 23 apresenta um programa Java que consiste no *script* capturado automaticamente pela *SeleniumIDE*. É importante observar que esse *script* pode ser executado automaticamente sem que ocorra intervenção humana. Além disso, não há nenhum empecilho que o programa Java seja criado por um programador sem a utilização da *SeleniumIDE*, pois a biblioteca utilizada nesse programa é parte do pacote *Selenium RC*.

13 Exemplo baseado no apresentado em Delamaro, Chaim & Vincenzi (2010).

```
package com.example.tests;
import com.thoughtworks.selenium.*;
import java.util.regex.Pattern;

public class Untitled extends SeleneseTestCase {
  public void setUp() throws Exception {
       setUp(“https://www.google.com/”, “*chrome”);
  }
  public void testUntitled() throws Exception {
       selenium.open(“/accounts/ServiceLogin?service=mail “);
       selenium.click(“Email”);
       selenium.type(“Email”, “fulano”);
       selenium.type(“Passwd”, “123”);
  }
}
```

Figura 23 Código do *script* Java.

Httperf[14] é uma ferramenta para análise quantitativa de servidores Web e tem como objetivo principal fornecer estatísticas relativas ao desempenho do servidor Web, medindo a sua capacidade de atendimento a requisições.

Apache JMeter[15] é outra ferramenta para a avaliação de desempenho de sítios Web. JMeter permite aos usuários criar cenários reais e executá-los concorrentemente em múltiplas linhas de execução (*threads*).

5.5 Implantação

O principal objetivo da atividade de implantação é configurar a aplicação Web em seu ambiente operacional e disponibilizá-la aos usuários finais para que possa iniciar o período de avaliação. O *feedback* da avaliação é apresentado à equipe de Engenharia Web e o incremento é modificado conforme a necessidade.

14 Disponível em: <http://www.hpl.hp.com/research/linux/httperf>. Acesso em: 24 jan. 2012.
15 Disponível em: <http://jakarta.apache.org/jmeter/index.html>. Acesso em: 24 jan. 2012.

Figura 24 Implantação.

Dessa forma, a atividade de implantação (Figura 24) compreende duas ações: entrega (empacotamento + liberação) e avaliação. Como o desenvolvimento da aplicação Web é incremental por natureza, a implantação acontece não uma vez, mas diversas vezes enquanto a aplicação Web se encaminha para o término de seu desenvolvimento (PRESSMAN & LOWE, 2009).

Pressman & Lowe (2009) argumentam que em quase todos os casos é melhor empacotar um grupo de mudanças e liberá-las como um conjunto (exceto onde uma mudança estiver corrigindo um erro prejudicial), pois se fossem liberadas muitas mudanças pequenas e contínuas, os usuários poderiam ficar confusos e a probabilidade de erros de integração e efeitos colaterais não intencionais aumentaria.

Cada ciclo de liberação de pacote oferece aos usuários finais um incremento operacional da aplicação Web que fornece funcionalidades e recursos úteis. Cada ciclo de avaliação oferece à equipe de Engenharia Web orientações importantes que resultam em modificações no conteúdo, funcionalidades, características e enfoque usados nesse ou no próximo incremento.

Ambientes de desenvolvimento. É importante observar que ambientes de desenvolvimento de Engenharia Web possuem um efeito significativo sobre quando e como diversos incrementos da aplicação Web e seus componentes relacionados são construídos e implantados.

Pressman & Lowe (2009) argumentam que em aplicações Web muito simples e que não são de missão crítica, pode ser aceitável ter um único desenvolvedor criando conteúdo local. O conteúdo é então publicado no servidor de produção para acesso imediato pelos usuários.

No entanto, para aplicações Web mais complexas (em que a qualidade da aplicação é crucial), um ambiente de desenvolvimento mais complexo torna-se apropriado. Nesse caso, Pressman & Lowe (2009) sugerem um ambiente de desenvolvimento possível, contendo quatro tipos de servidores distintos:

- Servidores de desenvolvimento. Desenvolvedores e autores utilizam esses servidores para realizar toda a autoria de páginas e testes unitários. Esses servidores, na realidade, são caixas de areia em que os desenvolvedores podem criar, manipular e testar seus próprios componentes da aplicação Web.
- Servidor de teste. Quando os desenvolvedores completam o teste unitário em seus componentes, eles podem integrá-los para verificação dentro do ambiente completo da aplicação Web.
- Servidor de homologação. Esse servidor tem como finalidade oferecer um espelho completo do ambiente de produção, de modo que o teste abrangente do usuário da aplicação Web possa ocorrer sem ter que liberar a aplicação ao servidor de produção e, daí, à população de usuários completa.
- Servidor de produção. Quando a aplicação Web está pronta para ser liberada para uso por todos os usuários, ela é levada para esse servidor. É importante salientar que é crucial que as mudanças não sejam liberadas nesse servidor até que sejam totalmente testadas no servidor de homologação.

Finalizando essa discussão, o ambiente real a ser adotado para um projeto de aplicação Web em particular dependerá de uma série de fatores: (1) escala do projeto, (2) número de desenvolvedores e interessados e (3) estado crítico de negócios. Em muitos casos, os quatro diferentes servidores esboçados poderiam ser reduzidos para três servidores (desenvolvimento, homologação e produção) ou ainda dois servidores (desenvolvimento e produção). Apesar dessas variações, Pressman & Lowe (2009) sugerem que alguns princípios básicos sejam obedecidos:

- Mantenha o ambiente de desenvolvimento e de produção separados. Não desenvolva diretamente nos servidores que são acessíveis aos seus usuários.
- Forneça aos desenvolvedores um ambiente que facilite sua produção.
- Quando for possível, realize o teste no mesmo ambiente que os seus usuários acessarão.

5.6 Considerações finais

Esta unidade apresentou os fundamentos das atividades de construção e implantação do Arcabouço de Processo de Engenharia Web. Como discutido, essas atividades são o resultado final de um, e possivelmente muitas iterações de processo de Engenharia Web. Depois da comunicação minuciosa, do planejamento atencioso e da modelagem de engenharia detalhada, a equipe de Engenharia Web constrói e implanta um incremento da aplicação Web. O resultado visível é um incremento funcional que está disponível aos usuários finais.

5.7 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

PRESSMAN, R. S. Teste de Aplicações Web. In:_____. *Engenharia de Software*. 6. ed. São Paulo: McGraw-Hill, 2006.

PRESSMAN, R. S.; LOWE, D. Construção e Implantação. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

UNIDADE 6

Atividades guarda-chuva

6.1 Primeiras palavras

Finalizando a discussão sobre as atividades do Arcabouço de Processo de Engenharia Web, esta unidade discute as atividades guarda-chuva que ocorrem em segundo plano. Desde que essas atividades são igualmente importantes, uma equipe de desenvolvimento deve considerá-las explicitamente. Esta unidade discute quatro atividades fundamentais para um projeto bem-sucedido de Engenharia Web: gerência de mudança, gerência de qualidade, gerência de riscos e gerência de projeto.

6.2 Problematizando o tema

Ao final desta unidade espera-se que o leitor seja capaz de reconhecer e distinguir precisamente os conceitos básicos relacionados às atividades guarda-chuva do Arcabouço de Processo da Engenharia Web. Dessa forma, esta unidade pretende discutir as seguintes questões:

- O que são atividades guarda-chuva do Arcabouço de Processo de Engenharia Web?
- Quais são os objetivos das atividades guarda-chuva?
- Quais são as principais ações realizadas durante as atividades guarda-chuva?

6.3 Atividades guarda-chuva

Enquanto as atividades de arcabouço discutidas nas unidades anteriores são aplicadas a cada incremento da aplicação Web, uma coleção de atividades guarda-chuva, igualmente importantes para o sucesso do projeto, ocorre em segundo plano.

Entre as muitas atividades guarda-chuva que podem ser definidas, quatro atividades são fundamentais para um projeto bem-sucedido de Engenharia Web (PRESSMAN & LOWE, 2009):

- Gerência de riscos. Considera os riscos de projeto e técnicos à medida que um incremento é desenvolvido. Os fundamentos da gerência de riscos foram brevemente discutidos na Seção 3.3.1.
- Gerência de projeto. Acompanha e monitora o progresso à medida que um incremento é desenvolvido. Os fundamentos da gerência de projetos foram brevemente discutidos na Seção 3.3.4.

- Gerência de mudança. Gerencia o efeito da mudança à medida que cada incremento é desenvolvido, integrando ferramentas que auxiliam na gerência de todo o conteúdo da aplicação Web.
- Garantia da qualidade. Define e conduz aquelas tarefas que ajudam a garantir que cada produto de trabalho e o incremento implantado apresentam qualidade.

6.3.1 Gerência de mudanças

Desde que a incerteza é algo inerente à maioria dos projetos de Engenharia Web, as mudanças nos requisitos não são raras. A mudança é um empecilho desde que ela geralmente interrompe o fluxo normal do processo (a entrega de um incremento é adiada para acomodar a mudança), absorve recursos (tarefas normais são temporariamente abandonadas para atender às mudanças) e tira o foco da equipe. No entanto, a mudança é inevitável e normalmente construtiva. O que fazer para gerenciar as mudanças?

Felizmente, a estratégia de desenvolvimento incremental, definida pelo Arcabouço de Processo de Engenharia Web, auxilia uma equipe a gerenciar a mudança. Desde que o tempo de desenvolvimento para um incremento é curto, é possível adiar a introdução das mudanças solicitadas para o próximo incremento. Porém, essa estratégia implica em uma mudança sutil no desenvolvimento de cada incremento. Um incremento da aplicação Web não apenas implementa conteúdo e funcionalidade inerente a ele, mas também pode incorporar as mudanças no conteúdo e na funcionalidade solicitados para o incremento anterior.

A estratégia de retardar a mudança para o próximo incremento funciona na maioria dos casos. No entanto, existem algumas situações em que uma mudança deve ser feita imediatamente.

Pressman & Lowe (2009) também sugerem que a mudança deverá ser feita imediatamente, como parte do incremento atual, se um ou mais dos critérios a seguir forem atendidos:

1. O atraso da mudança resulta em mais trabalho do que fazê-la no ato.
2. A usabilidade do incremento pelos usuários finais for seriamente prejudicada sem a mudança.
3. Danos financeiros significativos ocorrerão se a mudança não for feita imediatamente.
4. Requisitos regulamentares ou legais exigirem que o incremento inclua a mudança.

Impacto de uma mudança. Outro aspecto fundamental na gerência de mudanças está relacionado a como é avaliado o impacto de uma mudança. Pressman (2006) sugere que cada mudança solicitada seja categorizada em uma de quatro classes:

Classe 1. Mudança de conteúdo ou de funcionalidade que corrige um pequeno erro ou melhora o conteúdo ou uma funcionalidade local.

Classe 2. Mudança de conteúdo ou de funcionalidade que tem um impacto sobre outros objetos de conteúdo ou sobre componentes funcionais dentro do incremento.

Classe 3. Mudança de conteúdo ou de funcionalidade que tem um grande impacto em uma aplicação Web (por exemplo, extensão de funcionalidade importante, melhoria ou redução significativa no conteúdo e mudanças importantes na navegação).

Classe 4. Mudança de projeto importante (por exemplo, uma mudança no projeto da interface ou na técnica de navegação) que será imediatamente observável a uma ou mais categorias de usuários finais.

Quando a mudança solicitada tiver sido categorizada, ela poderá ser avaliada de acordo com o algoritmo apresentado na Figura 25.

As mudanças de classe 1 e 2 são tratadas informalmente de uma maneira ágil. Para uma mudança de classe 1, deve-se avaliar o impacto da mudança sobre os itens de configuração[16] que serão afetados, mas nenhuma revisão ou documentação externa é exigida. O item de configuração a ser alterado é retirado (*check-out*) do sistema de controle de versões, a mudança é realizada e as atividades apropriadas de garantia da qualidade são aplicadas. A seguir, o item de configuração é devolvido (*check-in*) ao sistema de controle de versões, e os mecanismos apropriados são usados para construir a próxima versão do software.

Para as mudanças de classe 2, a análise do impacto da mudança sobre os itens de configuração relacionados fica sob a responsabilidade de um membro da

16 Ver discussão sobre controle de versões adiante nesta seção.

equipe de Engenharia Web. Se a mudança puder ser feita sem exigir mudanças significativas em outros itens, a modificação ocorre sem análise ou documentação adicional. Se forem exigidas mudanças substanciais, mais avaliação e planejamento são necessários.

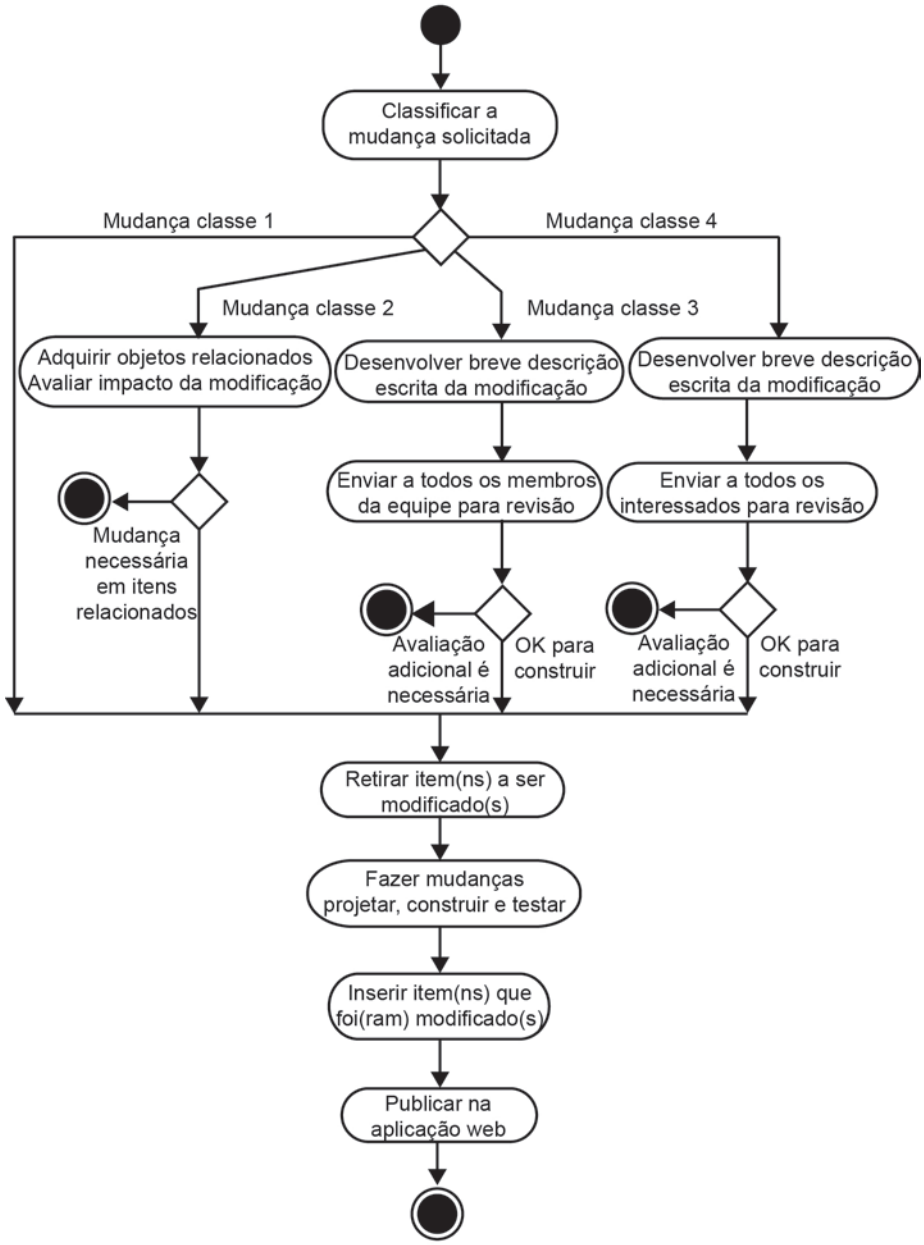


Figura 25 Gestão de configuração.

As mudanças de classe 3 e 4 também são tratadas de uma maneira ágil; no entanto, alguma documentação descritiva e procedimentos de análise mais formais são exigidos. Uma descrição de mudança, descrevendo a alteração e fornecendo uma breve avaliação do seu impacto, é desenvolvida para as mudanças de classe 3. A descrição é distribuída a todos os membros da equipe de Engenharia Web, que a analisam para avaliar melhor o seu impacto. Uma descrição

da mudança também é desenvolvida para mudanças de classe 4, mas, nesse caso, a análise é realizada por todos os interessados.

Controle de Versões. Embora apenas uma versão única da aplicação Web esteja disponível aos usuários finais, outras versões podem existir. Por exemplo, a organização decide arquivar uma versão mais antiga ou desenvolver uma versão alternativa que contém estética diferente e novas funcionalidades.

Um *Sistema de Controle de Versão* (PRESSMAN & LOWE, 2009) implementa ou é integrado diretamente a seis funcionalidades principais:

1. Repositório de projeto que armazena todos os itens de configuração relevantes. No contexto deste livro, um item de configuração é qualquer artefato de software (código-fonte Java, página HTML, modelo de análise etc.) produzido como parte do processo de Engenharia Web e que necessita ser gerenciado pelo sistema de controle de versões.
2. Gerenciamento de versão que armazena todas as versões de um item de configuração (ou permite que qualquer versão seja construída usando diferenças das versões passadas).
3. Controle de acesso que assegura quais membros da equipe de Engenharia Web (ou outros interessados) estão autorizados a acessar e modificar um item de configuração em particular.
4. Controle de sincronismo que ajuda a garantir que mudanças paralelas, realizadas por duas pessoas diferentes, não estarão em conflito.
5. Uma ferramenta de *make*, que permite coletar todos os itens de configuração relevantes e construir uma versão específica da aplicação Web.
6. E, por fim, um sistema de controle de versão que normalmente implementa uma ferramenta de acompanhamento de problemas, que permite que a equipe registre e acompanhe o *status* de todos os problemas pendentes associados a cada item de configuração.

6.3.2 Gerência de qualidade

Após um incremento de aplicação Web ser entregue aos usuários finais, ele é experimentado e o *feedback* é fornecido à equipe de Engenharia. Esse *feedback* oferece uma boa indicação do que está funcional e do que precisa de ajustes. Dessa forma, parece uma boa alternativa apenas considerar a qualidade da aplicação Web quando o incremento é implantado, resolvendo os problemas quando eles forem apontados pelos usuários finais. No entanto, conforme apontado por Presmann & Lowe (2009), existem diversas falhas nessa abordagem:

1. Problemas de qualidade encontrados pelos usuários finais quase sempre são mais dispendiosos para se corrigir do que teriam sido se fossem descobertos anteriormente no fluxo de processo de Engenharia Web. Por exemplo, reparar um erro de modelagem antes que a aplicação Web seja construída pode ser muitas vezes menos dispendioso (em esforço gasto) do que tentar corrigir uma aplicação já implementada.
2. Cada problema descoberto pelos usuários finais exige retrabalho, e o retrabalho absorve recursos (pessoas e esforço) que teriam sido aplicados ao próximo incremento. O projeto geral fica atrasado.
3. Os primeiros usuários podem não testar o incremento totalmente, deixando conteúdos e funcionalidades não exercitados.

Dessa forma, a abordagem mais apropriada é que a equipe de Engenharia Web considere a qualidade da aplicação Web à medida que os incrementos estão sendo desenvolvidos.

Problemas de qualidade surgem de diversas fontes. No entanto, suas origens normalmente podem ser rastreadas desde uma falta de entendimento até uma falta de atendimento das necessidades dos usuários finais. Portanto, o primeiro mecanismo de garantia de qualidade para Engenharia Web é uma atividade de comunicação (Unidade 2) atenciosa e completa.

Como vimos anteriormente, a atividade de modelagem (análise e projeto) é iniciada assim que os requisitos de conteúdo, funcionalidades, restrições e desempenho forem coletados. É fundamental que a qualidade dos modelos de análise e projeto seja avaliada. Por um lado, para modelos simples, a equipe de Engenharia Web pode criar uma lista de verificação genérica usada para avaliar o modelo. Por outro lado, para modelos mais complexos ou críticos, uma revisão em pares ou em equipe (discutidos a seguir) pode ser realizada para cada modelo que for criado.

Assim que o código do incremento da aplicação Web estiver finalizado, a qualidade desse código pode ser avaliada por meio de uma sequência sistemática de testes (Seção 5.4). É fundamental mencionar que os testes precisam exercitar todos os requisitos do usuário e, ao mesmo tempo, examinar os aspectos técnicos do incremento (por exemplo, integridade arquitetural, mecanismos de navegação etc.).

Revisão em pares. Abordagem, adaptada do conceito de programação em pares (Seção 7.5.2.1) do método de Programação Extrema (BECK & ANDRES, 2004), eficaz e ágil de garantir a qualidade dos produtos de trabalho que são produzidos como consequência das atividades de Engenharia Web. Essa abordagem sugere que todos os produtos de trabalho (por exemplo, modelos de

análise e projeto, código HTML ou XML, diversos *scripts*, e conteúdo e funções) sejam revisados por um par de engenheiros Web. O responsável pelo produto de trabalho o apresenta a outro membro da equipe, que procura os erros, inconsistências e omissões. Trabalhando juntos, o par tenta melhorar a qualidade do produto de trabalho.

Revisão em equipe. Em algumas situações é necessário que toda a equipe de Engenharia Web revise um produto de trabalho. Isso ocorre quando, por exemplo, o produto de trabalho (por exemplo, o projeto arquitetural) tem grande impacto no desenvolvimento da aplicação Web.

Pressman & Lowe (2009) sugerem a seguinte mecânica para a revisão em equipe: dois membros da equipe usam a abordagem de revisão em pares para desenvolver algum produto de trabalho. Quando eles estiverem convencidos de que o produto de trabalho está concluído (pode ser na forma de rascunho), o par (chamado de produtores do produto de trabalho) pede a outros membros da equipe para participarem de uma revisão em equipe. Os produtores fornecem aos outros revisores qualquer informação que tenha sido produzida (seja em forma de cópia física ou em formato eletrônico). Os revisores prometem gastar pelo menos 30 a 45 minutos revisando o produto de trabalho e listando quaisquer questões, problemas ou impressões com base na revisão.

Dentro de no máximo 24 horas, a revisão em equipe é iniciada. Os responsáveis pelo produto de trabalho começam apresentando-o, explicando o que ele representa e como o leitor poderia interpretar o que é mostrado. Enquanto isso acontece, os revisores fazem perguntas (geralmente, baseando-se em notas preparadas antes da revisão) e apontam problemas em potencial. Os produtores observam cada um destes sem tentar solucioná-los imediatamente. Com o prosseguimento da revisão, os participantes seguem estas orientações (PRESSMAN & LOWE, 2009):

1. Revisar o produto, e não o produtor. Conduzida corretamente, a revisão em equipe deverá deixar todos os participantes com um sentimento caloroso de realização. Conduzida incorretamente, a revisão poderá assumir a aura de uma inquisição. Os erros deverão ser apontados gentilmente; o tom da revisão deverá ser solto e construtivo; a intenção não deve ser de embaraçar ou depreciar, mas sim de auxiliar.
2. Defina uma agenda e mantenha-a. Um dos principais problemas em todos os tipos de reunião é perder o rumo. Uma revisão deve ser mantida no rumo e no horário.
3. Limite o debate e a réplica. Quando uma questão for levantada por um revisor, pode não haver um acordo sobre o seu impacto. Ao invés de gastar tempo debatendo a questão, ela deverá ser registrada para ser resolvida posteriormente.
4. Anuncie problemas, mas não tente solucionar cada problema observado. Uma revisão não é uma sessão para resolver problemas.
5. Faça anotações. As notas podem ser feitas diretamente em um computador.
6. Gaste tempo suficiente para desvendar os problemas de qualidade, mas nem um minuto a mais. Em geral, uma revisão em equipe deverá ser concluída dentro de 60 a 90 minutos no máximo.

6.4 Considerações finais

Esta unidade finaliza a discussão sobre o Arcabouço de Processo de Engenharia Web ao apresentar os fundamentos das atividades guarda-chuva – atividades ocorrem em segundo plano enquanto as atividades principais do arcabouço estão acontecendo. Apesar de serem executadas em segundo plano, são atividades igualmente importantes e a equipe de Engenharia Web deve considerá-las explicitamente durante o ciclo de desenvolvimento da aplicação Web.

6.5 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

PRESSMAN, R. S.; LOWE, D. Gerenciamento de Mudança. In: _____. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

PRESSMAN, R. S. Formulação e Planejamento para Engenharia da Web. In: _____. *Engenharia de Software*. 6. ed. São Paulo: McGraw-Hill, 2006.

UNIDADE 7

Metodologias ágeis x Engenharia Web

7.1 Primeiras palavras

Esta unidade discute com mais detalhes o relacionamento entre Engenharia Web e Agilidade de Processo. Como discutido, Agilidade de Processo implica um enfoque de Engenharia de Software enxuto que incorpora ciclos de desenvolvimentos rápidos. E cada ciclo resulta na implantação de um incremento da aplicação Web.

Ou seja, o desenvolvimento da aplicação Web é realizado por meio da entrega de uma série de versões, chamadas de incrementos, que fornecem progressivamente mais funcionalidade para os clientes à medida que cada incremento é entregue.

Metodologias ágeis, discutidas nesta unidade, tais como Extreme Programming (XP) e Scrum, são bons exemplos de modelos de processo que aplicam o conceito de agilidade de processo no desenvolvimento de sistemas de software.

7.2 Problematizando o tema

Ao final desta unidade, espera-se que o leitor seja capaz de reconhecer e distinguir precisamente os conceitos básicos relacionados às metodologias ágeis e suas correlações com a Engenharia Web. Dessa forma, esta unidade pretende discutir as seguintes questões:

- Por que a agilidade no processo de Engenharia Web é tão importante?
- Quais são as principais características das metodologias ágeis?
- Quais são as principais semelhanças e diferenças das principais metodologias ágeis: XP, Scrum, Crystal e FDD?

7.3 Agilidade no processo de Engenharia Web

Conforme discutido na Unidade 1, um modelo de processo de Engenharia Web deve ser ágil. Isso implica um enfoque de engenharia enxuto, que incorpora ciclos de desenvolvimento rápidos. Cada ciclo resulta na implantação de um incremento da aplicação Web. Aoyama (1998) descreve a motivação para o enfoque ágil no desenvolvimento de aplicações Web da seguinte maneira:

> A internet mudou a prioridade principal do desenvolvimento de software de *que* para *quando*. O menor tempo para o mercado tornou-se a margem competitiva pela qual as empresas lutam. Assim, reduzir o ciclo de desenvolvimento agora é uma das missões mais importantes da engenharia de software (AOYAMA, 1998, p. 56).

Analogamente, Pressman & Lowe (2009) apresentam a seguinte motivação:

> Os engenheiros web devem entender que a empresa moderna exige adaptação, estratégias de negócios e regras mudam rapidamente, a gerência exige respostas quase instantâneas (até mesmo quando essas demandas são absurdas), e os interessados (*stakeholders*) continuam mudando de ideia até mesmo quando pedem entrega rápida. Os clientes se interessam por uma aplicação *Web* que é entregue quando eles precisam dela, e não pelo trabalho envolvido na criação de uma aplicação *Web* que pode ser entregue. Com tudo isso em mente, uma equipe de engenharia web precisa enfatizar a *agilidade* (PRESSMAN & LOWE, 2009, p. 12).

Dessa forma, é importante reconhecer que o desenvolvimento de aplicações Web (Unidade 1) deve ser definido dentro de um processo que:

1. Adota mudanças;
2. Encoraja a criatividade e a independência do pessoal de desenvolvimento e a interação forte com os interessados da aplicação Web;
3. Constrói sistemas usando pequenas equipes de desenvolvimento;
4. Enfatiza o desenvolvimento incremental usando ciclos de desenvolvimento curtos.

7.4 Metodologias ágeis

Nas décadas de 1980 e 1990, havia uma visão geral de que a melhor maneira de obter melhor software era por meio de um cuidadoso planejamento de projeto, garantia de qualidade formalizada, uso de métodos de análise e projeto apoiados por ferramentas CASE e controlados por um rigoroso processo de desenvolvimento de software. Essa abordagem envolve um *overhead* significativo de planejamento, de projeto e de documentação de sistema.

Segundo Sommerville (2007), esse *overhead* é justificado quando o trabalho de várias equipes de desenvolvimento necessita ser coordenado, quando o sistema é crítico e quando muitas pessoas diferentes serão envolvidas na manutenção do software durante seu tempo de vida. Contudo, quando essa abordagem de desenvolvimento pesada e baseada em planos foi aplicada a sistemas de pequenas e médias empresas, o *overhead* envolvido era tão grande que algumas vezes o tempo gasto para determinar como o sistema deveria ser desenvolvido era maior do que o empregado no desenvolvimento do programa e em testes.

Diante dessa situação, era preciso mudar. Não mudar as práticas, mas os valores. Em 2001, um grupo de renomados engenheiros de software (conhecidos como a *Aliança Ágil*) reuniu-se em Utah (EUA) e lançou o *Manifesto para o Desenvolvimento Ágil de Software*, divulgando os princípios que toda metodologia ágil deveria respeitar:

> Estamos descobrindo melhores maneiras de desenvolver software fazendo-o nós mesmos e ajudando outros a fazerem o mesmo. Por meio desse trabalho, passamos a valorizar:
> **Indivíduos e interações** mais que processos e ferramentas.
> **Software** funcionando mais que documentação detalhada.
> **Colaboração do cliente** mais que negociação de contrato.
> **Responder a mudanças** mais que seguir um plano.
> Ou seja, mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda.
>
> (BECK et al., 2012)

Embora as ideias subjacentes que guiam o desenvolvimento ágil já estivessem presentes há muitos anos, somente durante a década de 1990 é que essas ideias foram cristalizadas em um "movimento". Em essência, os métodos ágeis foram desenvolvidos em um esforço para vencer as fraquezas percebidas e reais da Engenharia de Software convencional.

Segundo Pressman (2006), o desenvolvimento ágil pode fornecer importantes benefícios, mas ele não é aplicável a todos os projetos, pessoas e situações. Ele também não é contrário à sólida prática de Engenharia de Software e pode ser aplicado com uma filosofia inerente a todo o trabalho de desenvolvimento de software.

Geralmente os métodos ágeis contam com uma abordagem iterativa e incremental para a especificação, o desenvolvimento e a entrega de software, e foram criados principalmente para apoiar o desenvolvimento de aplicações de negócios nas quais os requisitos de sistema mudam rapidamente durante o processo de desenvolvimento. Eles destinam-se a entregar um software de trabalho rapidamente aos clientes, que podem então propor novos requisitos e alterações a serem incluídos nas iterações posteriores do sistema (SOMMERVILLE, 2007).

Nas seções seguintes são apresentados alguns métodos ágeis de desenvolvimento de software. Há muitas semelhanças (em filosofia e prática) entre esses métodos. Pretende-se enfatizar as características de cada método que o tornam singular. É importante observar que todos os métodos ágeis apresentados satisfazem (em maior ou menor grau) aos 12 princípios que o *Manifesto*

para o Desenvolvimento Ágil de Software define para aqueles que querem alcançar agilidade:

1. Nossa maior prioridade é satisfazer ao cliente desde o início por meio de entrega contínua de software valioso.
2. Mudanças de requisitos são bem-vindas, mesmo que tardias no desenvolvimento. Os processos ágeis aproveitam as mudanças como vantagens para a competitividade do cliente.
3. Entrega frequente de software funcionando, a cada duas semanas até dois meses, de preferência no menor espaço de tempo.
4. O pessoal de negócio e os desenvolvedores devem trabalhar juntos diariamente durante todo o projeto.
5. Construa projetos em torno de indivíduos motivados. Forneça-lhes o ambiente e apoio necessário e confie que eles farão o trabalho.
6. O método mais eficiente e eficaz de levar informação para e dentro de uma equipe de desenvolvimento é a conversa face a face.
7. Software funcionando é a principal medida de progresso.
8. Processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente.
9. Atenção contínua à excelência técnica e ao bom projeto facilitam a agilidade.
10. Simplicidade – a arte de maximizar a quantidade de trabalho não realizado – é essencial.
11. As melhores arquiteturas, requisitos e projetos emergem de equipes auto-organizadas.
12. Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz, então sintoniza e ajusta adequadamente seu comportamento.

(BECK et al., 2012)

7.5 Extreme Programming (XP)

O *Extreme Programming* (XP) – Programação Extrema – é talvez o mais conhecido e mais amplamente utilizado dos métodos ágeis. O nome foi cunhado por Beck (1999) porque a abordagem enfatiza o desenvolvimento iterativo e o envolvimento do cliente em níveis “extremos”. Esta seção traz uma compilação dos principais conceitos desse método ágil, baseada principalmente nos seguintes trabalhos: Beck (1999) e Teles (2004).

No XP, todos os requisitos são expressos como cenários (chamados histórias de usuário), que são implementados diretamente como uma série de tarefas. Os programadores trabalham em pares e desenvolvem testes para cada tarefa antes da escrita do código. Todos os testes devem ser executados com sucesso quando um novo código for integrado ao sistema. Há um pequeno espaço de tempo entre as entregas dos incrementos do sistema (SOMMERVILLE, 2007).

7.5.1 Valores do XP

O XP se baseia em quatro valores fundamentais (TELES, 2004):

Feedback. Teles (2004) afirma que "o feedback é o mecanismo fundamental que permite que o cliente conduza o desenvolvimento diariamente e garanta que a equipe direcione as suas atenções para aquilo que irá gerar mais valor" (TELES, 2004, p. 22). Dessa forma, quando o cliente utiliza o sistema e reavalia as suas necessidades, ele realimenta a equipe com alterações nas necessidades que ainda serão implementadas e, eventualmente, naquelas que já fazem parte do software.

Comunicação. O *Extreme Programming* busca aproximar todos os envolvidos no projeto de modo que todos possam se comunicar face a face ou da forma mais rica possível, "a comunicação entre o cliente e a equipe permite que todos os detalhes do projeto sejam tratados com a atenção e a agilidade que merecem" (TELES, 2004, p. 22).

Simplicidade. Teles (2004) afirma que "o valor da simplicidade nos ensina a implementar aquilo que é suficiente para atender a cada necessidade do cliente" (TELES, 2004, p. 22). Ou seja, ao codificar uma funcionalidade, deve-se preocupar apenas com os problemas atuais e deixar os problemas do futuro para o futuro. Dessa forma, deve-se não tentar especular sobre funcionalidades que poderão ser necessárias futuramente. Ao evitar especulações, ganha-se tempo e permite que o cliente tenha acesso à funcionalidade o mais rápido possível.

Coragem. Desde que o sistema é desenvolvido de forma incremental, a equipe continuamente realiza manutenção do software e cria novas funcionalidades. Não é raro que a equipe tenha que alterar alguma funcionalidade que está funcionando, o que leva ao risco de introduzir falhas no sistema. Por essa razão, Teles (2004) argumenta que "a equipe necessita ser corajosa e acreditar que, utilizando as práticas e valores do XP, será capaz de fazer o software evoluir com segurança e agilidade" (TELES, 2004, p. 23).

7.5.2 Práticas do XP

O XP se baseia nas seguintes práticas (TELES, 2004):

Cliente presente. O XP trabalha com a premissa de que o *feedback* do cliente é fundamental na condução do desenvolvimento do sistema. Dessa forma, para que o *feedback* possa ocorrer, é essencial que o cliente participe ativamente do processo de desenvolvimento. Em adição, a presença do cliente simplifica o processo em diferentes aspectos, principalmente na comunicação. Dessa forma, a equipe de desenvolvimento é formada pelo cliente e pelos desenvolvedores do sistema.

Jogo do planejamento. O desenvolvimento é feito em iterações curtas de poucas semanas (em torno de duas, em média). No início da iteração, desenvolvedores e cliente reúnem-se para priorizar as funcionalidades. Essa reunião recebe o nome de jogo do planejamento. Nela, o cliente identifica prioridades, e os desenvolvedores as estimam. O cliente é essencial nesse processo e dessa maneira ele fica sabendo o que está acontecendo e o que vai acontecer no projeto. Ao final de cada iteração, o cliente recebe novas funcionalidades, completamente testadas e prontas para serem postas em produção.

Reuniões em pé.[17] A equipe de desenvolvimento se reúne a cada manhã para avaliar o trabalho que foi realizado no dia anterior e priorizar aquilo que será implementado no dia que se inicia. Consiste em reuniões em pé para não se perder o foco nos assuntos, apenas abordando tarefas já realizadas e tarefas a serem realizadas pela equipe.

Desenvolvimento orientado a testes. O XP é destinado à construção de sistemas com alta qualidade, o que leva à necessidade de diversos mecanismos de validação para assegurar que o software está correto. Os desenvolvedores escrevem testes para cada funcionalidade antes de codificá-las. Ou seja, primeiro criam-se os testes unitários e depois é realizada a implementação do código para que os testes funcionem. Essa abordagem é complexa no início, pois vai contra o processo de desenvolvimento de muitos anos. No entanto, os testes unitários são essenciais para que a qualidade do projeto seja mantida. Além disso, *testes de aceitação* são testes construídos pelo cliente e pelo conjunto de analistas e testadores para aceitar um determinado requisito do sistema.

Refatoração. É o ato de alterar um código sem afetar a funcionalidade que ele implementa (TELES, 2004). É um processo que permite a melhoria contínua da programação, com o mínimo de introdução de erros e mantendo a compatibilidade com o código já existente. Refatorar melhora a clareza (leitura) do

17 Em inglês: Stand-up Meeting.

código, divide-o em módulos mais coesos e de maior reaproveitamento, evitando a duplicação de código-fonte.

Código coletivo. O código-fonte não tem dono e ninguém precisa solicitar permissão para poder modificá-lo. O objetivo com isso é fazer a equipe conhecer todas as partes do sistema. Isso fornece maior agilidade ao processo e cria mais um mecanismo de revisão e verificação do código, já que aquilo que é escrito por um par hoje[18] acaba sendo manipulado por outro amanhã. Se alguma coisa estiver confusa no código, o par deverá fazer refatoração para torná-lo mais legível.

Padrões de codificação. A equipe de desenvolvimento precisa estabelecer regras para programar, e todos devem seguir essas regras. Dessa forma parecerá que todo o código-fonte foi editado pela mesma pessoa, mesmo quando a equipe possui 10 ou 100 membros. Essa prática serve para tornar o sistema mais homogêneo e permitir que qualquer manutenção futura seja efetuada com maior velocidade.

Projeto simples. Simplicidade é um princípio do XP. Ao invés de criar generalizações, de modo a prepará-lo para possíveis necessidades, a equipe deve optar por um código mais simples possível que atenda as necessidades da funcionalidade que está implementando. A equipe se baseia na premissa de que os membros são capazes de incorporar qualquer necessidade futura quando e se surgir. Para tal, eles contam com a refatoração, os testes e as demais práticas do XP para apoiá-los.

Metáfora. Procura facilitar a comunicação com o cliente, entendendo a realidade dele. Metáforas têm o poder de transmitir ideias complexas de forma simples, por meio de uma linguagem comum que é estabelecida entre a equipe de desenvolvimento e o cliente. Dessa forma, metáforas servem para traduzir as palavras do cliente para o significado que ele espera dentro do projeto.

Ritmo sustentável. Para garantir que a equipe tenha sempre o máximo de rendimento e produza software com melhor qualidade possível, o XP recomenda trabalhar com qualidade, buscando ter ritmo de trabalho saudável (40 horas/semana e 8 horas/dia), sem horas extras. Horas extras são permitidas quando trouxerem produtividade para a execução do projeto. Outra prática que se verifica nesse processo é a utilização de trabalho energizado, em que se busca trabalho motivado sempre. Para isso, o ambiente de trabalho e a motivação da equipe devem estar sempre em harmonia.

Integração contínua. Quando uma nova funcionalidade é incorporada ao sistema, ela pode afetar outras que já estavam implementadas. Dessa forma, sempre que produzir uma nova funcionalidade, nunca esperar uma semana para

18 Ver discussão sobre programação em par na Seção 7.5.2.1

integrar à versão atual do sistema. Isso só aumenta a possibilidade de conflitos e a possibilidade de erros no código-fonte. Integrar de forma contínua permite saber o *status* real do desenvolvimento, pois essa prática leva os desenvolvedores a integrarem seus códigos ao restante do sistema diversas vezes ao dia.

Pequenas versões (*releases*). A liberação de pequenas versões funcionais do projeto auxilia muito no processo de aceitação por parte do cliente, que já pode testar uma parte do sistema que está comprando.

Programação em par. Programação em par é uma das práticas mais conhecidas e mais polêmicas do XP. Dessa forma, é justificável que seja apresentada em um nível maior de detalhes.

7.5.2.1 Programação em par

Programação em par sugere que todo e qualquer código produzido no projeto seja sempre implementado por dois desenvolvedores ao mesmo tempo e em um mesmo computador. Enquanto uma pessoa (o condutor) assume o teclado e digita os comandos que farão parte do programa, a outra (o navegador) a acompanha fazendo um trabalho de estrategista.

À primeira vista, a programação em par parece ser uma prática fadada ao fracasso e ao desperdício. Afinal, embora possa haver benefícios, tem-se a impressão de que ela irá consumir mais recursos ou irá elevar o tempo do desenvolvimento. Entretanto, não é exatamente isso o que ocorre. Nesta seção são apresentadas as principais características dessa prática do XP.

Programação em par – **uma analogia**. Para explicar o conceito de programação em par, Vinícius Teles apresenta a analogia descrita a seguir:

> Você embarcaria em um avião que não tivesse um copiloto? Provavelmente não, porque mesmo que você não ligasse para isso, a companhia aérea se preocupa (e muito) com essa questão, de modo que ela nem cogita permitir uma coisa dessas. Mas, para que serve um copiloto se, na prática, seria perfeitamente possível pilotar um avião com uma única pessoa? Aliás, cada vez mais, é o próprio computador de bordo quem faz a maior parte do trabalho. Então, porque as companhias aéreas fazem questão de ter piloto e copiloto em 100% dos voos?
>
> O que aconteceria se o piloto tivesse um problema de saúde exatamente na hora do pouso, por exemplo? Havendo um copiloto, isso não seria um problema. Acredita-se (e as estatísticas confirmam) que a chance de duas pessoas errarem seja significativamente menor que a chance de uma única pessoa se equivocar. No caso da aviação, falhas humanas podem ter consequências drásticas. Os custos materiais podem ser elevados e o custo humano pode ser impagável. Por essa razão, entre outras, companhias aéreas colocam

> dois profissionais (exaustivamente treinados e relativamente caros) para fazer o trabalho que um só poderia fazer tranquilamente.
>
> Isso não é suficiente para garantir a segurança de um voo, outros mecanismos também são usados e representam papéis essenciais. Porém a utilização de um copiloto é uma das práticas que colaboram para elevar a segurança dos voos. Embora pareça inconcebível colocar dois profissionais para fazer o trabalho que apenas um poderia fazer sozinho, como a programação em par sugere, as práticas da aviação são um exemplo de que a utilização de pares também é adotada em outras áreas, nas quais são, inclusive, bastante valorizadas (TELES, 2012).

Simplicidade e revisão contínua. A programação em par ajuda os desenvolvedores a criarem soluções mais simples, mais rápidas de implementar e mais fáceis de manter. Isso ocorre em grande parte devido à oportunidade de dialogar e trocar ideias sobre programas que estejam sendo desenvolvidos.

Teles (2004) argumenta que a programação em par congrega diversas técnicas em uma só. Ela é utilizada, por exemplo, para fazer a revisão do código em par. Isto é, enquanto o condutor digita, o navegador está permanentemente revisando o código e tentando evitar que eventuais erros do condutor passem despercebidos. Ou seja, do ponto de vista de revisão, a programação em par tem o objetivo de fazer com que as correções sejam aplicadas imediatamente, assim que os erros apareçam no código.

Disseminação do conhecimento e confiança. Uma das características mais marcantes da programação em par é a sua capacidade de disseminação de conhecimento. Os desenvolvedores sempre trocam de pares, fazendo com que ocorra maior compartilhamento de informações ao longo do projeto. Além disso, os desenvolvedores também se revezam no desenvolvimento das funcionalidades. Ou seja, é possível que hoje um desenvolvedor trabalhe em uma parte do sistema e amanhã possa vir a codificar outra parte completamente diferente, com outro par.

A programação em par também é uma forma de fazer com que o desenvolvedor tenha mais confiança no código que produz. Afinal, o código foi produzido por ele e mais outra pessoa que o ajudou a revisá-lo. Quando é conhecido *a priori* que mais de uma pessoa, ou talvez várias, já olhou para o código no qual um desenvolvedor está trabalhando e estão de acordo sobre este, o desenvolvedor tem mais confiança de que esse código realmente irá funcionar.

Produtividade. O conjunto de características apresentadas anteriormente faz com que a programação em par acelere o desenvolvimento significativamente, embora à primeira vista pareça o contrário. Estudos científicos (WILLIAMS & KESSLER, 2000) demonstraram que praticamente não existem diferenças de

produtividade em equipes que utilizam programação em par e outras que não utilizam. Dada certa quantidade de desenvolvedores e o mesmo tempo, ambas as equipes são capazes de produzir basicamente a mesma coisa.

Esse efeito colateral que afeta a produtividade da equipe se dá principalmente pelo termo conhecido como *pressão do par*. O ato de programar demanda grande concentração e produz energia considerável. Entretanto, no dia a dia de um programador, ele se depara com diversas fontes de distração – e-mail, mensagens instantâneas, bate-papo com colegas, cansaço – que contribuem para que o programador diminua seu ritmo de trabalho no código.

Teles (2004) alega que quando o programador está acompanhado de outra pessoa, ele deixa de ter um compromisso apenas consigo mesmo. O seu compromisso se expande e passa a englobar também seu colega. Ou seja, sua responsabilidade aumenta já que passa a envolver mais alguém. O efeito disso é que diversos focos de distração são eliminados.

Problemas em sua adoção. Programar em par exige que as pessoas envolvidas sejam receptivas, compreensivas umas com as outras, engajadas e, sobretudo, humildes. É necessário aceitar que somos falíveis para que possamos programar em par. Teles (2004) argumenta que é essencial que o desenvolvedor não veja o código como uma extensão de si próprio. Ele não pode achar que um erro no código significa uma falha sua como pessoa. O código é apenas uma obra que tem vida própria e não representa o caráter e a personalidade de quem o criou. Pois, se o desenvolvedor imaginar que o código representa a si próprio, dificilmente irá buscar problemas e falhas no código. Afinal, ninguém gosta de saber que tem falhas.

7.5.3 Ciclo de desenvolvimento do XP

Para finalizar a discussão sobre o XP e consolidar os conceitos apresentados anteriormente, a Figura 26 apresenta o ciclo de desenvolvimento do XP. Como discutido, o XP se baseia em um conjunto de valores e práticas que ocorrem no contexto de quatro atividades do processo XP: *planejamento*, *projeto*, *codificação* e *teste*. Essas atividades são discutidas nos próximos parágrafos. Essa discussão é baseada na apresentada em Pressman (2006).

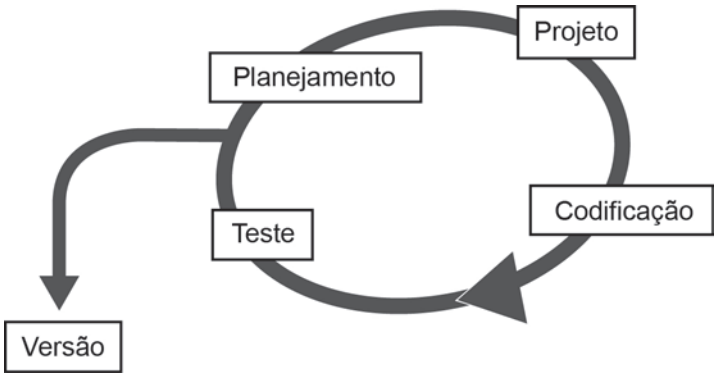


Figura 26 Ciclo de desenvolvimento – Processo XP.

Planejamento. Essa atividade inicia-se com a escrita pelo cliente de um conjunto de histórias de usuário que descrevem as características e funcionalidades que representam os requisitos para o sistema a ser implementado.

A cada história, o cliente define uma prioridade que é atribuída baseando-se na importância da característica ou da funcionalidade no negócio apoiado pelo sistema a ser desenvolvido. Membros da equipe XP avaliam cada história e lhe atribuem um custo – medido em semanas de desenvolvimento. Pressman (2006) argumenta que se uma história precisar de mais que três semanas de desenvolvimento deve-se pedir ao cliente que ele divida essa história em histórias menores, e as atribuições de prioridade e de custo são realizadas novamente.

A seguir, os clientes e a equipe XP trabalham em conjunto para decidir o conjunto de histórias a ser implementado na próxima versão (incremento). Ou seja, é firmado um acordo quanto às histórias a serem incluídas, à data de entrega e a outros assuntos relacionados ao desenvolvimento de software.

Ao longo do ciclo de desenvolvimento, o cliente pode adicionar novas histórias, mudar a prioridade de histórias existentes, subdividir histórias e eliminá-las. A equipe XP então deve reconsiderar todos os incrementos remanescentes e modificar seus planos adequadamente (PRESSMAN, 2006).

Projeto. O projeto XP segue rigorosamente o valor “mantenha a simplicidade” (vide discussão). Em relação à discussão sobre a importância de projetos simples, Pressman (2006) argumenta que um projeto XP fornece diretrizes para a implementação de uma história de usuário como ela está descrita – nada mais, nada menos. Ou seja, o projeto de funcionalidade extra (em razão de o desenvolvedor considerar que será necessária futuramente) é desencorajado.

XP encoraja o uso de refatoração (vide discussão anterior) e de cartões CRC[19] (WIRFS-BROCK, WILKERSON & WEINER, 1990) que identificam e organizam as

19 CRC é acrônimo de Class-Responsibility-Colaboration.

classes orientadas a objetos que são relevantes ao incremento de software atual. O uso de cartões CRC é recomendado de forma que permita o projeto em equipe. Cartões CRC permitem a descrição dos conceitos identificados na metáfora na forma de classes. Responsabilidades são identificadas para cada classe. As colaborações determinam as interações entre classes. Os cartões permitem que todo o time possa colaborar com o projeto. Na realidade, os cartões CRC são os únicos artefatos de projeto que são produzidos como parte do processo XP. O Quadro 1 ilustra um possível cartão CRC para a classe Pedido da aplicação Web **LivrariaVirtual.com.br**.

Quadro 1 Cartão CRC para LivrariaVirtual.com.br.

Classe: Pedido	
Descrição da classe Pedido	
Responsabilidade	Colaboração
Verifica se o item está em estoque	Item
Determina preço	Item
Verifica pagamento válido	Cliente
Envia para endereço destino	

Se durante o projeto de uma história de usuário um problema difícil for detectado, o XP recomenda a implementação e avaliação de um protótipo operacional – denominado *solução de ponta* – daquela parte do projeto. Pressman (2006) argumenta que a intenção da criação da solução de ponta é diminuir o risco quando a implementação verdadeira começa e validar as estimativas originais correspondentes à história que contém o problema de projeto.

Cartões CRC consistem em uma abordagem simples para identificar e organizar as classes relevantes aos requisitos do sistema. A modelagem utilizando cartões CRC é descrita por Ambler (1995) da seguinte maneira:

- "Um modelo CRC é na realidade uma coleção de cartões de indexação que representam as classes. Os cartões são divididos em três seções. No alto do cartão, você escreve o nome da classe. No corpo do cartão você lista as responsabilidades da classe à esquerda e os colaboradores à direita" (AMBLER, 1995, p. 55).

O objetivo da utilização de cartões CRC é desenvolver uma representação organizada de classes. Responsabilidades são os atributos e operações relevantes à classe. Dito de forma simples, uma responsabilidade "é qualquer coisa que a classes sabe ou faz" (AMBLER, 1995, p. 58). Colaboradores são as classes necessárias para dar à classe a informação requerida para completar uma responsabilidade. Em geral, uma colaboração implica tanto solicitação de informação como solicitação de alguma ação.

Codificação. Como discutido, o XP recomenda o Desenvolvimento Orientado a Testes. Os desenvolvedores escrevem testes para cada funcionalidade antes de codificá-las. Ou seja, primeiro criam-se os testes unitários e depois é realizada a implementação do código para que estes funcionem.

Outro conceito-chave durante a atividade de codificação é a programação em par. Como discutido, essa prática recomenda que duas pessoas trabalhem juntas em uma estação de trabalho para criar o código correspondente a uma história de usuário.

E por fim, o XP prega a integração contínua que permite saber o *status* real do desenvolvimento, pois essa prática leva os desenvolvedores a integrarem seus códigos ao restante do sistema diversas vezes ao dia.

Teste. Como mencionado, os testes unitários são escritos pelos desenvolvedores antes da codificação das funcionalidades. É importante salientar que os testes unitários que são criados devem ser implementados usando uma ferramenta (por exemplo, JUnit) que lhes permita ser automatizados (consequentemente, eles podem ser executados fácil e repetidamente). Além disso, Pressman (2006) argumenta que a automatização de testes unitários encoraja uma estratégia de teste de regressão, sempre que o código é modificado – o que pode ser frequente em razão da prática de refatoração do XP. O teste de regressão consiste na reexecução de algum subconjunto de testes, que já foi conduzido anteriormente, com o objetivo de garantir que as modificações no código não propaguem efeitos colaterais indesejáveis (PRESSMAN, 2006).

Por fim, o XP prega a utilização de testes de aceitação – também denominados de testes do cliente –, que são especificados pelo cliente e focam as características e as funcionalidades globais do sistema, que são visíveis e revisadas pelo cliente. Testes de aceitação são derivados das histórias do usuário que foram implementadas como parte de um incremento de software.

7.6 Scrum

O Scrum é outra abordagem ágil cuja popularidade vem crescendo nos últimos anos. O termo Scrum é uma referência ao jogo de *Rugby*, no qual denota uma estratégia em que jogadores colocam uma bola quase perdida novamente em jogo por meio de trabalho em equipe. Em desenvolvimento de software, o trabalho do time Scrum consiste em desenvolver um *release* de software. Esta seção apresenta uma compilação dos principais conceitos do Scrum, baseada principalmente nos trabalhos de Schwaber & Sutherland (2010) e Schwaber (2004).

Segundo Schwaber & Sutherland (2010), Scrum não é um processo ou uma técnica para o desenvolvimento de produtos. Ao invés disso, é um framework dentro do qual você pode empregar diversos processos e técnicas. O papel do Scrum é fazer transparecer a eficácia relativa de suas práticas de desenvolvimento para que você possa melhorá-las, enquanto provê um framework dentro do qual produtos complexos possam ser desenvolvidos.

Scrum é fundamentado na teoria de controle de processos empíricos e emprega uma abordagem iterativa e incremental para otimizar a previsibilidade e controlar riscos. Ou seja, Scrum busca, de uma forma empírica, lidar com o caos, em detrimento a um processo bem definido. Três pilares sustentam cada implementação de controle de processos empíricos (SCHWABER & SUTHERLAND, 2010):

- O primeiro pilar é a *transparência*. A transparência garante que aspectos do processo que afetem o resultado sejam visíveis para aqueles que gerenciam os resultados. Esses aspectos não apenas devem ser transparentes, mas também o que está sendo visto deve ser conhecido.
- O segundo pilar é a *inspeção*. Os diversos aspectos do processo devem ser inspecionados com uma frequência suficiente para que variações inaceitáveis no processo possam ser detectadas.
- O terceiro pilar é a *adaptação*. Se o inspetor determinar, a partir da inspeção, que um ou mais aspectos do processo estão fora dos limites aceitáveis e que o produto resultante será inaceitável, ele deverá ajustar o processo ou o material que está sendo processado. Esse ajuste deve ser feito o mais rápido possível para minimizar desvios posteriores.

Existem três pontos para inspeção e adaptação em Scrum. A *Daily Scrum* é uma reunião diária utilizada para inspecionar o progresso em direção à meta da *Sprint* e para realizar adaptações que otimizem o valor do próximo dia de trabalho. Além disso, as reuniões de *Revisão da Sprint* e de *Planejamento da Sprint* são utilizadas para inspecionar o progresso em direção à meta da entrega (*release*) e para fazer as adaptações que otimizem o valor da próxima *Sprint*. Finalmente, a *Retrospectiva da Sprint* é utilizada para revisar a *Sprint* passada e definir que adaptações tornarão a próxima *Sprint* mais produtiva, recompensadora e gratificante. Esses conceitos serão discutidos nas próximas seções.

7.6.1 Framework Scrum

O framework Scrum (SCHWABER, 2004) consiste em um conjunto formado por times de Scrum e seus papéis associados, *Time-Boxes* (eventos com duração fixa), *Artefatos* e *Regras*.

7.6.1.1 Papéis no Scrum

Times de Scrum são projetados para otimizar flexibilidade e produtividade. Para esse fim, eles são auto-organizáveis, multidisciplinares e trabalham em iterações. Cada time de Scrum possui três papéis: o *ScrumMaster*, o *Product Owner* e o *Time*.

O *ScrumMaster* é responsável por garantir que o time de Scrum esteja aderindo aos valores do Scrum, às práticas e às regras. O *ScrumMaster* ajuda o time de Scrum e a organização a adotarem o Scrum. O *ScrumMaster* educa o time de Scrum treinando-o e levando-o a ser mais produtivo e a desenvolver produtos de maior qualidade.

O *Product Owner* é a única pessoa responsável pelo gerenciamento do *Product Backlog* (discutido na próxima seção) e por garantir o valor do trabalho realizado pelo *Time*. Essa pessoa mantém o *Product Backlog* e garante que ele esteja visível para todos. Todos sabem quais itens têm a maior prioridade, de forma que todos sabem em que se irá trabalhar.

O *Time* executa o trabalho propriamente dito. O *Time* consiste em desenvolvedores com todas as habilidades necessárias para transformar o *Product Backlog* em incrementos de funcionalidades potencialmente entregáveis em cada *Sprint*. *Times* também são multidisciplinares: membros do time devem possuir todo o conhecimento necessário para criar um incremento no trabalho.

Times também são auto-organizáveis. Ninguém – nem mesmo o *ScrumMaster* – diz ao time como transformar o *Product Backlog* em incrementos de

funcionalidades entregáveis. O time descobre por si só. Cada membro do time aplica sua especialidade a cada um dos problemas. A sinergia que resulta disso melhora a eficiência e a eficácia geral do time como um todo.

Segundo Schwaber & Sutherland (2010), o tamanho ótimo para um time é de sete pessoas, podendo somar ou diminuir duas pessoas desse time. Quando há menos do que cinco membros em um time, há menor interação e, como resultado, há menor ganho de produtividade. Menos do que isso, o time poderá encontrar limitações de conhecimento durante partes da *Sprint* e não será capaz de entregar uma parte pronta do produto. Se há mais do que nove membros, há simplesmente a necessidade de muita coordenação. Times grandes geram muita complexidade para que um processo empírico consiga gerenciar.

7.6.1.2 Time-Boxes

Scrum emprega eventos com duração fixa – os *Time-Boxes* – para criar regularidade. Entre os elementos do Scrum que têm duração fixa, pode-se citar a *Reunião de Planejamento da Release (Entrega)*, a *Reunião de Planejamento da Sprint*, a *Sprint*, a *Daily Scrum*, a *Revisão da Sprint* e a *Retrospectiva da Sprint*.

O propósito do *Planejamento da Release (Entrega)* é o de estabelecer um plano e metas que o time de Scrum e o resto da organização possam entender e comunicar. O plano da *release* estabelece a sua meta da *release*, as maiores prioridades do *Product Backlog*, os principais riscos e as características gerais e funcionalidades que estarão contidas na *release*. Ele estabelece também uma data de entrega e custo prováveis que devem se manter se nada mudar. A organização pode então inspecionar o progresso e fazer mudanças nesse plano da entrega a cada *Sprint*.

A *Sprint* é uma iteração de um mês ou menos, de duração fixa e consistente com o esforço de desenvolvimento. Todas as *Sprints* têm como resultado um incremento do produto final que é potencialmente entregável. Durante a *Sprint*, o *ScrumMaster* garante que não será feita nenhuma mudança que possa afetar a sua meta da *Sprint*. Tanto a composição do time quanto as metas de qualidade devem permanecer constantes durante a *Sprint*. As *Sprints* consistem na *Reunião de Planejamento de Sprint*, no *Trabalho de Desenvolvimento*, na *Revisão da Sprint* e na *Retrospectiva da Sprint*. As *Sprints* ocorrem uma após a outra, sem intervalos entre elas.

A *Reunião de Planejamento da Sprint* é o momento no qual a iteração é planejada. Ela consiste de duas partes: a parte do “o quê?” e a parte do “como?”.

A primeira parte é o momento no qual é decidido o que será realizado na *Sprint*. O *Product Owner* apresenta ao time o que é mais prioritário no *Product*

Backlog. Eles trabalham em conjunto para definir quais funcionalidades deverão ser desenvolvidas durante a próxima *Sprint*. As entradas para essa reunião são o *Product Backlog*, o incremento mais recente ao produto, a capacidade do time e o histórico de desempenho do time. Cabe somente ao time a decisão de quanto do *Backlog* ele deve selecionar. As funcionalidades selecionadas, com os demais artefatos produzidos na reunião de trabalho, formam o *Selected Product Backlog*. Somente o time pode avaliar o que ele é capaz de realizar na próxima *Sprint*. Uma vez selecionado o *Product Backlog*, a meta da *Sprint* é delineada. Essa meta é o objetivo que será atingido por meio da implementação do *Selected Product Backlog*. Ela é uma descrição que fornece orientação ao time sobre a razão pela qual ele está desenvolvendo o incremento. A meta da *Sprint* é um subconjunto da meta da *Release*.

A segunda parte é o momento no qual o time entende como desenvolverá essa funcionalidade em um incremento do produto durante a *Sprint*. O time geralmente começa projetando o trabalho. Enquanto projeta, o time identifica tarefas, que são pedaços detalhados do trabalho necessário para converter o *Product Backlog* em software funcional. As tarefas devem ser decompostas para que possam ser feitas em menos de um dia. Essa lista de tarefas é chamada de *Sprint Backlog*. O time se auto-organiza para se responsabilizar pelo trabalho contido no *Sprint Backlog*, tanto durante a *Reunião de Planejamento da Sprint* quanto no próprio momento da execução da *Sprint*. O *Product Owner* está presente durante a segunda parte da *Reunião de Planejamento da Sprint* para esclarecer o *Product Backlog* e para ajudar a efetuar trocas. Se o time concluir que ele tem trabalho demais ou de menos, ele pode renegociar o *Product Backlog* escolhido com o *Product Owner*.

Ao final da *Sprint*, é feita a reunião de *Revisão da Sprint*. Durante essa reunião, o time de Scrum e as partes interessadas concordam sobre o que acabou de ser feito. Baseados nisso e em mudanças no *Product Backlog* feitas durante a *Sprint*, eles concordam sobre quais são as próximas coisas que podem ser feitas. Essa é uma reunião informal, com a apresentação da funcionalidade, que tem a intenção de promover a colaboração sobre o que fazer em seguida.

Após a *Revisão da Sprint* e antes da próxima reunião de *Planejamento da Sprint*, o time de Scrum tem a *Reunião de Retrospectiva da Sprint*. Nessa reunião, o *ScrumMaster* encoraja o time a revisar, dentro do modelo de trabalho e das práticas do processo do Scrum, seu processo de desenvolvimento, de forma a torná-lo mais eficaz e gratificante para a próxima *Sprint*. A finalidade da retrospectiva é inspecionar como ocorreu a última *Sprint* em se tratando de pessoas, das relações entre elas, dos processos e das ferramentas.

E por fim, cada time se encontra diariamente para uma *reunião em pé* de 15 minutos chamada *Daily Scrum*. Essa reunião é sempre feita no mesmo

horário e no mesmo local durante as *Sprints*. Durante a reunião, cada membro explica:

- o que ele realizou desde a última reunião;
- o que ele vai fazer antes da próxima reunião;
- quais obstáculos estão em seu caminho.

As *Daily Scrums* melhoram a comunicação, eliminam outras reuniões, identificam e removem empecilhos para o desenvolvimento, ressaltam e promovem a tomada rápida de decisões e melhoram o nível de conhecimento de todos acerca do projeto.

O *ScrumMaster* garante que o time realize essa reunião. O Time é responsável por conduzir a *Daily Scrum*. O *ScrumMaster* ensina o time a manter a *Daily Scrum* com curta duração, reforçando as regras e garantindo que as pessoas falem brevemente.

A *Daily Scrum* é uma inspeção do progresso na direção da meta da *Sprint* (as três perguntas). Geralmente acontecem reuniões subsequentes para realizar adaptações ao trabalho que está por vir na *Sprint*. A intenção é otimizar a probabilidade de que o time alcance sua meta. Essa é uma reunião fundamental de inspeção e adaptação no processo empírico do Scrum.

7.6.1.3 Artefatos e regras

Scrum utiliza quatro artefatos principais. O *Product Backlog* contém uma lista de requisitos de todos os entregáveis para que aquele produto faça sentido. Essa lista deve estar sempre priorizada por valor de negócio. Requisitos podem ser adicionados ou removidos a qualquer momento, assim como a prioridade também pode mudar. Ou seja, ele precisa ser continuamente mantido pelo *Product Owner*, visando maximizar o retorno sobre o investimento.

O *Sprint Backlog* é uma lista de tarefas para transformar o *Product Backlog*, por uma *Sprint*, em um incremento do produto potencialmente entregável. Um *burndown* é uma medida do *backlog* restante pelo tempo e geralmente é atualizado diariamente. Um *Sprint Burndown* mede os itens do *Sprint Backlog* restantes ao longo do tempo de uma *Sprint* e geralmente é atualizado diariamente após a *Daily Scrum*. Um *Sprint Burndown* é um gráfico com as informações referentes à execução da *Sprint*. O eixo X mostra os dias de trabalho da *Sprint* e o eixo Y mostra a quantidade de esforço necessária para terminar a *Sprint*. Esse gráfico mostra a execução indo de cima para baixo com relação ao esforço necessário de execução e da esquerda para direita com relação aos dias de

trabalho. A Figura 27 ilustra um exemplo de um *Sprint Burndown* (ciclo de treze dias). Um *Release Burndown* mede o *Product Backlog* restante ao longo do tempo de um plano de *release*.

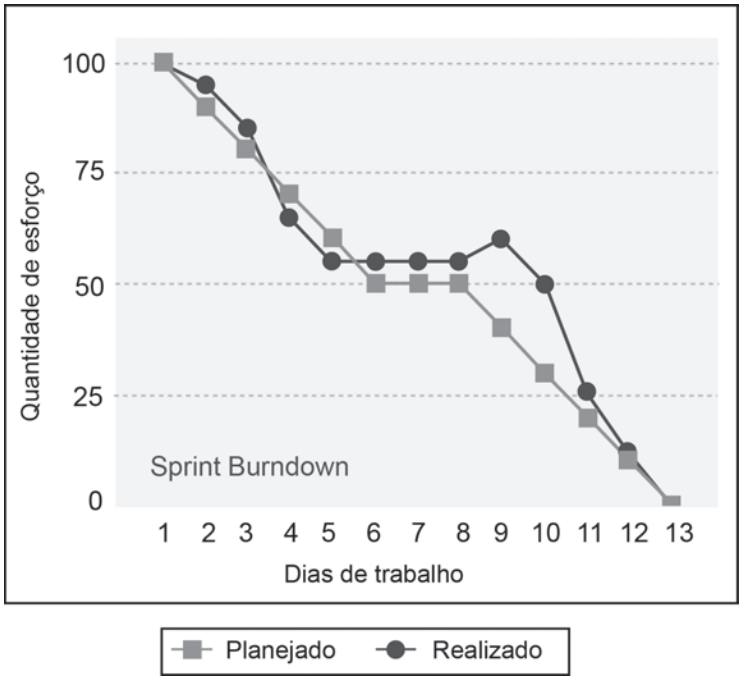


Figura 27 *Sprint Burndown*.

As regras fazem o elo entre os *Time-Boxes*, os papéis e os artefatos do Scrum. Como exemplos, podemos citar três regras do Scrum (SCHWABER & SUTHERLAND, 2010):

- O *Product Backlog* representa tudo que é necessário para desenvolver e lançar um produto de sucesso.
- O objetivo de cada *Sprint* é ter como resultado incrementos de funcionalidade (presentes no *Product Backlog*) potencialmente entregáveis.
- Somente membros do time – as pessoas comprometidas em transformar o *Product Backlog* em um incremento – podem falar durante uma *Daily Scrum*.

7.6.1.4 Ciclo de desenvolvimento do Scrum

Para finalizar a discussão sobre o Scrum e consolidar os conceitos apresentados anteriormente, a Figura 28 apresenta o ciclo de desenvolvimento do Scrum:

1. O ciclo começa com o *Product Owner* definindo uma visão de produto compartilhada com o time.

2. Essa visão é então transformada, junto com o time, no que é chamado de *Product Backlog*.
3. Com o *Backlog* em mãos, o time e o *Product Owner* fazem a primeira parte do *Sprint Planning*,[20] em que será definido "o que" será feito durante a *Sprint*.
4. O *Selected Product Backlog* é o resultado dessa reunião, e define a quantidade de trabalho com a qual o time se comprometeu a entregar naquela *Sprint*. Ele se mantém inalterado durante toda a *Sprint*.
5. A segunda parte do *Sprint Planning* é a hora de definir "como" a solução será implementada. É a fase em que o time deve identificar as melhores soluções para resolver cada um dos problemas, identificando as tarefas necessárias para atingir o objetivo.
6. O resultado dessa reunião é o *Sprint Backlog*, uma lista de atividades necessárias para entregar a versão final das funcionalidades que serão aceitas pelo cliente.
7. Começa a *Sprint*. É a fase de "construção", em que diariamente (*Daily Scrum*) o time se reúne para sincronizar o trabalho que está sendo feito. O ciclo menor representa a primeira iteração, em geral de duas a quatro semanas, em que todas as fases do projeto são exercitadas para que seja possível concluir um incremento de funcionalidade pronto para ser utilizado. Dentro deste, o ciclo maior representa as iterações diárias no qual o time se replaneja constantemente para achar soluções aos problemas que surgem durante o desenvolvimento.
8. Ao final da *Sprint*, uma reunião de *Revisão da Sprint* acontece, na qual são apresentadas ao cliente as funcionalidades que estão disponíveis para o uso. O cliente pode aceitar ou rejeitar as funcionalidades nesse momento, além de sugerir melhorias ou novas ideias.
9. Finalmente, para encerrar o ciclo de melhoria contínua, acontece a *Retrospectiva da Sprint*, na qual será identificado "o que deu certo" e "o que deu errado" na *Sprint*. São identificados pontos de melhoria no processo e levantados impedimentos que atrapalham o melhor desempenho do time. Esse é um momento de reflexão do time, que precisa se sentir seguro para falar tudo o que tem que ser dito.

20 Reunião de Planejamento da Sprint.

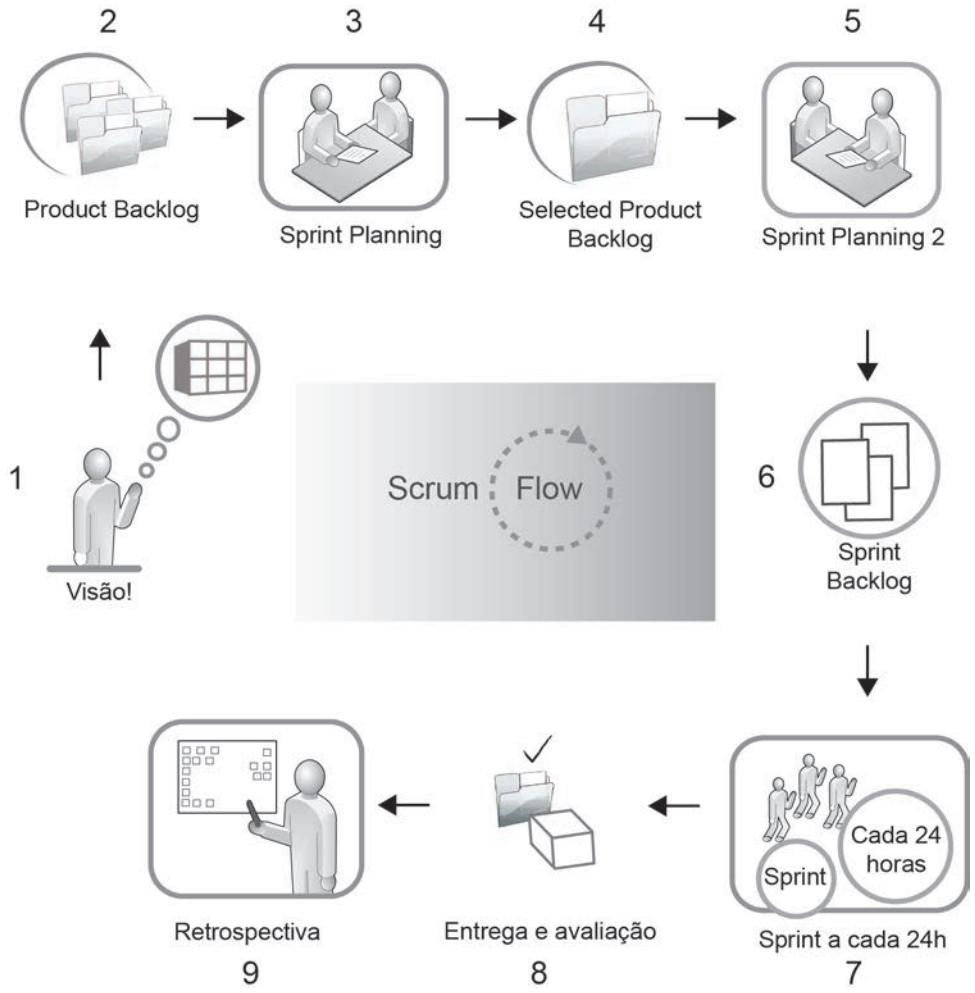


Figura 28 Ciclo de desenvolvimento do Scrum.

7.7 Outras metodologias ágeis

As metodologias ágeis mais conhecidas e empregadas são XP e Scrum, apresentadas nas seções anteriores. Contudo, outros métodos ágeis incluem Crystal (COCKBURN, 2001) e FDD (PALMER & FELSING, 2002). Nesta seção, é apresentado um breve panorama desses dois métodos ágeis de desenvolvimento de software.

7.7.1 Metodologias Crystal

As metodologias Crystal[21] são uma família de métodos ágeis que em diferentes tipos de projetos exigem diferentes tipos de metodologias, e são definidas por meio do número de pessoas no projeto e das consequências dos erros.

21 Disponível em: <http://alistair.cockburn.us/crystal/crystal.html>. Acesso em: 24 jan. 2012.

Essa família de métodos ágeis foi definida com o objetivo de conseguir uma abordagem de desenvolvimento de software que valoriza a “manobrabilidade” durante o que Cockburn (2002 apud PRESSMAN, 2006, p. 71) caracteriza como “um jogo cooperativo de invenção e comunicação de recursos limitados, com o principal objetivo de entregar softwares úteis funcionando e com o objetivo secundário de preparar-se para o jogo seguinte”. Para conseguir a manobrabilidade foi definido um conjunto de metodologias, cada qual com elementos centrais que são comuns a todas, e papéis, padrões de processos, produtos de trabalho e práticas específicas de cada uma.

A família Crystal é, na verdade, um conjunto de processos ágeis que se mostram efetivos para diferentes tipos de projeto. A intenção é permitir que equipes ágeis selecionem o membro da família Crystal mais apropriado para o seu projeto e ambiente (PRESSMAN, 2006).

7.7.2 FDD (Feature Driven Development)

O FDD[22] (Feature Driven Development – Desenvolvimento Guiado por Características) foi originalmente concebido por Coad, Lefebvre & De Luca (1999) como um modelo prático para a Engenharia de Software orientada a objetos. Palmer & Felsing (2002) estenderam esse trabalho ao definir um processo ágil e adaptativo que pode ser aplicado a projetos de software de tamanho moderado e grande.

No contexto do FDD, uma característica é uma função valorizada pelo cliente que pode ser implementada em duas semanas ou menos (COAD, LEFEBVRE & DE LUCA, 1999). Isto é, uma característica é um pequeno bloco de funcionalidades passível de entrega (incremento de entrega do FDD), podendo ser organizada em um agrupamento hierárquico relacionado ao negócio.

A abordagem FDD define duas fases (Figura 29): concepção & planejamento (pensar antes de fazer) e construção (fazer de forma repetitiva), com iterações de duas semanas ou menos. Além disso, o FDD utiliza cinco atividades colaborativas (que são denominadas “processos”): DMA (desenvolver um modelo abrangente); CLF (construir a lista de funcionalidades); PPC (planejar por característica); DPC (detalhar/projetar por característica); e CPC (construir por característica).

22 Disponível em: <http://www.featuredrivendevelopment.com>. Acesso em: 24 jan. 2012.

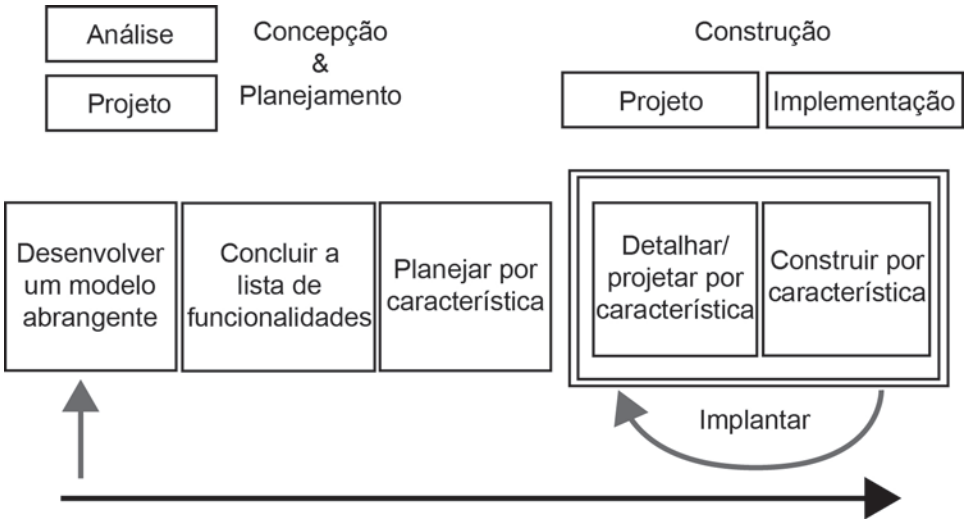


Figura 29 Processo de Desenvolvimento do FDD.

7.8 Considerações finais

Esta unidade discutiu o relacionamento entre Engenharia Web e metodologias ágeis que prega um processo que: (1) adota mudanças, (2) encoraja a criatividade e a independência do pessoal de desenvolvimento e interação forte com os interessados da aplicação Web, (3) constrói sistemas usando pequenas equipes de desenvolvimento e (4) enfatiza o desenvolvimento incremental usando ciclos de desenvolvimento curtos. O que se pode abstrair dessa discussão é que metodologias são fortemente indicadas para o desenvolvimento Web.

Além disso, esta unidade apresentou alguns métodos ágeis de desenvolvimento de software, tentando enfatizar as características de cada método que o tornam singular.

7.9 Estudos complementares

Para estudos complementares sobre os tópicos abordados nesta unidade, o leitor interessado pode consultar as seguintes referências:

BECK, K. *Extreme Programming Explained*: embrace change. Boston: Addison-Wesley, 1999.

BECK, K.; ANDRES, C. *Extreme Programming Explained*: embrace change. 2. ed. Boston: Addison-Wesley, 2004.

COAD, P.; LEFEBVRE, E.; DE LUCA, J. *Java Modeling in Color with UML*: enterprise components and process. New Jersey: Prentice-Hall, 1999.

PALMER, S.; FELSING, J. M. *A Practical Guide to Feature-Driven Development*. New Jersey: Prentice-Hall, 2002.

SCHWABER, K. *Agile Project Management with SCRUM*. Washington: Microsoft Press, 2004.

SCHWABER, K.; SUTHERLAND, J. *Scrum Guide*. Disponível em: <http://scrum.org>. Acesso em: 23 jan. 2012.

TELES, V. M. *Extreme Programming*. São Paulo: Novatec, 2004.

UNIDADE 8

Grails

8.1 Primeiras palavras

Como discutido, as atividades de construção e implantação são o resultado final de uma e, possivelmente, de muitas iterações de processo de Engenharia Web. Esta unidade apresenta o framework web Grails, que é bem inerente à filosofia ágil e poderia ser utilizado durante as atividades de construção e implantação do Arcabouço de Processo de Engenharia Web.

Na realidade, Grails, por si só, não é ágil, pois nenhuma ferramenta por si só pode ser ágil. No entanto, ele se encaixa muito bem dentro de metodologias ágeis. Como discutido na Unidade 7, as metodologias ágeis são as técnicas para desenvolvimento de software que promovem uma série de princípios de boas práticas. Grails é inerente a muitas dessas práticas, incluindo as seguintes:

- *Ser adaptável à mudança*: graças ao seu mecanismo de *autoreloading* e natureza dinâmica, Grails promove a mudança e o desenvolvimento iterativo.
- *Entrega precoce do software funcional*: a simplicidade de Grails permite uma abordagem de desenvolvimento rápido de aplicações, aumentando as probabilidades de entrega rápida. Além disso, Grails prega a automação dos testes (unitários e de integração), conscientizando os desenvolvedores de sua importância.
- *Simplicidade é essencial*: grails visa proporcionar simplicidade. Ou seja, Grails tem conceitos complexos, como ORM (Object-Relational Mapping),[23] porém ela encapsula esses conceitos em uma simples API. Mapeamento objeto-relacional é uma técnica de desenvolvimento de software que é utilizada com o objetivo de reduzir os problemas inerentes à utilização conjunta de banco de dados relacionais e o paradigma de desenvolvimento orientado a objetos. As tabelas do banco de dados são representadas por meio de classes, e os registros de cada tabela são representados como instâncias das classes correspondentes.
- *Equipe entusiasmada, auto-organizada com o ambiente certo*: desde que Grails permite aos desenvolvedores centrar-se principalmente na lógica de negócios necessários para resolver um problema específico – ao invés de aspectos relacionados à configuração de sua aplicação –, a equipe está mais propensa a ter desenvolvedores entusiasmados.

23 Mapeamento Objeto-Relacional.

8.2 Problematizando o tema

Ao final desta unidade espera-se que o leitor esteja apto a usar o framework Grails no desenvolvimento de aplicações Web. Dessa forma, esta unidade pretende discutir as seguintes questões:

- Quais são as principais características do framework Grails?
- Como esse framework pode auxiliar a equipe de Engenharia Web nas atividades de construção e implantação do Arcabouço de Processo de Engenharia Web?

8.3 Grails

Grails é um framework web baseado no padrão arquitetural MVC (discutido na Unidade 4) que utiliza a linguagem Groovy, executa sobre a máquina Virtual Java (JVM) e objetiva a alta produtividade no desenvolvimento de aplicações Web. Ele combina os principais frameworks (Hibernate,[24] Spring[25] etc.) utilizados na plataforma Java e respeita o paradigma *Convention Over Configuration* (Convenção ao invés de Configuração).

Groovy. Groovy[26] é uma linguagem dinâmica, ágil para a plataforma Java inspirada em Python[27] e Ruby[28], que possui sua sintaxe semelhante à de aplicações desenvolvidas em Java. Apesar de poder ser usada como uma linguagem de *script*, ou seja, não gerar arquivos executáveis e não precisar ser compilada, Groovy não se limita a isso. Aplicações feitas nessa linguagem podem ser compiladas utilizando-se de um compilador Java, gerando Java Bytecodes (mesmo formato da compilação de uma aplicação escrita em Java); além disso, podem ser utilizadas em aplicações escritas puramente em Java.
A linguagem foi desenvolvida em 2004 por James Strachan. A sua sintaxe é extremamente parecida com a do Java; além disso, é possível "integrar" aplicações Java e Groovy de forma transparente. O Groovy, inclusive, simplifica a implementação por "adicionar" dinamicamente às suas classes os métodos de acesso (gets e sets), economizando tempo e esforço. O objetivo de Groovy é simplificar a sintaxe de Java para representar comportamentos dinâmicos, como consultas a banco de dados, escritas e leituras de arquivos e geração

24 Disponível em: <http://www.hibernate.org/>. Acesso em: 24 jan. 2012.
25 Disponível em: <http://springsource.org>. Acesso em: 24 jan. 2012.
26 Disponível em: <http://groovy.codehaus.org/>. Acesso em: 24 jan. 2012.
27 Disponível em: <http://www.python.org/>. Acesso em: 24 jan. 2012.
28 Disponível em: <http://www.ruby-lang.org/pt/>. Acesso em: 24 jan. 2012.

de objetos em tempo de execução ao invés de compilação (KÖNIG, 2007).
Convention Over Configuration (CoC). O CoC é um paradigma que visa diminuir a quantidade de decisões que o desenvolvedor precisa tomar, tendo como "padrão" algo que é comumente usado, uma convenção. Se o padrão escolhido pelo framework for o que o desenvolvedor precisar, este não gasta tempo tendo que alterá-lo; entretanto, se ele necessitar de algo diferente, fica livre para configurar da forma que desejar. No caso do Grails, ele assume diversas configurações, tais como as de banco de dados, as de localização do código-fonte, entre outras.

Esta unidade discute como o framework Grails pode ser utilizado para realizar a implementação de duas aplicações web simples: a Seção 8.4 descreve uma implementação parcial da aplicação **LivrariaVirtual.com.br** discutida nas unidades anteriores, enquanto a Seção 8.5 descreve a implementação da aplicação *Agenda de Contatos* que incorpora algumas funcionalidades AJAX. O código-fonte dessas aplicações encontra-se disponível no CD suplementar que acompanha o livro.

8.3.1 Configuração do ambiente de desenvolvimento

Esta seção apresenta algumas dicas importantes no processo de instalação e configuração do ambiente de desenvolvimento necessário para compilar e executar os exemplos discutidos nesta unidade. A instalação do ambiente é simples e consiste de poucos passos:

Instalação Java. O Java Development Kit (JDK) – versão igual ou superior a 1.5 – será necessário para executar os exemplos apresentados nesta unidade. A última versão do JDK pode ser obtida em <http://java.com/pt_BR/download/index.jsp> (Este livro utiliza o JDK versão 1.6.0_22).

Dicas importantes: (1) a variável de ambiente **JAVA_HOME** precisa apontar para o diretório no qual o JDK foi instalado. (2) Digite **java -version** em um terminal para verificar se o Java foi instalado corretamente (Figura 30).

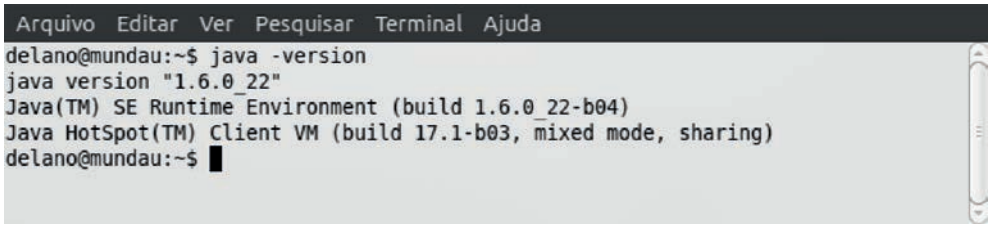


Figura 30 Verificação da instalação do Java.

Instalação Grails. A última versão do Grails pode ser obtida em <http://grails.org/Download> (Este livro utiliza a versão 1.3.6).

Após realizar o *download*, execute os seguintes passos:

- Descompacte o Grails em um diretório.
- Crie uma variável de ambiente **GRAILS_HOME** e faça-o apontar para o diretório no qual o Grails foi descompactado.
- Adicione **GRAILS_HOME/bin** na variável de ambiente **PATH**.

Dica importante: digite *grails* em um terminal para verificar se o *Grails* foi instalado corretamente e está pronto para uso (Figura 31). Para mais informações sobre a instalação do Grails, o leitor pode consultar o seguinte link <http://grails.org/Installation+Portuguese>.

Figura 31 Verificação da instalação do Grails.

Instalação NetBeans. O NetBeans IDE será utilizado para executar os exemplos apresentados nesta unidade. A última versão do NetBeans IDE pode ser obtida em <http://netbeans.org/downloads> (Este livro utiliza a versão 6.9.1).

Dicas importantes: (1) faça o *download* da distribuição **Java** ou da distribuição **Tudo** (Figura 32). São as únicas distribuições que vêm com suporte a linguagem de programação Groovy/Grails e ao desenvolvimento Web. (2) O seguinte link <http://netbeans.org/kb/docs/web/grails-quickstart_pt_BR.html> contém um breve tutorial sobre a utilização do NetBeans IDE + Grails no desenvolvimento de aplicações Web.

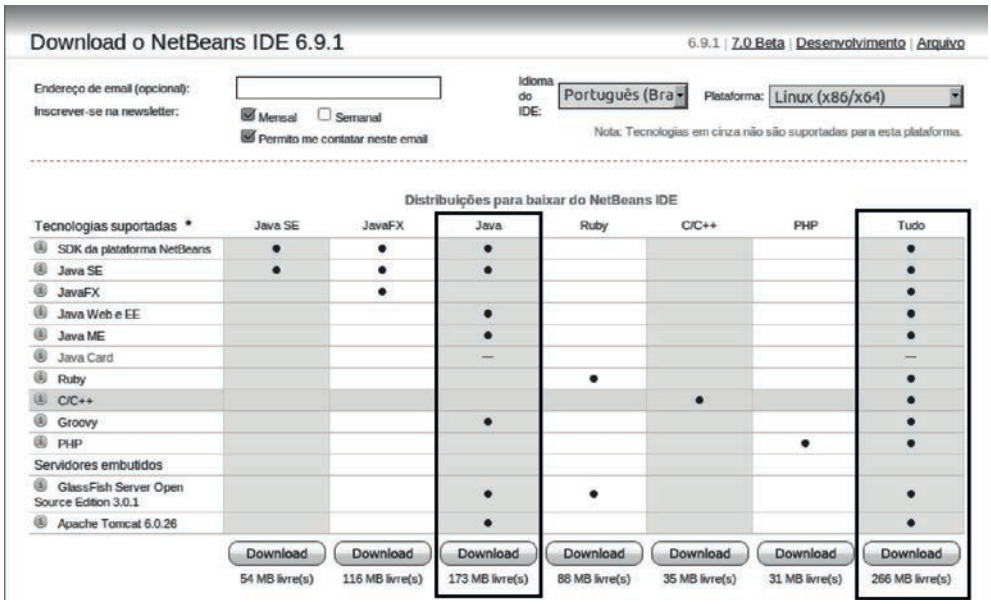


Figura 32 Distribuições do NetBeans IDE 6.9.1.

Ao final da instalação, é necessário configurar o NetBeans IDE (localização do diretório no qual foi desempacotado o Grails). Abra o NetBeans ID, vá para a janela **Ferramentas**/**Opções** e defina a localização do Grails no Painel "Groovy" (Figura 33).

Figura 33 Configuração do NetBeans IDE.

Na realidade, o Grails não exige a utilização de um IDE. Todos os comandos necessários ao desenvolvimento poderiam ser feitos em um terminal de comando. No entanto, assim como em outras linguagens de programação, o IDE torna ágil o processo de desenvolvimento ao integrar diferentes funcionalidades (edição, compilação e execução etc.) e abstrair a sintaxe dos comandos necessários relacionados a essas atividades. O NetBeans IDE será utilizado neste livro, mas o leitor deve se sentir a vontade para usar outro IDE se este já está instalado e configurado.

Banco de dados. A instalação do Grails já incorpora uma cópia do HSQLDB, um banco de dados relacional totalmente implementado em Java. HSQLDB

é ótimo para aplicações de demonstração, mas em algum momento os desenvolvedores precisarão de um banco de dados mais robusto, tais como MySQL, Postgresql e Oracle. Desde que o GORM (Grails Object-Relational Mapping) é uma fachada sobre o framework *hibernate*, qualquer banco de dados que possua um driver JDBC e um dialeto *hibernate* pode ser utilizado.

8.4 Aplicação LivrariaVirtual

Nesta seção, será apresentado o processo de criação do projeto Grails **LivrariaVirtual** que realiza a implementação parcial da aplicação **LivrariaVirtual.com.br**, discutida nas unidades anteriores.[29]

- Para criar uma aplicação Grails no NetBeans, escolha “Arquivo”, “Novo projeto” (Ctrl-Shift-N) e selecione “Aplicativo do Grails” na categoria “Groovy”. Em seguida, clique em “Próximo” (Figura 34).
- Em nome do projeto, digite **LivrariaVirtual**; em Localização do projeto, selecione a pasta na qual o aplicativo será criado. Clique em “Finalizar”. O NetBeans IDE executa o comando “*grails create-app*”,[30] apresentando a saída na janela Saída.

Caso esses passos forem realizados com sucesso, o projeto da aplicação (hierarquia de diretórios) está criado. Ou seja, foi criada uma série de arquivos e diretórios para o projeto. Essa hierarquia de diretórios segue o paradigma *Convention over Configuration*. Ou seja, os desenvolvedores seguem as convenções e já sabem *a priori* onde se encontram todos os elementos que compõem a aplicação em desenvolvimento. Um *overview* desses diretórios (abas Projetos e Arquivos – Figura 35) é apresentado na Tabela 3.

29 O código-fonte dessa aplicação encontra-se no CD suplementar que acompanha o livro.
30 Para obter a lista completa de comandos Grails, execute o comando *grails help* em um terminal.

Tabela 3 Projeto Grails: overview dos diretórios.

Aba Projetos	Aba Arquivos	Descrição
Configuração	grails-app/conf	É onde se encontram as configurações da aplicação, tais como a configuração do banco (DataSource. groovy), e onde podem ser feitas as configurações de inicialização (BootStap.groovy), entre outros.
Controladores	grail-app/controllers	É onde se encontra o C do MVC. Ou seja, onde se encontram os controladores.
Classes de domínio	grails-app/domain	É onde se encontra o M do MVC. Ou seja, onde se encontram as classes de Domínio, ou modelos.
Pacotes de mensagens	grails-app/i18n	É onde se encontram os arquivos relacionados à internacionalização.
Serviços	grails-app/services	É onde se encontram as classes utilizadas na camada de serviços (Web Services).
Biblioteca de marcas	grails-app/taglib	É onde se encontram as bibliotecas de marcas (taglibs) criadas pelo usuário.
Visualizações e layouts	grails-app/views grails-app/templates	É onde se encontra o V do MVC. Ou seja, onde se encontram as visões (arquivos.gsp – Groovy Server Pages). É onde se encontram os templates. Em Grails, templates são visões incluídas em outras visões.
Bibliotecas	lib	É onde se encontram as bibliotecas externas, tais como driver JDBC de conexão a banco de dados.
Pacotes de código-fonte Groovy	src/groovy	É onde se encontram outros códigos-fonte Groovy que não são modelos, controladores, visões ou serviços.
Pacotes de código-fonte Java	src/Java	É onde se encontram outros códigos-fonte Java que não são modelos, controladores, visões ou serviços.
Teste de integração	test/integration	É onde se encontram os testes de integração da aplicação.
Testes unitários	test/unit	É onde se encontram os testes unitários da aplicação.
Aplicação Web	web-app	css – folha de estilo. images – imagens. js – código JavaScript. META-INF e WEB-INF – arquivos relacionados à implantação da aplicação.

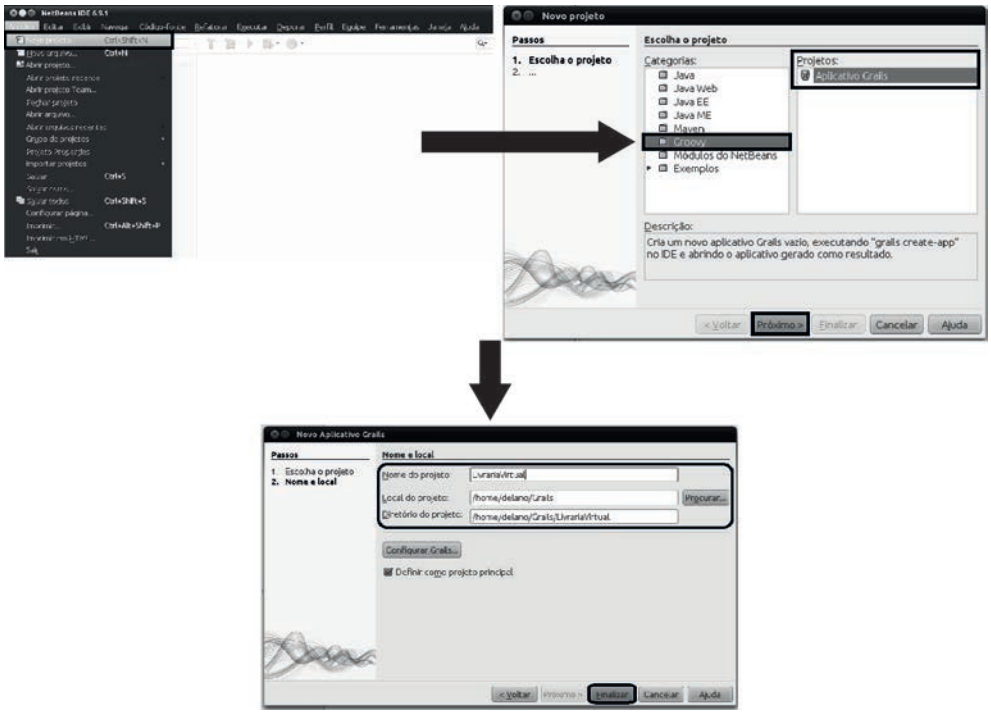


Figura 34 Criação do Projeto LivrariaVirtual.

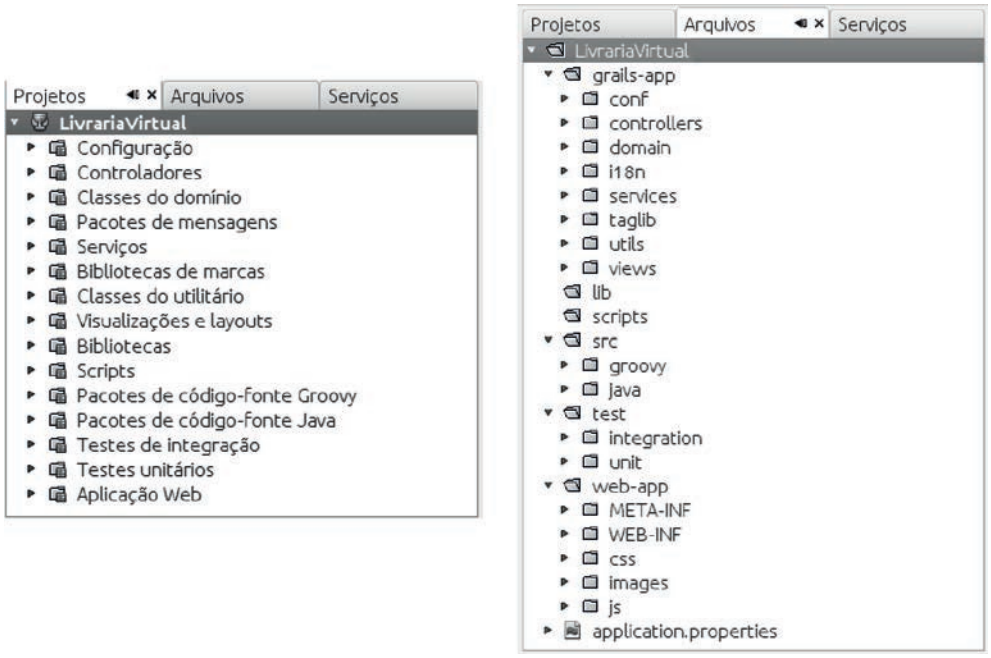


Figura 35 Projeto LivrariaVirtual (abas Projetos e Arquivos).

8.4.1 Implementando as primeiras funcionalidades

Agora que o projeto foi criado, o próximo passo é implementar as primeiras funcionalidades da aplicação **LivrariaVirtual.com.br**. Conforme discutido na Unidade 2 e reapresentado aqui, a principal motivação do desenvolvimento dessa aplicação é:

LivrariaVirtual.com.br permitirá aos clientes adquirir produtos (livros, CDs, DVDs etc.) e recebê-los em sua residência.

Dessa forma, é justificável iniciar a implementação da aplicação pelas operações de CRUD de produtos (livros, CDs, DVDs etc.). CRUD é o acrônimo para *Create*, *Read*, *Update* e *Delete*. Ou seja, as operações de criação, acesso, atualização e remoção de produtos.

Levando em consideração o padrão MVC (discutido na Unidade 4), Produtos fazem parte do modelo (o M do MVC) da aplicação. Assim, é necessário criar uma classe de Domínio[31] que representa produtos.

- Para criar uma Classe de domínio, clique com o botão direito no *mouse* no nó Classes de domínio e escolha Novo – Classe de domínio do Grails (Figura 36).
- Digite "Produto" como o *Nome do artefato*, e "lv" como o *Pacote* e clique em Finalizar. A classe de domínio "Produto.groovy" é criada no nó Classes de domínio.
- O IDE NetBeans IDE executa o comando "*grails create-domain-class*", apresentando a saída na janela Saída.
- Abra a classe Produto e insira os atributos (nome, preço e quantidade) dessa classe (ver classe de projeto na Figura 20 – Unidade 4).

Observações importantes:

- O atributo identificador é gerado automaticamente pelo Grails. Logo, não é necessário incluí-lo na implementação da classe Produto.
- O atributo descrição será discutido adiante nesta unidade. Por enquanto, ele será desconsiderado.
- Os tipos de dados utilizados no Groovy são os mesmos do Java, portanto com as mesmas características.

31 Em Grails, os modelos são denominados de classes de domínio.

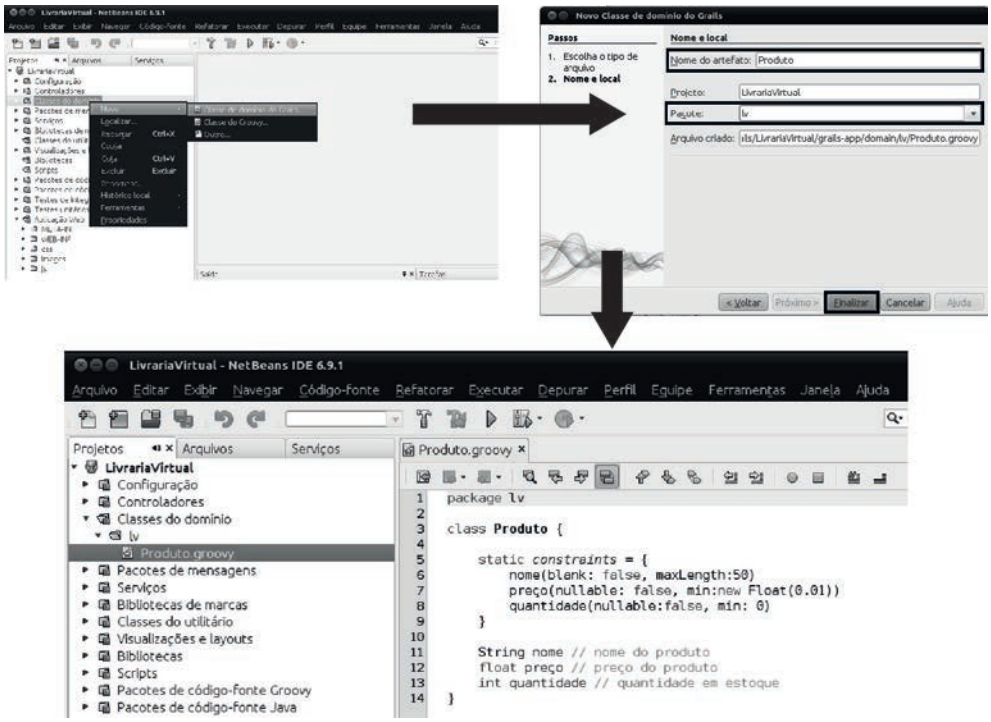


Figura 36 Criação da Classe de Domínio Produto.

8.4.1.1 Validação de dados

O bloco *static constraints* permite que os desenvolvedores coloquem regras de validação nas classes de domínio. Por exemplo, é possível impor restrições sobre o tamanho máximo de um atributo String (por padrão, o tamanho é 255 caracteres). Além disso, é possível garantir que campos de texto (*Strings*) correspondem a um determinado padrão (como um endereço de e-mail ou URL). E, por fim, é possível até mesmo tornar campos opcionais ou obrigatórios. Além dessas validações, o bloco *static constraints* também possibilita definir a ordem em que os atributos de um modelo são apresentados nas visões associadas. A Figura 36 descreve o uso do bloco *static constraints* para validar os atributos da classe Produto e definir a ordem em que os atributos dessa classe são apresentados. A Tabela 4 apresenta algumas restrições com descrição e exemplos de utilização.

Tabela 4 Regras de restrições.

Nome	Exemplo	Descrição
blank	nome(blank:false)	Coloque *false* se o valor da String não pode estar em branco.
email	e-mail(email:true)	Coloque *true* se a String necessitar ser um endereço de e-mail válido.
inList	nome(inList:[“João”,“José”])	O valor deve estar contido na lista.
length	nome (length:5..15)	Usa uma faixa para restringir o tamanho de uma String ou array.
min	quantidade(min:0)	Define o valor mínimo.
minLength	nome(minLength:5)	Define o tamanho mínimo de uma String ou array.
matches	nome(matches:/[a-zA-Z]/)	Corresponde a uma expressão regular fornecida.
max	quantidade(max:100)	Define o valor máximo.
maxLength	nome(maxLength:15)	Define o tamanho máximo de uma String ou array.
notEqual	nome(notEqual:”Fred”)	Não deve ser igual ao valor especificado.
nullable	preço(nullable:false)	Coloque *false* se o valor do atributo não poder ser nulo.
range	quantidade(range:5..15)	Valor deve estar dentro do intervalo especificado.
size	list(size:5..15)	Usa uma faixa para restringir o tamanho de uma coleção.
unique	nome(unique:true)	Defina como *true* se o valor do atributo não pode repetir.
url	url(url:true)	Coloque *true* se o valor da String precisar ser um endereço URL válido.

Na realidade, na aplicação **LivrariaVirtual.com.br**, instâncias de Produto não são criadas – apenas de suas subclasses (Cd, Dvd e Livro). Dessa forma, será necessário criar três classes de domínio: Cd, Dvd e Livro.

- Crie, usando os mesmos passos da criação de *Produto*, a classe *Cd* do pacote *lv*. Abra a classe *Cd* (Figura 37) e insira o atributo *artista*, que representa o nome do artista (ou conjunto musical) do CD. Por fim, acrescente as validações no atributo *artista*.

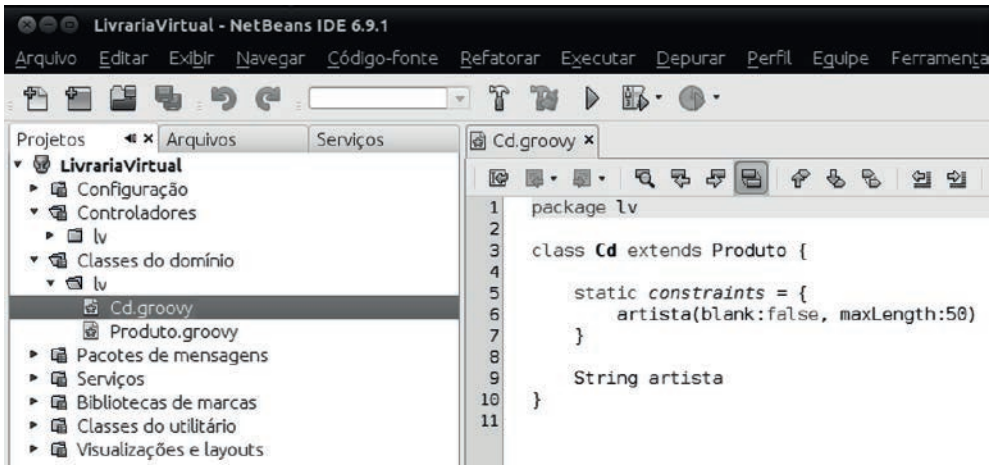


Figura 37 Criação da Classe de Domínio Cd.

Crie, usando os mesmos passos da criação de *Produto*, a classe *Dvd* do pacote *lv*. Abra a classe *Dvd* (Figura 38) e insira o atributo *diretor*, que representa o nome do diretor do filme gravado no DVD. Por fim, acrescente as validações de dados no atributo *diretor*.

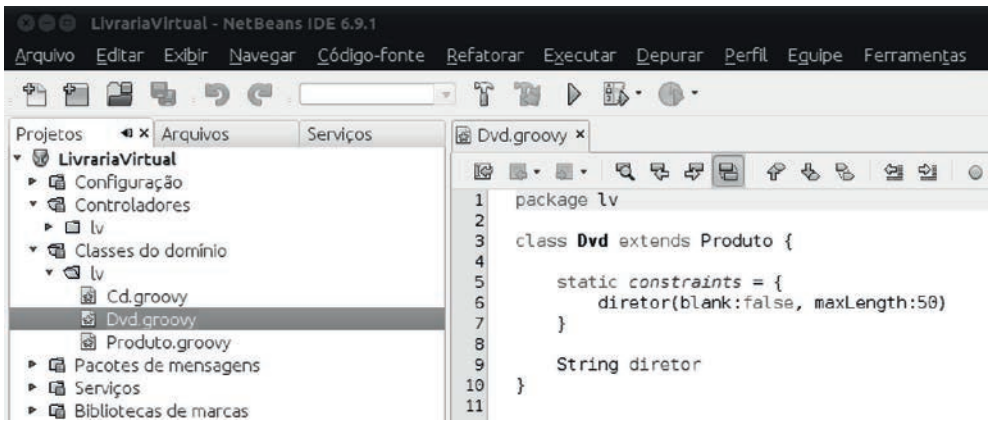


Figura 38 Criação da Classe de Domínio Dvd.

Crie, usando os mesmos passos da criação de *Produto*, a classe *Livro* do pacote *lv*. Abra a classe *Livro* (Figura 39) e insira o atributo *autor*, que representa o nome do(s) autor(es) do livro. Por fim, acrescente as validações de dados no atributo *autor*.

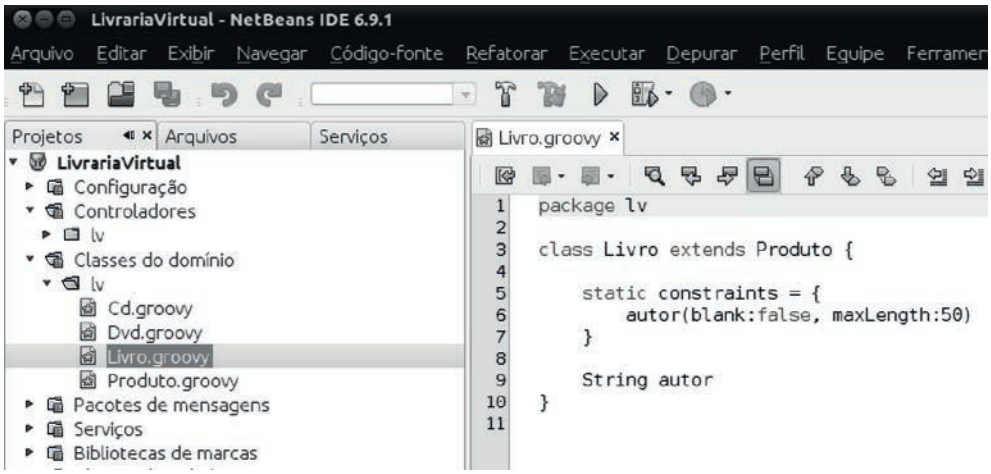


Figura 39 Criação da Classe de Domínio Livro.

Como mencionado anteriormente, programadores não precisam se preocupar em criar os métodos *getters* e *setters*: eles são gerados dinamicamente pelo Groovy. Grails por meio do mecanismo GORM (*Grails Object-Relational Mapping*) realiza um mapeamento automático entre modelos (classes de domínio) e tabelas em um SGBD. Dessa forma, o Groovy gera outros métodos dinâmicos bastante úteis, cujos nomes são autoexplicativos:

- **Produto.save()** armazena os dados na tabela Produto (do SGBD).
- **Produto.delete()** apaga os dados da tabela Produto.
- **Produto.list()** retorna uma lista de Produtos.
- **Produto.get()** retorna uma única instância de Produto.

Todos esses e outros métodos estão disponíveis quando os desenvolvedores precisam deles. Note que Produto não estende nenhuma classe pai ou implementa uma interface mágica. Graças aos recursos de metaprogramação Groovy, esses métodos simplesmente aparecem quando necessários.

Observação: apenas as classes presentes no diretório *grails-app/domain* (ou seja, classes de domínio) possuem os métodos relacionados à persistência de dados: **save()**, **delete()**, **list()** e **get()**.

8.4.1.2 *Scaffolding*

Agora que as Classes de domínio estão criadas, é preciso criar os controladores e as visões relacionados a essas classes de domínio. Na geração de tais artefatos de software será utilizada a abordagem *scaffolding*, que é um termo

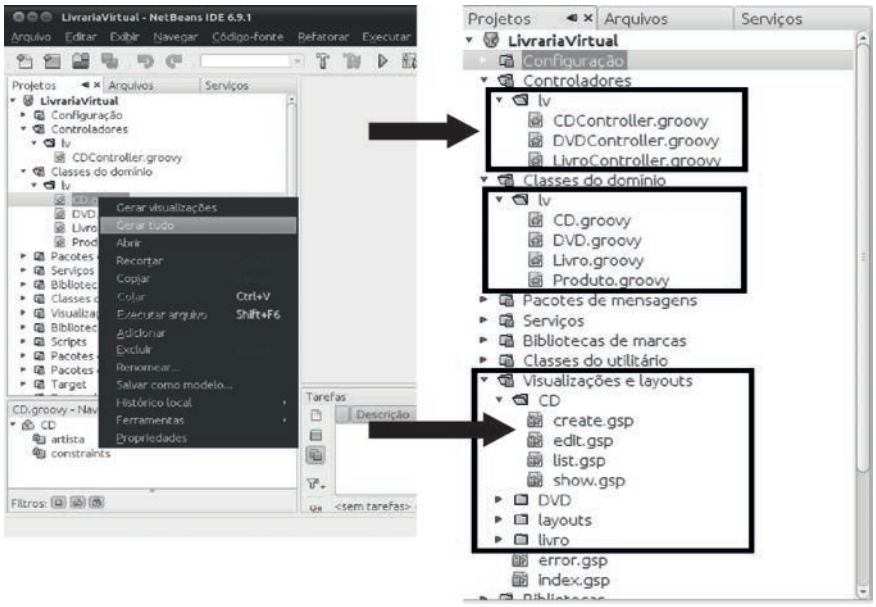
cunhado pelo framework Rails[32] e adotado pelo Grails para a geração dos artefatos (controladores, visões etc.) que implementam as operações CRUD. Esse termo foi estendido e atualmente é possível criar código de autenticação/autorização, testes unitários, entre outras operações.

Scaffolding pode ser dinâmico ou estático. No *scaffolding* dinâmico, o controlador e as visões são gerados em tempo de execução. Essa abordagem simplifica o desenvolvimento, pois nenhum código relacionado aos controladores e visões precisa ser desenvolvido. O *scaffolding* dinâmico pode servir a vários propósitos, por exemplo, é bastante útil na criação das interfaces de administração de uma aplicação Web. No entanto, ele não é útil quando a equipe web deseja personalizar o sistema para seus propósitos, principalmente as visões geradas em tempo de execução. Já o *scaffolding* estático produz, a partir de *templates*, o código relacionado aos controladores e visões que pode ser personalizado pela equipe Web. Levando em consideração esses aspectos, este livro adotará o *scaffolding estático*. Para mais detalhes sobre o *scaffolding* dinâmico, o leitor interessado deve consultar <http://netbeans.org/kb/docs/web/grails-quickstart_pt_BR.html>, que apresenta um exemplo do uso dessa abordagem no desenvolvimento de uma aplicação Web.

- Para criar um controlador e as visões (empregando *scaffolding* estático) relacionados à Classe de domínio *Cd*, clique com o botão direito do *mouse* no nó *Cd.groovy* e escolha “Gerar Tudo” (Figura 40). Faça o mesmo para as outras duas classes de domínio, *Dvd* e *Livro*.
- O NetBeans IDE executa o comando “grails generate-all”, apresentando a saída na janela Saída. Ou seja, ele cria a classe controlador *CdController.groovy* e correspondente conjunto de Groovy Server Pages (GSPs) no diretório produto de Visualizações e Layouts (grails-app/views). Para cada ação (método) *x* em um controlador, existe uma correspondente visão (arquivo *x.gsp*). Ou seja, a ação *list* tem o correspondente *list.gsp*, enquanto a ação *create* tem o correspondente *create.gsp*. A Figura 40(b) ilustra os controladores e visões gerados automaticamente pelo *scaffolding*.

Aqui é possível ver novamente os benefícios do paradigma *Convention Over Configuration* em ação: nenhum arquivo XML é necessário para associar esses elementos. As classes de domínio estão associadas a controladores baseado em seus respectivos nomes (**Cd.groovy** → **CdController.groovy**). Toda ação em um controlador está associada a uma visão baseada em seu nome (*list* → *list.gsp*). Desenvolvedores podem configurar para que o mapeamento seja feito de outra maneira. No entanto, na maioria das vezes, basta seguir a convenção e sua aplicação funcionará perfeitamente sem maiores configurações.

32 Disponível em: <http://rubyonrails.org/>. Acesso em: 24 jan. 2012.



a) b)

Figura 40 *Scaffolding* estático das classes de Domínio.

Agora que o CRUD de Produtos está pronto, a aplicação pode ser executada. Porém, antes de executá-la, crie algumas instâncias de Produto (CD, DVD e livro) na classe *BootStrap.groovy*, que está no nó *Configuração*. O código adicionado à classe é apresentado na Figura 41. Essa classe é executada durante o *boot* da aplicação e serve, entre outros propósitos, para inicializar a aplicação – por exemplo, criando algumas instâncias de objetos.

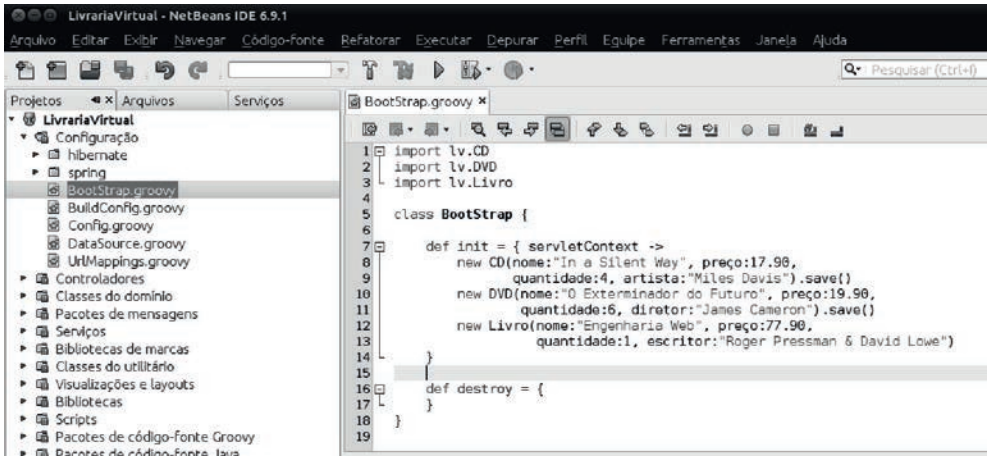
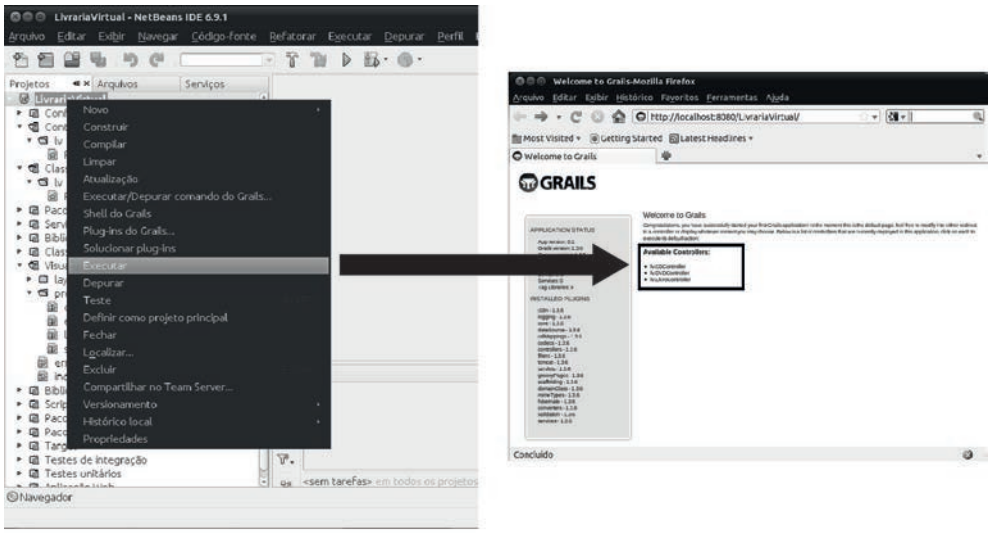


Figura 41 Inserção de algumas instâncias de Produto (CD, DVD, livro).

Para executar a aplicação, clique no botão direito do *mouse* no projeto LivrariaVirtual e escolha Executar – Figura 42(a). A aplicação é implantada no servidor Web Jetty,[33] como pode ser visto na janela Serviços do NetBeans IDE.

33 Disponível em: <http://jetty.codehaus.org/jetty>. Acesso em: 23 jan. 2012.

- A URL <http://localhost:8080/LivrariaVirtual> é impressa na janela Saída. Se o navegador não abrir automaticamente, cole a URL em um navegador e a aplicação será acessada. Os três controladores serão listados – Figura 42(b).



a) b)

Figura 42 Execução da Aplicação LivrariaVirtual.com.br.

- Ao clicar no link *lv.CdController*, o CD inserido anteriormente (BootStrap.groovy) será apresentado (Figura 43).

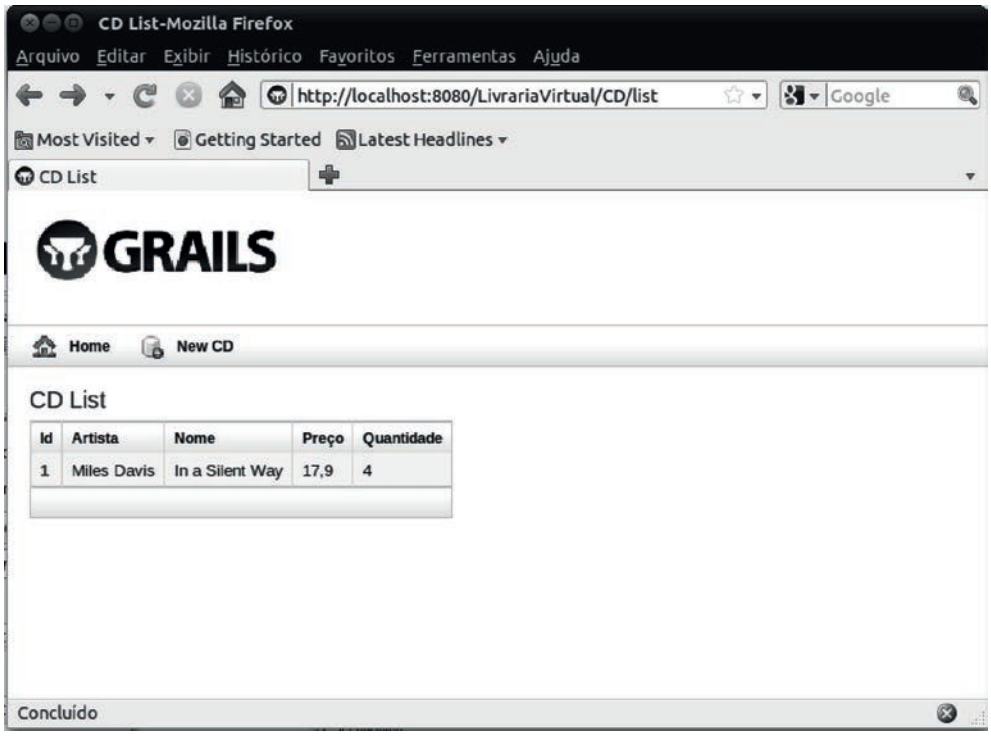


Figura 43 Lista de CDs.

Grails usa uma convenção (Figura 44) para automaticamente configurar o caminho para uma ação em particular. A URL a seguir pode ser entendida da seguinte forma: "execute a ação *list* do controlador *cd* – um dos controladores da aplicação LivrariaVirtual hospedada na porta *8080* do servidor *localhost*".

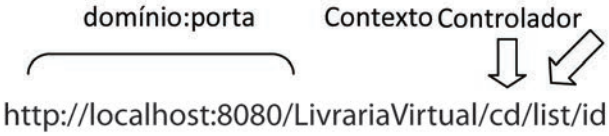


Figura 44 Convenção na nomenclatura de URLs.

A regra principal de roteamento no Grails forma URLs segundo o padrão /controller/action/id, em que *controller* é o controlador responsável por atender a requisição especificada por aquela URL, *action* é um método dentro da classe e *id* é um parâmetro opcional passado para identificar um objeto qualquer sobre o qual a ação será efetuada. A visão associada a essa ação geralmente é invocada (o controlador pode redirecionar para outra visão). Ou seja, a visão *action* (arquivo controller/action.gsp) está associada por padrão a *action* invocada. Exemplo: /produto/edit/1 invocaria a visão produto/edit.gsp.

- Clique em "New CD" e crie um novo CD. Quando você clicar em "Create", observe que você poderá editar ("Edit") ou excluir ("Delete") o CD. E, por fim, ao clicar em "CD List", a nova entrada é refletida na lista de CDs

(Figura 45). É importante salientar que para a criação de DVDs e livros os passos são análogos.

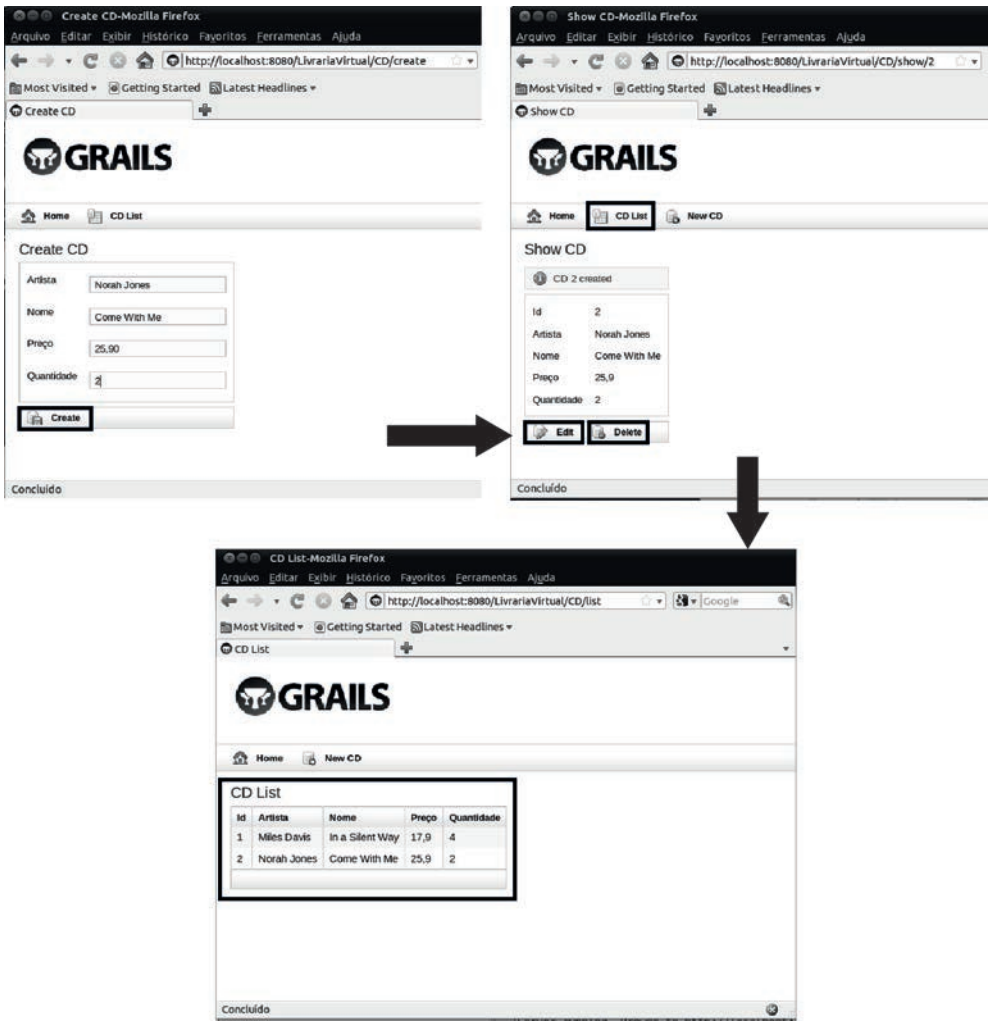


Figura 45 Criação de um novo CD.

- Clique em “New CD” e crie um novo CD com valores inválidos (Figura 46). Pode-se notar nessa figura que o CD não foi criado – as validações de dados foram violadas.

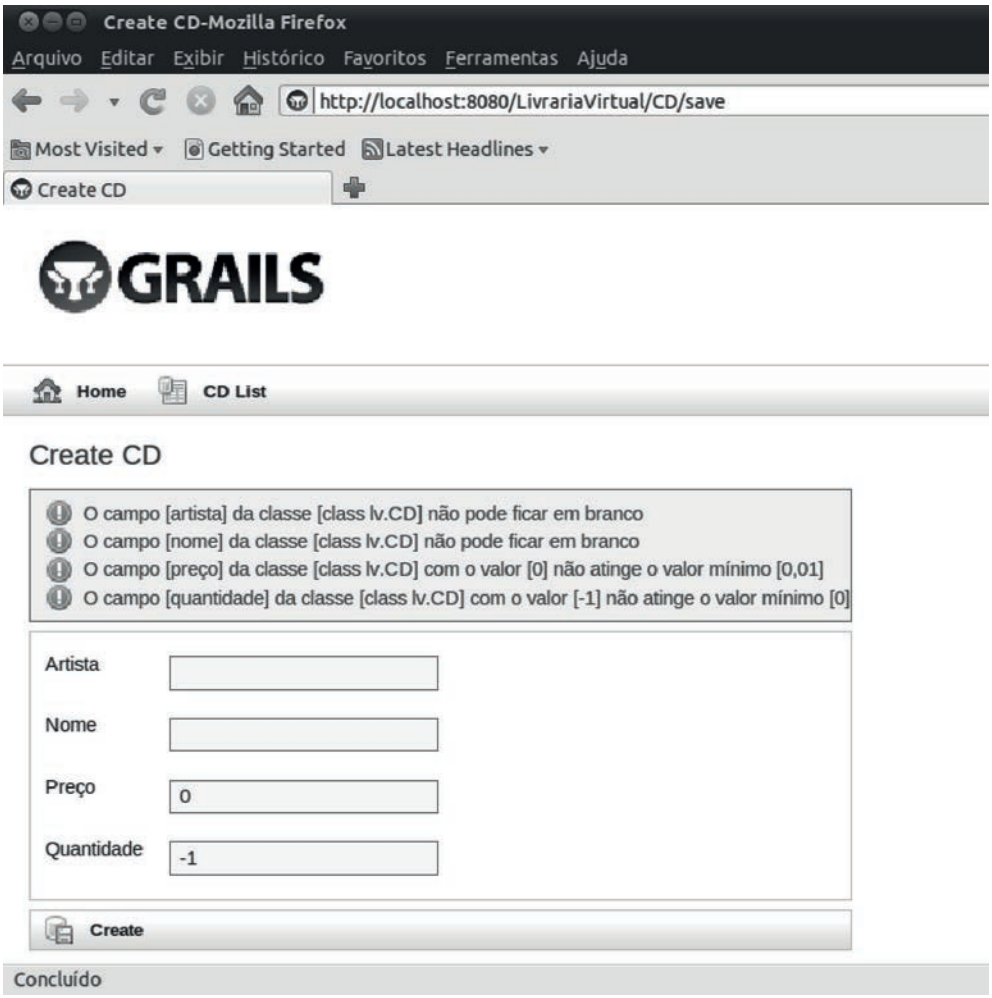


Figura 46 Criação de um CD com valores inválidos.

8.4.2 Descrição de produto

Conforme discutido na Unidade 4, cada produto tem um atributo *descricao* representado como uma classe denominada *Descricao*, composta de quatro objetos: DescriçãoTécnica, Imagem, Áudio e Vídeo. Esta seção apresenta os passos para implementar essa classe de domínio e o relacionamento de um para um entre as classes de domínio *Produto* e *Descricao*.

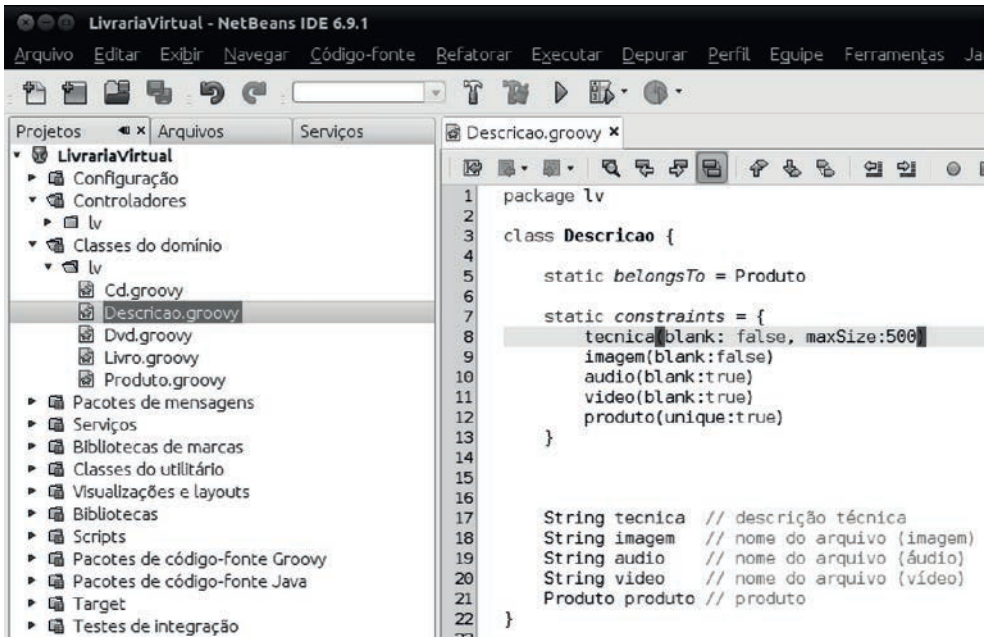


Figura 47 Criação da classe de domínio Descrição.

- Seguindo os passos descritos anteriormente, crie uma nova classe de domínio denominada *Descricao* no pacote lv (Figura 47), insira os atributos: *tecnica* (representa a descrição técnica), *imagem* (representa a imagem), *audio* (representa o áudio) e *video* (representa o vídeo). Por fim, acrescente as validações de dados nos atributos. Conforme discutido, o áudio e o vídeo não são obrigatórios (*blank:true*). O atributo *produto* representa o produto ao qual ele está associado.
- Abra a classe de domínio *Produto* (Figura 48) e insira o seguinte comando: *static hasMany[descricao:Descricao]*.

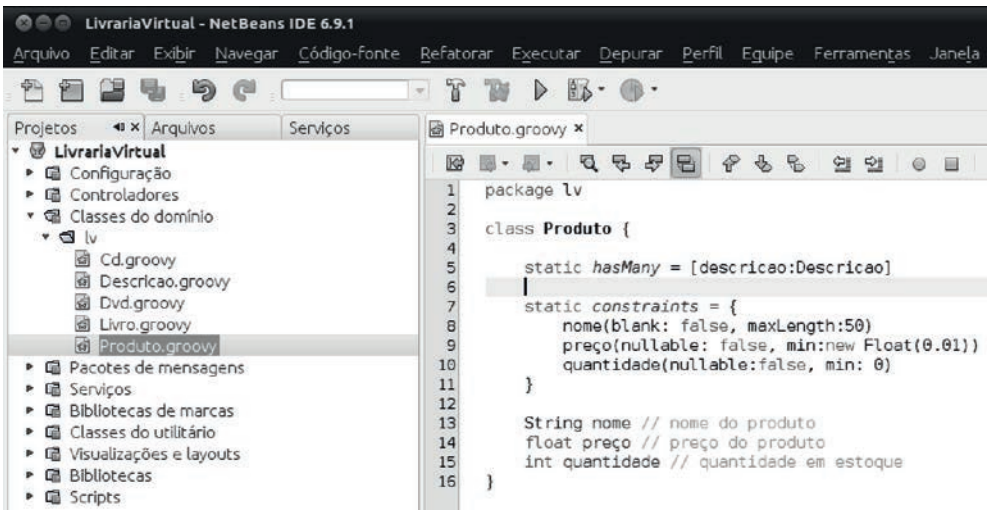


Figura 48 Classe Produto associada a uma Descrição.

Observação: os comandos *static hasMany* na classe *Produto*, *static belongsTo* e *produto(unique:true)* na classe *Descricao*, usados em conjunto, implementam um mapeamento bidirecional um para um entre *Produto* e *Descricao*. Na realidade, em Grails os comandos *hasMany* e *belongsTo* são, geralmente, utilizados para implementar um relacionamento um para muitos. No entanto, ao adicionar a restrição *produto(unique:true)* esse relacionamento se torna um para um. Ou seja, o comando *belongsTo* garante que uma descrição pertença a um único produto, onde esse produto pode repetir para diferentes descrições. Logo, duas descrições podem pertencer ao mesmo produto. Porém, ao adicionar a restrição *produto(unique:true)*, isso garante que não irá existir duas descrições pertencentes a um mesmo produto (o atributo produto é único).

- Seguindo os passos descritos anteriormente, gere novamente o controlador e as visões relacionados às subclasses de *Produto* (Cd, Dvd e Livro) devido mudanças na classe *Produto*.

Observações importantes: (1) lembre-se que sempre que o modelo for modificado, o controlador e as visões associados a ele precisam ser alterados, e (2) é necessário confirmar o desejo em substituir os arquivos existentes (janela saída do NetBeans IDE).

- Seguindo os passos descritos anteriormente, gere o controlador e as visões relacionadas à classe de domínio *Descricao*.

8.4.3 *Upload* de arquivos

Conforme discutido, os atributos *imagem*, *audio* e *video* são arquivos que descrevem um produto. Dessa forma, é necessária a implementação da funcionalidade de *upload* de arquivos. Os passos a seguir descrevem como implementar tal funcionalidade no contexto da classe de domínio *Descricao* – são passos que descrevem algumas alterações a serem realizadas nos arquivos gerados automaticamente (controlador e visões) com o objetivo de permitir o *upload* de arquivos.

- Na visão create (*grails-app/views/descricao/create.gsp*) do controlador *DescricaoController*, realize as seguintes alterações:

Tabela 5 Alteração na visão create.

Linha 26	<g:form action=”save” >
Novo texto	<g:uploadForm action = “save”>
Linha 45	<g:textField name=”imagem” value=”${descricaoInstance?.imagem}” />
Novo texto	<input type = “file” id=”imagem” name=”imagem”/>
Linha 54	<g:textField name=”audio” value=”${descricaoInstance?.audio}” />
Novo texto	<input type = “file” id=”audio” name=”audio”/>
Linha 63	<g:textField name=”video” value=”${descricaoInstance?.video}” />
Novo texto	<input type = “file” id=”video” name=”video”/>
Linha 82	</g:form>
Novo texto	</g:uploadForm>

- Na visão edit (*grails-app/views/descricao/edit.gsp*) do controlador DescricaoController, realize as seguintes alterações:

Tabela 6 Alteração na visão edit.

Linha 27	<g:form action=”save” >
Novo texto	<g:uploadForm action = “save”>
Linha 48	<g:textField name=”imagem” value=”${descricaoInstance?.imagem}” />
Novo texto	<input type = “file” id=”imagem” name=”imagem”/>
Linha 57	<g:textField name=”audio” value=”${descricaoInstance?.audio}” />
Novo texto	<input type = “file” id=”audio” name=”audio”/>
Linha 66	<g:textField name=”video” value=”${descricaoInstance?.video}” />
Novo texto	<input type = “file” id=”video” name=”video”/>
Linha 86	</g:form>
Novo texto	</g:uploadForm>

Pelas tabelas anteriores, pode-se notar que as atualizações realizadas estão classificadas basicamente em duas categorias: (1) campos textos (*textField*) que foram substituídos por arquivos de entrada (*input type=“file”*) e (2) o formulário *<g:form>* foi substituído por *<g:uploadForm>*, um formulário que aceita *upload* de arquivos.

- Agora que os campos textos (*textField*) foram substituídos, é necessário que o controlador *DescricaoController* trate adequadamente o *upload* de arquivos. Substitua a ação *save* como descrita a seguir. Pode-se notar pela implementação dessa ação que os arquivos são obtidos (operação *request.getFile*), armazenados em um diretório (operação *transferTo*) e, por fim, os nomes dos arquivos são armazenados nos atributos *audio*, *video* e *imagem* (exemplo: descricaoInstance.imagem = imageFile.originalFilename).

```
def save = {
   def descricaoInstance = new Descricao(params)
   def webRootDir = servletContext.getRealPath(“/”)
    def prodDir = new File(webRootDir, “/produto/${descricaoInstance.produto.
id}”)
    prodDir.mkdirs()

    def imageFile = request.getFile(‘imagem’)
    if(!imageFile.empty){
        imageFile.transferTo( new File( prodDir, imageFile.originalFilename))
        descricaoInstance.imagem = imageFile.originalFilename
    }
    def audioFile = request.getFile(‘audio’)
    if(!audioFile.empty){
        audioFile.transferTo( new File( prodDir, audioFile.originalFilename))
        descricaoInstance.audio = audioFile.originalFilename
    }
    def videoFile = request.getFile(‘video’)
    if(!videoFile.empty){
        videoFile.transferTo( new File( prodDir, videoFile.originalFilename))
        descricaoInstance.video = videoFile.originalFilename
    }
    if (descricaoInstance.save(flush: true)) {
        flash.message = “${message(code: ‘default.created.message’, args:
[message(code: ‘descricao.label’, default: ‘Descricao’), descricaoInstance.id])}”
         redirect(action: “show”, id: descricaoInstance.id)
    }
    else {
      render(view: “create”, model: [descricaoInstance: descricaoInstance])
    }
}
```

- Na visão show (*grails-app/views/descricao/show.gsp*) do controlador DescricaoController, realize as seguintes alterações:

Tabela 7 Alteração na visão show.

Linha 42	<td valign=”top” class=”value”>${fieldValue(bean: descricaoInstance, field: “imagem”)}</td>
Novo texto	<td valign=”top” class=”value”> <g:if test=”${descricaoInstance.imagem}”> </g:if> </td>
Linha 49	<td valign=”top” class=”value”>${fieldValue(bean: descricaoInstance, field: “audio”)}</td>
Novo texto	<td valign=”top” class=”value”> <g:if test=”${descricaoInstance.audio}”> <href=”${createLinkTo(dir:’produto/’+descricaoInstance.produto.id, file:”+descricaoInstance.imagem)}”/>${descricaoInstance.audio} </g:if> </td>
Linha 56	<td valign=”top” class=”value”>${fieldValue(bean: descricaoInstance, field: “video”)}</td>
Novo texto	<td valign=”top” class=”value”> <g:if test=”${descricaoInstance.video}”> <href=”${createLinkTo(dir:’produto/’+descricaoInstance.produto.id, file:”+descricaoInstance.imagem)}”/>${descricaoInstance.video} </g:if> </td>
Linha 63	<g:link controller="produto" action="show" id="${descricaoInstance?.produto?.id}">
Novo texto	<g:link controller=”${descricaoInstance?.produto?.getClass().getSimpleName().toLowerCase()}” action=”show” id=”${descricaoInstance?.produto?.id}”>
Linha 86	</g:form>
Novo texto	</g:uploadForm>

Pela tabela anterior, pode-se notar que as atualizações realizadas estão classificadas basicamente em duas categorias: (1) a apresentação de textos foi

substituída pela apresentação da imagem (<*img src*>) ou por um *link* para os arquivos de áudio ou vídeo (<*a href*>) e (2) na linha 63, o controlador é determinado em tempo de execução pela classe da instância de produto – método *getClass().getSimpleName()*. Por exemplo, se o produto é um Cd, invoca o controlador CdController.

- Como última alteração, acrescente o método *toString()* nas classes *Produto* e *Descricao*. Esse método retorna uma representação (por exemplo, o que é apresentada nas visões – páginas HTML) das instâncias das classes. Para a classe *Descricao*, esse método retorna o conteúdo do atributo *tecnica* (descrição técnica). Para a classe Produto, retorna o tipo de produto (CD, DVD, livro) concatenado ao nome do produto.

Classe Descricao	**Classe Produto**

String toString() {
return tecnica
}
	String toString () {
	return "[" + this.getClass().getSimpleName() +"] " + nome
	}

Agora que essas atualizações foram realizadas, a aplicação pode ser executada. Para testar a aplicação, acesse a lista de Cds e depois clique para acessar suas propriedades (Figura 49(a)). Clique em "Add Descricao", para adicionar uma descrição a esse produto (no caso, é um Cd, Figura 49(b). Para criar uma descrição, insira uma descrição técnica e três arquivos (imagem, áudio e vídeo), como na Figura 49(c). Note que os três produtos são apresentados por "tipo concatenado ao nome" – método *toString()*. Por fim, a descrição é apresentada. Pode-se notar pela Figura 49(d) que a imagem armazenada é apresentada. Além disso, os arquivos de áudio e vídeo podem ser acessados por meio de *links*.

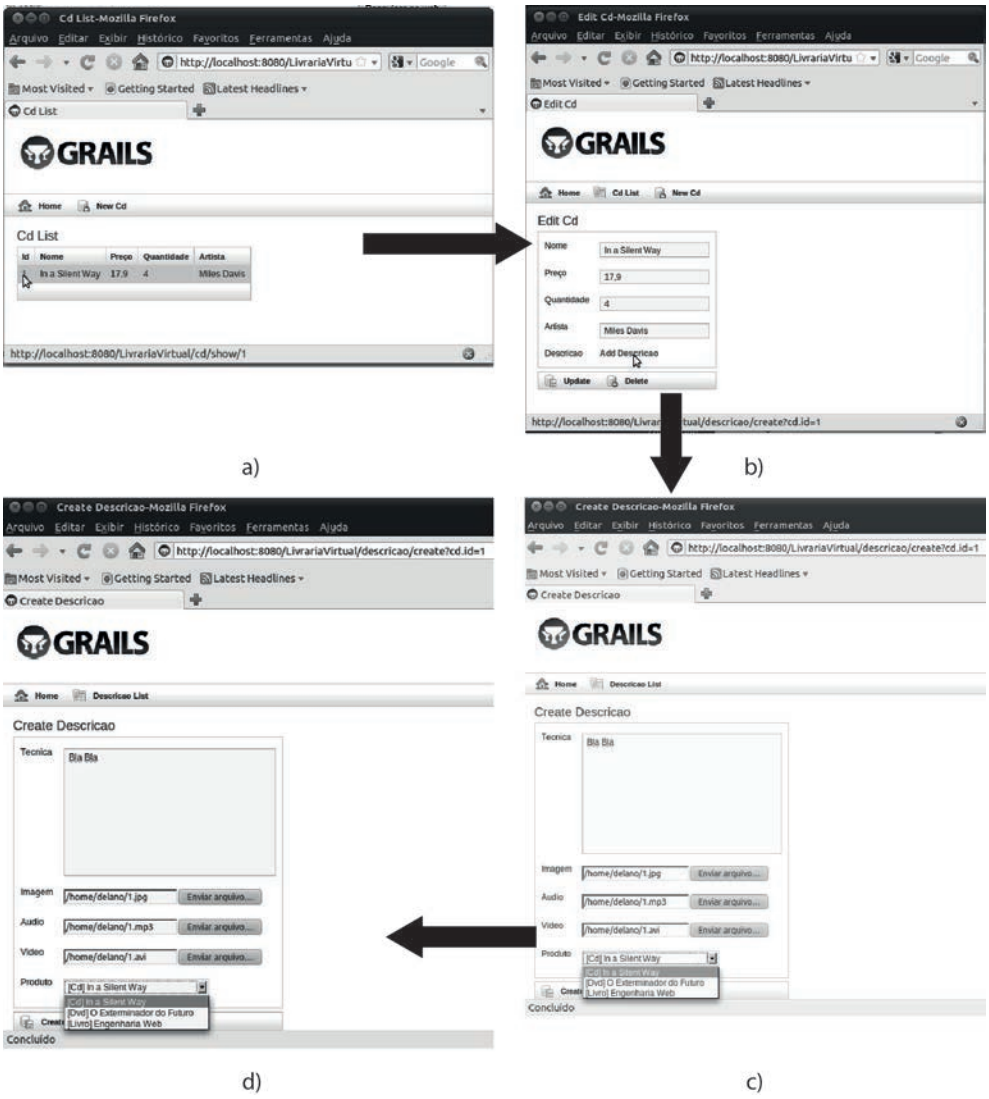


Figura 49 Inserindo Descrição em Produtos.

8.4.4 CRUD das demais classes

Dando continuidade à implementação parcial da aplicação **LivrariaVirtual.com.br**, esta seção apresenta a implementação do CRUD das demais classes de domínio da aplicação: *Usuario*, *Compra* e *Item*. Pela observação da Figura 50, é possível notar que existem dois relacionamentos um para muitos: (1) *Usuario* e *Compra* e (2) *Compra* e *Item*.

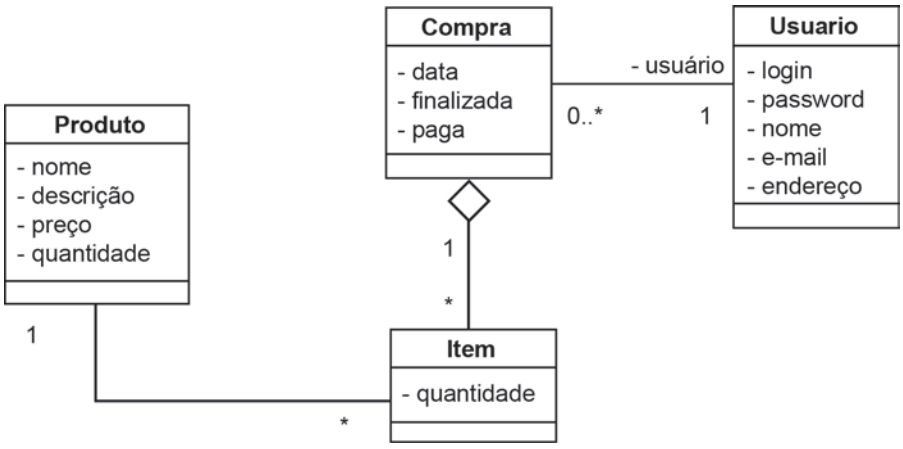


Figura 50 Diagramas de classes: Usuario, Compra e Item.

Siga os passos descritos a seguir na implementação das funcionalidades relacionadas ao CRUD das classes *Compra*, *Item* e *Usuario*.

- Crie, usando os passos da Seção 8.4.1, a classe de domínio *Compra* do pacote lv. Abra a classe *Compra* (Figura 51) e insira os atributos dessa classe. Por fim, acrescente as validações de dados nos atributos. Pela Figura 51, pode-se notar que: (1) uma *Compra* está relacionada a um *Usuario* (um usuário pode fazer várias compras). Logo, uma compra pertence a um usuário (*belongsTo*) e (2) uma *Compra* está relacionada com *Itens* – uma compra contém vários itens (*hasMany*).
- Crie, usando os passos da Seção 8.4.1, a classe de domínio *Item* do pacote lv. Abra a classe *Item* (Figura 52) e insira os atributos dessa classe. Por fim, acrescente as validações de dados nos atributos. Como discutido, uma compra pode possuir vários itens – um item pertence a uma compra (*belongsTo*).
- Crie, usando os passos da Seção 8.4.1, a classe de domínio *Usuario* do pacote lv. Abra a classe *Usuario* (Figura 53) e insira os atributos dessa classe. Por fim, acrescente as validações de dados nos atributos. Como discutido, um usuário pode realizar várias compras (*hasMany*).

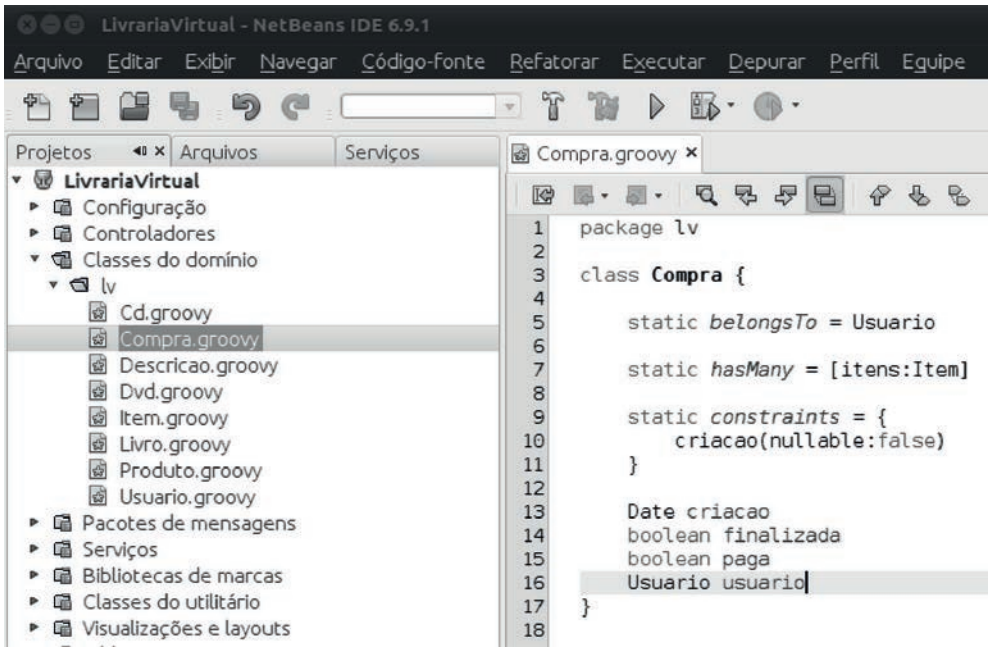


Figura 51 Criação da classe de domínio Compra.

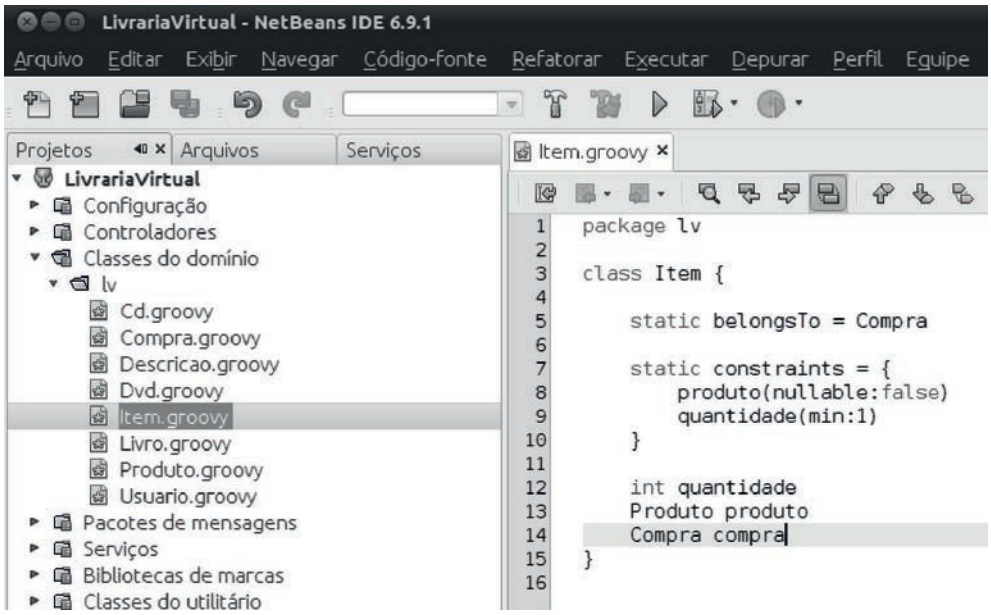


Figura 52 Criação da classe de domínio Item.

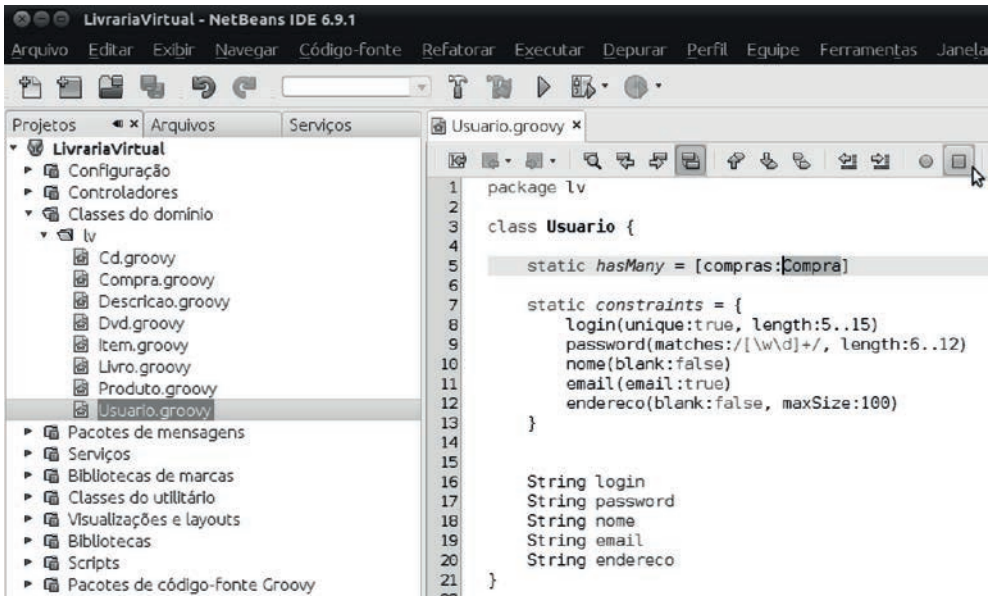


Figura 53 Criação da classe de domínio Usuario.

- Seguindo os passos da Seção 8.4.1.2, gere o controlador e as visões relacionadas às classes *Compra* e *Item*. O controlador associado à classe **Usuario** será discutido mais adiante.
- Como última alteração, acrescente o método *toString()* nas classes *Compra* e *Item*. Esse método retorna uma representação (por exemplo, o que é apresentada nas visões – páginas HTML) das instâncias das classes. Para a classe *Compra*, retorna o nome do usuário – *usuario.getNome()* – concatenado ao id da compra. Para a classe *Item*, esse método retorna a quantidade de produtos concatenada ao nome do produto – *produto.getNome()*.

Classe Compra	**Classe Item**

String toString() {
return usuario.getNome()
+ “[“ + this.id + “]”
}
	String toString () {
	return quantidade “ X “ + produto.getNome()
	}

Agora que essas atualizações foram realizadas, a aplicação pode ser executada.

- Para testar a aplicação, exercite as operações de CRUD das classes *Compra* e *Item*.

- Porém, antes de testar a aplicação, inclua um usuário na classe BootStrap.groovy.

```
new Usuario(login:"admin", password:"123456", nome:"Administrador",
    email:"fulano@gmail.com", endereco:"Rua X, 1000 São Paulo, SP").save()
```

8.4.5 Autenticação de usuários

A autenticação de usuários é uma funcionalidade muito importante em aplicações Web. Nesta seção é apresentado o uso de interceptação de chamadas (programação orientada a aspectos) na implementação da autenticação de usuários no contexto da aplicação **LivrariaVirtual.com.br**. Além disso, esta seção demonstra que existem situações em que os desenvolvedores necessitam criar os controladores e as visões manualmente (ao invés de utilizar o *scaffolding* – "Gerar Tudo", descrito na Seção 8.4.1.2).

- Nessa implementação será necessária a criação de três controladores – UserController, SecureController e ProdutoController – e duas visões, *login* e *register*.

8.4.5.1 *UsuarioController* e *SecureController*

Esta seção descreve os passos relacionados à criação de dois novos controladores – *UsuarioController* e *SecureController*.

UsuarioController. Para criar um novo controlador (*UsuarioController*), clique no botão direito do *mouse* no nó "Controladores" e escolha "Novo > Controlador do Grails" – Figura 54. Digite *UsuarioController* como o *Nome do artefato*, e lv como o *Pacote* e clique em Finalizar. O controlador *UsuarioController* é criado no nó Controladores.

SecureController. Repita os passos para criar o controlador *SecureController* no pacote lv.

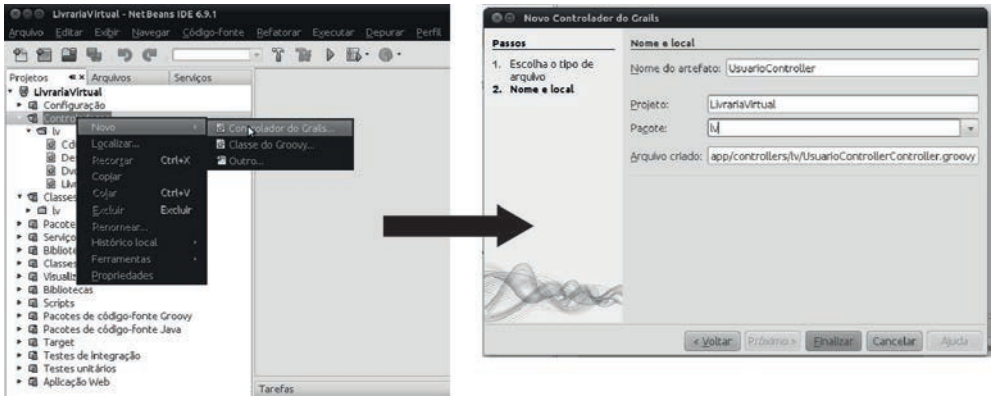


Figura 54 Novo Controlador.

Quanto à implementação do controlador *SecureController*, esta se encontra a seguir:

```
package lv

abstract class SecureController {

    def beforeInterceptor = [ action:
        this.&auth,
        except: [ 'handleLogin', 'login','handleRegistration',
'register' ]
    ]

    private auth() {
        if(!session.usuario) {
            redirect(controller:'usuario',action: 'login')
            return false
        }
```

Nesse controlador, o interceptador *beforeInterceptor* implementa uma interceptação das chamadas de ações. O comportamento dele é o seguinte: intercepte todas as chamadas a ações – com exceção das ações *handleLogin*, *login*, *handleRegistration* e *register* – e chame o método *auth*, que verifica se existe um usuário logado (*session.usuario*): (1) caso esteja logado, nenhuma ação é realizada; (2) caso não esteja logado, esse método redireciona para a ação *login* do controlador *UsuarioControlador*.

Para que a autenticação funcione, é necessário atualizar todos os controladores de tal forma que todos sejam subclasses de *SecureController* (analógo

ao *UsuarioController* descrito adiante). Isso garante que não importa qual seja o controlador (todos são subclasses de *SecureController*), o usuário precisa estar logado para acessar qualquer ação ou visão do sistema.

Quanto à implementação do controlador *UsuarioController*, esta se encontra a seguir:

```
package lv

class UsuarioController extends SecureController {
    def login = {
        if(session.usuario) {
            redirect(controller:'produto', action:'list')
        }
    }

    def handleLogin = {
        def usuario = Usuario.findByLogin( params.login )
        if(usuario) {
            if(usuario.password == params.password) {
                session.usuario = usuario
                redirect(controller:'produto', action:'list')
            }
            else {
                flash.message = "Incorrect password for ${params.
login}"
                redirect(action:login)
            }
        }
        else {
            flash.message = "User not found for login ${params.login}"
            redirect(action:login)
        }
    }
}
def logout = {
        session.usuario = null
        redirect(action:login)
    }
}
}
```

8.4.5.2 Visões da classe de domínio *Usuario*

O controlador *UsuarioController* é responsável pela autenticação do usuário e pela inscrição (registro) no sistema. Dessa forma, serão necessárias duas visões que possuem formulários HTML para a entrada desses dados. Crie duas visões associadas ao controlador *UsuarioController*: *login.gsp* e *register.gsp*.

- Para criar uma nova visão, clique no botão direito do *mouse* no nó Visualizações e Layouts e escolha "Novo > Arquivo GSP" – Figura 53. Digite "login" como o *Nome do arquivo*, e "grails-app/views/usuario" como a *Pasta* e clique em Finalizar. A visão login é criada no nó Visualizações e Layouts.

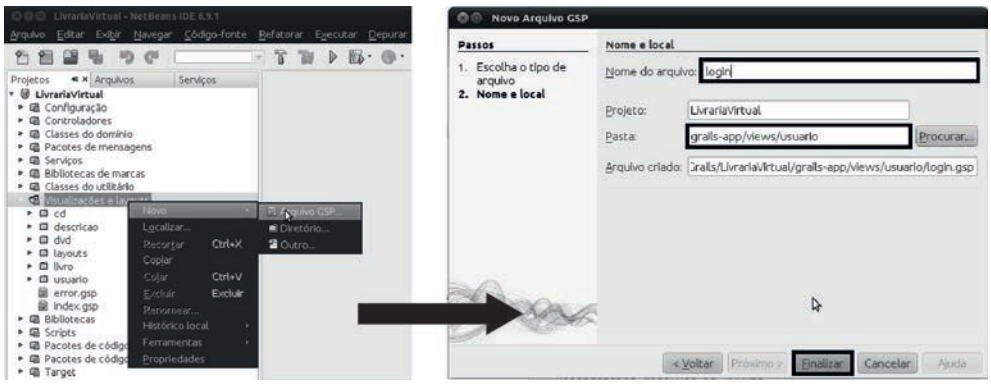


Figura 55 Criação de uma nova visão.

A implementação do arquivo *login.gsp* encontra-se a seguir. Pode-se observar que essa visão contém um formulário que invocará a ação *handleLogin* do controlador *UsuarioController*. A associação entre a visão *login.gsp* e o controlador *UsuarioConttroller* ocorre devido à convenção (paradigma COC) que todas as visões presentes no diretório grails/views/usuario estão associadas ao controlador de usuário (*UsuarioController*).

```
<html>
  <head>
    <title><g:message code="default.login.page"/></title>
  </head>
  <body>
      <h1><g:message code="default.login.message"/><g:link
action="register"> <g:message code="default.login.register"/></
g:link></h1>
      <g:form action="handleLogin">
        <table>
          <tbody>
            <tr class="prop">
              <td valign="top" class="name">
                <label for="login"><g:message code="default.login.
label"/></label>
              </td>
              <td> <input type="text" name="login" /> </td>
            </tr>
            <tr class="prop">
              <td valign="top" class="name">
                <label for="password"><g:message code="default.
password.label"/></label>
              </td>
              <td> <input type="password" name="password" /> </td>
            </tr>
          </tbody>
        </table>
        <input type="submit"/>
      </g:form>
  </body>
</html>
```

- Repita os passos anteriores para criar a visão *register* na pasta *grails-app/views/usuario*.

A implementação do arquivo *register.gsp* encontra-se a seguir. É importante notar que essa visão contém um formulário que invocará a ação *handleRegistration* do controlador *UsuarioController*.

```
<meta name="layout" content="main" />
  </head>
  <body>
    <div class="body">
      <div class="message">${flash.message}</div>
      <h1><g:message code="default.register.message"/></h1>
      <form action="handleRegistration">
        <table>
          <tbody>
            <tr class="prop">
              <td valign="top" class="name">
                <label for="login"><g:message code="default.register.
login"/></label>
              </td>
              <td>
                <input type="text" name="login" />
              </td>
            </tr>
            <tr class="prop">
              <td valign="top" class="name">
                <label for="password"><g:message code="default.
register.password"/></label>
              </td>
              <td>
                <input type="password" name="password" />
              </td>
            </tr>
            <tr class="prop">
              <td valign="top" class="name">
                <label for="confirm"><g:message code="default.
register.confirm"/></label>
              </td>
              <td>
                <input type="password" name="confirm" />
```

```
    </td>
            </tr>
            <tr class="prop">
              <td valign="top" class="name">
<label for="name"><g:message code="default.register.name"/></label>
              </td>
              <td>
                <input type="text" name="nome" />
              </td>
            </tr>
            <tr class="prop">
              <td valign="top" class="email">
                <label for="email"><g:message code="default.register.
email"/></label>
              </td>
              <td>
                <input type="text" name="email" />
              </td>
            </tr>
            <tr class="prop">
              <td valign="top" class="email">
                <label for="endereco"><g:message code="default.
register.address"/></label>
              </td>
              <td>
                <input type="text" name="endereco" />
              </td>
            </tr>
          <input type="submit"/>
          </tbody>
        </table>
      </form>
    </div>
  </body>
</html>
```

8.4.5.3 *ProdutoController*

Pode-se observar pelo código-fonte do controlador *UsuarioController* que as ações *handleLogin* e *handleRegister* redirecionam para a ação *list* do controlador *ProdutoController*. Dessa forma, é necessária a criação desse controlador. Repita os passos da Seção 8.4.1.2 para criar o controlador *ProdutoController* no pacote lv. A implementação do controlador *ProdutoController* encontra-se a seguir:

```
class ProdutoController extends SecureController{

    def index = {
        redirect(action: “list”, params: params)
    }

    def list = {
        params.max = Math.min(params.max ? params.int(‘max’) : 10,
100)
        [produtoInstanceList: Produto.list(params),
         produtoInstanceTotal: Produto.count()]
    }
}
```

- O controlador *ProdutoController* define uma ação *list* que recupera alguns dados do modelo. Dessa forma, é necessária a definição da visão *list.gsp* responsável pela apresentação desses dados. É importante ressaltar que devido às características do Grails, que permite consultas polimórficas, por exemplo, Produto.findAll() que retorna instâncias da classe Produto e das subclasses, a visão *list* apresenta uma lista de produtos não importando que subclasse ele pertença (Cd, Dvd, Livro).
- Repita os passos da Seção 8.4.5.2 para criar a visão *list* na pasta *grails-app/views/product*. O código-fonte encontra-se disponível no CD suplementar que acompanha o livro.

8.4.5.4 Internacionalização

Grails apoia a internacionalização (i18n). Ou seja, com o Grails é possível personalizar o texto que aparece em qualquer visão (*.gsp) com base no *Locale* dos usuários.

Um objeto *Locale* representa uma região geográfica específica, política ou cultural. Uma operação que requer uma localidade para executar a sua tarefa é denominada de sensível, usa o objeto *Locale* para prover a informação mais apropriada aos usuários. Por exemplo, exibir o preço de uma mercadoria é uma operação sensível à localidade – o número deve ser formatado de acordo com os costumes e convenções do país de origem do usuário, região ou cultura.

O objeto *Locale* é composto de: (1) um código do idioma ou (2) um código do idioma e um código do país. Por exemplo, *en* é o código para o idioma Inglês (não importando o país ou região geográfica), enquanto *pt_BR* e *pt_PT* são dois *Locales* que compartilham o mesmo idioma: o primeiro é o código para o Português do Brasil e o segundo é o código para o Português usado em Portugal.

Para tirar proveito do suporte a internacionalização em Grails, a equipe de desenvolvimento tem que criar *message bundles* – uma para cada idioma que a equipe deseja internacionalizar. *Message bundles* em Grails estão localizadas dentro do Diretório *grails-app/i18n* e são simples arquivos de propriedades Java. Por exemplo:

messages.properties → arquivo padrão.
messages_pt_BR → arquivo com a tradução para o Português do Brasil.
messages_en → arquivo com a tradução para o Inglês, não importando o país.

Por padrão, Grails procura em *messages.properties* para mensagens, a menos que o usuário tenha definido um *Locale* em específico. A equipe de desenvolvimento pode criar novas *message bundles* simplesmente criando novos arquivos de propriedades que terminam com a localidade que está interessada. Por exemplo, *messages_pt_BR.properties* para o Português do Brasil.

As visões são os locais mais comuns para o uso de mensagens internacionalizadas. Para acessar as mensagens personalizadas nas visões, a equipe de desenvolvimento basta utilizar a seguinte tag nas visões (*.gsp) desenvolvidas:
<g:message code="welcome.greeting" />

Se a equipe tiver incluído uma chave no arquivo *messages.properties* (com sufixo do *Locale* apropriado), tal como ilustrada a seguir, então Grails apresenta a mensagem personalizada:
welcome.greeting = Good Morning. My name is Bob.

Note que em algumas ocasiões é necessário passar argumentos para a mensagem. Isso também é possível com a mesma tag:
<g:message code="welcome.greeting" args="${ ['Night','Bill'] }" />

No entanto, é necessário utilizar os parâmetros de posicionamento na mensagem.
welcome.greeting = Good {0}. My name is {1}.

Por padrão, os templates usados no *scaffolding* de controladores e visões não levam em consideração a internacionalização (I18n). No entanto, é possível instalar o plugin i18n-templates, que é idêntico aos templates padrões, exceto o fato de que eles levam em consideração a internacionalização de componentes, tais como labels e botões. A instalação de plugins para o Grails será discutida a seguir nesta unidade.

Pela observação do código-fonte das visões *login* e *register* (assim como dos demais arquivos *.gsp gerados automaticamente), pode-se observar que essas visões incluem a tag <*g:message code*>, em que code é uma referência a algum termo a ser internacionalizado. O nó "Pacotes de mensagens" contém um conjunto de arquivos de propriedades (termos a serem traduzidos para diferentes línguas) – Figura 56. É importante salientar que esses arquivos são gerados automaticamente durante a execução dos comandos do Grails (create-app, generate-all etc.).

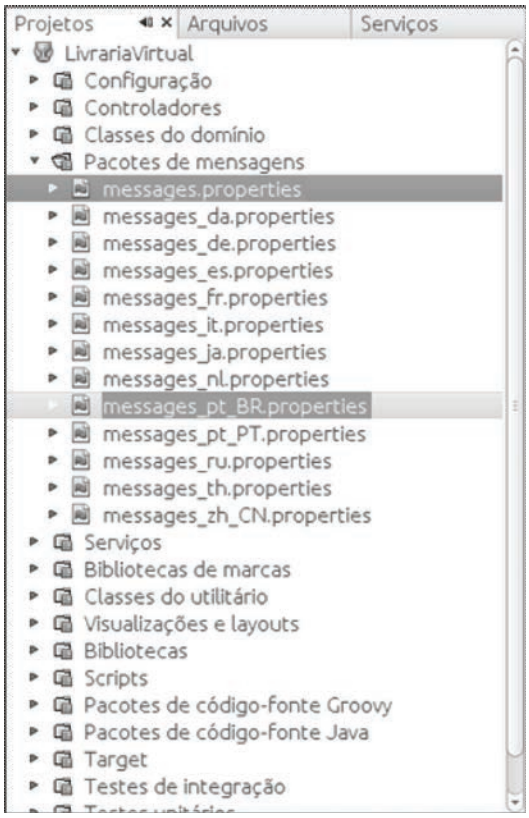


Figura 56 I18n (arquivos de propriedades).

Portanto, na implementação das visões *login* e *register* foi necessária a inserção de alguns termos no arquivo *messages.properties* (arquivo padrão da internacionalização). Os termos presentes no arquivo padrão são utilizados caso não seja encontrada a tradução desse termo em um arquivo de termos traduzidos (por exemplo, messages_fr que deveria conter traduções para o francês).

```
default.login.page=Login Page
default.login.message=Welcome to LivrariaVirtual. Login below or
default.login.register=register here
default.login.label=Login
default.password.label=Password
default.register.page=Register Page
default.register.message=Enter your details below to register for
LivrariaVirtual.
default.register.login=Login
default.register.password=Password
default.register.confirm=Confirm Password
default.register.name=Name
default.register.email=Email
default.register.address=Address
```

Com propósitos de simplicidade, os termos foram traduzidos apenas para a língua portuguesa (versão brasileira) e armazenados no arquivo *messages_pt_BR.properties*.

```
default.login.page=Página de Acesso
default.login.message=Bem-vindo ao sistema LivrariaVirtual. Entre com
usuário e senha ou
default.login.register=registre-se aqui
default.login.label=Usuário
default.password.label=Senha
default.register.page=Página de Registro
default.register.message=Entre com as informações abaixo para se
registrar no sistema LivrariaVirtual.
default.register.login=Usuário
default.register.password=Senha
default.register.confirm=Confirme Senha
default.register.name=Nome
default.register.email=Email
default.register.address=Endereço
```

Note que o esforço da equipe Web para internacionalizar sua aplicação consiste em determinar os termos a serem traduzidos, que serão inseridos em um arquivo de propriedades, e depois realizar a tradução para as diferentes línguas. As traduções serão armazenadas em arquivos cujos nomes terminam com o *Locale* (língua e região) desejado.

A Figura 57 apresenta a internacionalização da página de *login* da aplicação **LivrariaVirtual.com.br**. Vale a pena salientar que apenas foi necessário redefinir a linguagem padrão do navegador (de inglês para português e vice-versa) e atualizar a página (*reload*) para que os termos fossem apresentados na linguagem escolhida.

a) b)

Figura 57 Internacionalização da **LivrariaVirtual.com.br**.

O leitor interessado pode executar a aplicação e verificar que a página de inscrição (*register.gsp*) também foi internacionalizada. Quanto à internacionalização das demais visões da aplicação, ainda é necessário o esforço de tradução dos termos para a língua desejada. Fica como exercício para o leitor a realização dessa tarefa.

8.4.5.5 Personalização

Finalizando a implementação parcial da aplicação **LivrariaVirtual.com.br**, é possível realizar algumas personalizações que atendam as necessidades da equipe de Engenharia Web. Como exemplos, podemos citar:

- Alterar o arquivo URLMapping.groovy (diretório conf) para definir o controlador/ação padrão. Usando a configuração apresentada no arquivo listado a seguir, a URL – <http:/localhost:8080/LivrariaVirtual> – direciona para a ação *login* do controlador *UsuarioController*. Ou seja, a página de *login* do sistema.

```
class UrlMappings {


    static mappings = {
      “/$controller/$action?/$id?”{
            constraints {
                // apply constraints here
            }
        }
  “500”(view:’/error’)
        “/” {
            controller = “usuario”
            action = “login”
        }
    }
}
```

- Alterar o leiaute padrão da aplicação. Todas as visões geradas baseiam-se no leiaute definido no arquivo *main.gsp* (diretório views/layouts).

```
<!DOCTYPE html>
<html>
    <head>
        <title><g:layoutTitle default="Grails" /></title>
        <link rel="stylesheet" href="${resource(dir:'css',file:'main.
css')}" />
        <link rel="shortcut icon"
              href="${resource(dir:'images',file:'favicon.ico')}"
type="image/x-icon" />
        <g:layoutHead />
        <g:javascript library="application" />
    </head>
    <body>
        <div id="spinner" class="spinner" style="display:none;">
             <img src="${resource(dir:'images',file:'spinner.gif')}"
                  alt="${message(code:'spinner.
alt',default:'Loading...')}" />
        </div>
        <div id="grailsLogo">
         <img src="${resource(dir:'images',file:'lv.png')}"
              alt="Grails" border="0" />
        </div>
        <g:layoutBody />
    </body>
</html>
```

Ao substituir (a substituição está em negrito) a imagem (no caso, o logo do Grails), o leiaute de todas as páginas da aplicação torna-se diferente – como ilustra a Figura 58. Além disso, essa figura apresenta a visão *list* associada ao controlador *ProdutoController*, que foi discutida anteriormente. Essa visão *list* apresenta todos os produtos cadastrados, não importa a qual subclasse ele pertença. E, por fim, essa visão permite o acesso às funcionalidades relacionadas a compras e à operação de *logout*.

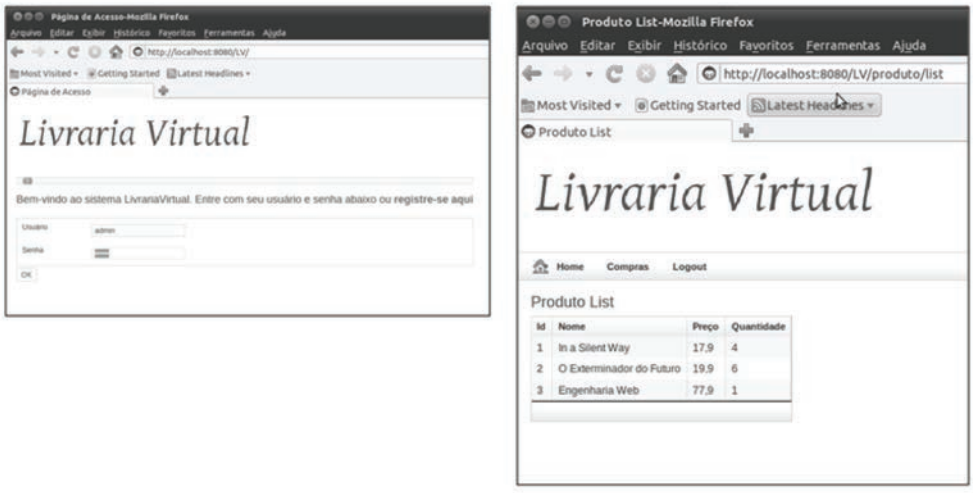


Figura 58 Novo leiaute para a aplicação **LivrariaVirtual.com.br**.

O código-fonte dessa aplicação encontra-se no CD suplementar que acompanha o livro. No entanto, fica a sugestão que os leitores sigam os passos descritos nesta unidade de tal forma que obtenham a implementação do sistema.

8.5 Funcionalidades AJAX – Agenda de Contatos

Esta seção apresenta uma aplicação web simples – Agenda de Contatos[34] – que incorpora algumas funcionalidades AJAX em sua implementação. Ela consiste em uma agenda simples que armazena informações de contato tais como endereço, telefone, e-mail etc.

A instalação padrão Grails já incorpora algum suporte básico (biblioteca Javascript prototype)[35] para a implementação de funcionalidades AJAX. No entanto, é possível o uso de outros *frameworks*, tais como Dojo[36] e Google Web Toolkit,[37] por meio da instalação de plugins Grails. No contexto da aplicação Agenda de Contatos, discutida nesta seção, as funcionalidades AJAX serão implementadas utilizando o suporte básico provido pelo *framework* Grails (biblioteca Javascript prototype), bem como os componentes AJAX fornecidos pelo plugin Grails RichUI.[38] Os passos necessários para a implementação da aplicação Agenda de Contatos são descritos a seguir.

34 O código-fonte dessa aplicação encontra-se no CD suplementar que acompanha o livro.
35 Disponível em: <http://www.prototypejs.org/>. Acesso em: 23 jan. 2012.
36 Disponível em: <http://www.dojotoolkit.org/>. Acesso em: 23 jan. 2012.
37 Disponível em: <http://code.google.com/webtoolkit/>. Acesso em: 23 jan. 2012.
38 Disponível em: <http://www.grails.org/plugin/richui>. Acesso em: 23 jan. 2012.

AJAX – acrônimo de *Asynchronous Javascript And XML* – é o uso metodológico de tecnologias como Javascript e XML, providas por navegadores, para tornar páginas Web mais interativas com o usuário, utilizando-se de solicitações assíncronas de informações.
O modelo clássico de aplicação web trabalha assim: a maioria das ações do usuário na interface dispara uma solicitação HTTP para o servidor Web (modelo cliente-servidor). O servidor processa algo, recuperando dados, realizando cálculos, conversando com vários sistemas legados, e então retorna uma página HTML para o cliente. Com a popularização de sistemas Web e o aumento da velocidade das conexões banda larga, o problema da espera pelo envio e retorno de toda a página HTML se tornou muito mais evidente para o usuário. As principais vantagens das aplicações que utilizam AJAX para determinadas requisições é que os dados trafegados pela rede são reduzidos e o usuário não precisa aguardar a página ser recarregada a cada interação com o servidor.

8.5.1 Criação do projeto e instalação de plugins

- Crie, utilizando os mesmos passos da criação do projeto **LivrariaVirtual.com.br** (Seção 8.3.2), o projeto **Agenda**.
- No desenvolvimento da aplicação **Agenda** serão utilizados os plugins Grails *richui* e *i18n-templates*. Dessa forma, é necessária a instalação desses plugins. Para instalar o plugin *richui*, clique com o botão direito do *mouse* no nome do projeto (**Agenda**) e escolha *Plug-ins do Grails...* – Figura 59(a). Na aba *Novos plugins*, selecione o nome do plugin desejado (*richui*) e clique em Instalar – Figura 59(b). Siga os mesmos passos na instalação do plugin *i18n-templates*.

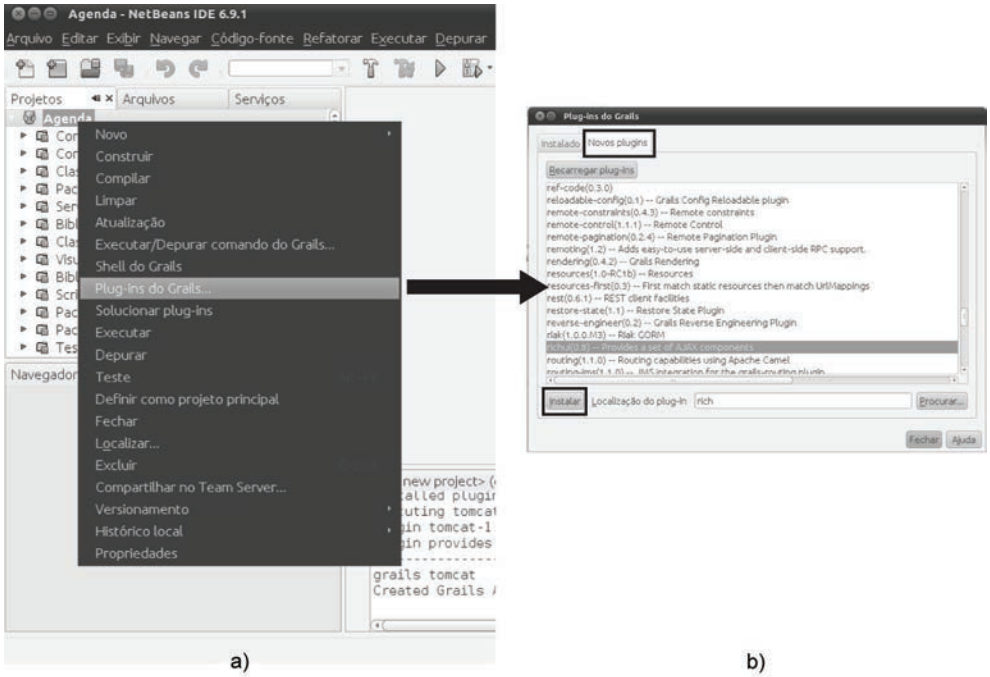


Figura 59 Instalação de plugins Grails.

8.5.2 Configuração do banco de dados

Como discutido, a instalação do Grails já incorpora uma cópia do HSQLDB. HSQLDB é ótimo para aplicações de demonstração, mas em algum momento os desenvolvedores precisarão de um banco de dados mais robusto, tal como MySQL, Postgresql e Oracle. Com propósitos de ilustração, o banco de dados MySQL será utilizado na implementação da aplicação **Agenda**. No entanto, o leitor deve sentir-se à vontade para usar outro banco de dados se este já estiver instalado e configurado. Uma cópia gratuita do MySQL Community Edition pode ser obtida em <http://dev.mysql.com/downloads>.

Os passos a seguir descrevem a configuração necessária à utilização do banco de dados MySQL no desenvolvimento da aplicação **Agenda**. Para o uso de outro banco de dados, tal como Oracle, os passos são análogos.

1. Crie uma base de dados e um usuário denominado contato e altere os privilégios de acesso a essa base de dados. Os comandos MySQL que realizam essas tarefas são apresentados a seguir:

```
CREATE USER ‘contato’@’localhost’ IDENTIFIED BY ‘contato’;
CREATE USER ‘contato’@’%’ IDENTIFIED BY ‘contato’;
CREATE DATABASE contato;
GRANT ALL PRIVILEGES ON contato.* TO ‘contato’@’localhost’ WITH GRANT OPTION;
GRANT ALL PRIVILEGES ON contato.* TO ‘contato’@’%’ WITH GRANT OPTION;
```

2. Copie o *driver* JDBC para o diretório *grails-app/lib directory* e ajuste o arquivo *grails-app/conf/DataSource.groovy* para configurar o acesso (url, username e password) ao banco de dados. Trechos desse arquivo são apresentados a seguir.

```
dataSource {
pooled = true
driverClassName = “com.mysql.jdbc.Driver”
username = “contato”
password = “contato”
url = “jdbc:mysql://localhost/contato”
}
hibernate {
cache.use_second_level_cache = true
cache.use_query_cache = true
cache.provider_class = ‘net.sf.ehcache.hibernate.EhCacheProvider’
}
environments {
development {
dataSource {
                      dbCreate = “create-drop” // one of ‘create’,
‘create-drop’,’update’
                      url = “jdbc:mysql://localhost/contato”
}
}
```

8.5.3 CRUD da classe de domínio contato

Agora que o projeto foi criado e configurado, o próximo passo é implementar as primeiras funcionalidades da aplicação: operações de CRUD da classe de domínio Contato. Esta representa os contatos gerenciados pela aplicação e está

relacionada (relacionamento um para muitos) com a classe de domínio *Cidade*, que representa as cidades brasileiras onde os contatos residem.

- Crie, usando os mesmos passos discutidos na Seção 8.3.3, a classe de domínio Cidade do pacote Contato. Abra a classe Cidade, insira os atributos e acrescente as validações nos mesmos. O código-fonte da classe Cidade é apresentado a seguir:

```
package contato

class Cidade {
    static constraints = {
        nome(blank:false, size:1..80)
        estado(blank: false, size:2..2)
    }

    String nome   // nome da cidade
    String estado // nome do estado

    String toString() {
        return nome + " - " + estado
    }
}
```

Figura 60 Classe de domínio Cidade.

Observa-se que a classe Cidade possui dois atributos: o atributo *nome* armazena o nome da cidade, enquanto o atributo *estado* armazena a sigla (dois caracteres) do estado onde a cidade está situada. Por exemplo, os objetos que representam as cidades de São Paulo e Belo Horizonte poderiam ser instanciados da seguinte forma:

def sp = new Cidade(nome:'São Paulo', estado: 'SP').

def bh = new Cidade(nome:'Belo Horizonte', estado: 'MG').

Crie, usando os mesmos passos discutidos na Seção 8.3.1, a classe de domínio Contato do pacote *contato*. Abra a classe Contato, insira os atributos e acrescente as validações nos mesmos.

```
package contato
class Contato {
    static constraints = {
        nome(blank:false)
        endereco(blank:true)
        complemento(blank:true)
        cidade(blank:true)
        fixo(matches:/^\(?(\d{2})?\)?\d{4}-\d{4}$/)
        celular(matches:/^\(?(\d{2})?\)?\d{4}-\d{4}$/)
        principal(email:true, blank:false)
        alternativo(email:true)
        homepage(url:true)
        dataNasc(blank:true)
    }

    String nome
    String endereco
    String complemento
    Cidade cidade
    String fixo
    String celular
    Date dataNasc
    String principal
    String alternativo
    String homepage

    static Set estados
    static Collection estados() {
        if (estados == null) {
            estados = new TreeSet()
            estados.add("")
            def cidades = Cidade.findAll()
            cidades.each { cidade ->
                estados.add(cidade.estado)

                }
        }
        return  estados
    }
}
```

Figura 61 Classe de domínio Contato.

A classe de domínio Contato possui os seguintes atributos:

- *nome* – que armazena o nome do contato;
- *endereço* – que armazena o endereço do contato;
- *complemento* – que armazena o complemento de endereço;
- *cidade* – que armazena a cidade (classe discutida anteriormente);
- *fixo* – que armazena o telefone fixo do contato;
- *celular* – que armazena o telefone celular do contato;
- *principal* – que armazena o endereço eletrônico do contato;
- *alternativo* – que armazena um endereço eletrônico alternativo;
- *homepage* – que armazena a página principal do contato;
- *dataNasc* – que armazena a data de aniversário do contato.

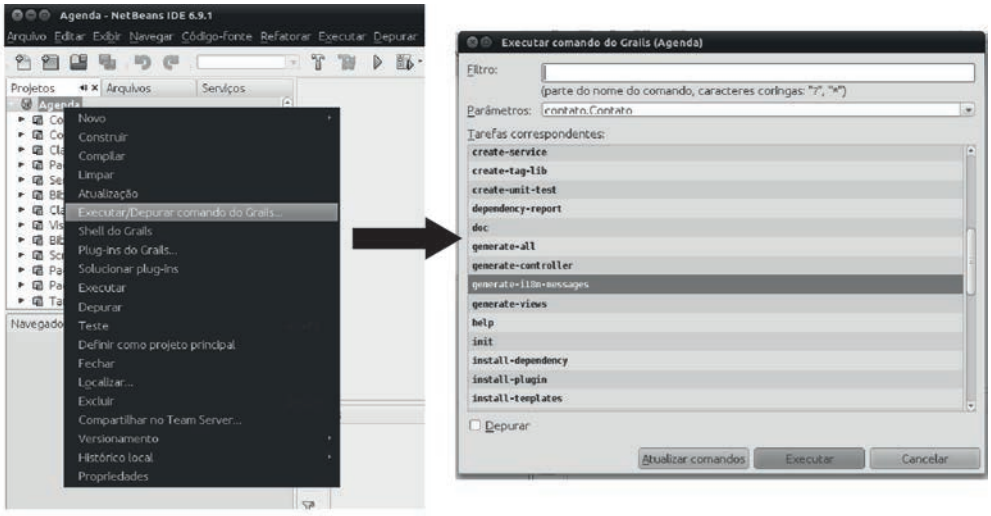
Observações: (1) é importante salientar que a implementação da classe Contato utiliza extensivamente as validações discutidas na Seção 8.4 e apresentadas na Tabela 4. Por exemplo, a validação *matches* e expressões regulares garantem que apenas telefones válidos sejam armazenados; (2) o atributo *estados* e o método *estados()* serão discutidos adiante nesta unidade. Por enquanto, o leitor deve desconsiderá-los. O código-fonte dessa classe é apresentado na Figura 61.

8.5.4 *Scaffolding* e internacionalização

Agora que as classes de domínio estão criadas, é preciso criar os controladores e as visões relacionados a essas classes de domínio. Utilizando os passos descritos anteriormente, realize o *scaffolding* da classe de domínio Contato. Para a classe de domínio Cidade, o *scaffolding* não é necessário.

Como discutido, na instalação padrão do Grails, os templates usados no *scaffolding* de controladores e visões não levam em consideração a internacionalização (I18n). No entanto, esse comportamento foi alterado pela instalação do plugin *i18n-templates*. Ou seja, as visões e o controlador gerados no passo anterior estão parcialmente internacionalizados. Para completar a tarefa de internacionalização é necessário executar o comando *grails generate-18n-messages*. Esse comando, baseando-se na descrição de uma classe de domínio, gera automaticamente as mensagens a serem internacionalizadas relacionadas a essa classe de domínio. Para executar um comando Grails, clique com o botão direito do *mouse* no nome do projeto (**Agenda**) e escolha *Executar/Depurar comando do*

Grails... – Figura 62(a). Selecione o nome do comando desejado (*generate-i18n-messages*), adicione na caixa de entrada *Parâmetros* o nome da classe de domínio (contato.Contato) e clique em *Executar* – Figura 62(b). Esse comando produz na janela *Saída* um conjunto de mensagens a serem internacionalizadas. Copie (possivelmente realizando a tradução) essas mensagens para o *message bundle* (presente no diretório *grails-app/i18n*) apropriado. A Figura 63 apresenta as mensagens relacionadas à classe de domínio Contato que foram traduzidas e copiadas para o arquivo messages_pt_BR.properties (*message bundle* para o Português do Brasil).



a) b)

Figura 62 Comando *grails generate-i18n-messages*.

```
# Contato messages
contato.create=Cria Contato
contato.edit=Edita Contato
contato.list=Lista de Contatos
contato.new=Novo Contato
contato.show=Apresenta Contato
contato.created=Contato {0} criado
contato.updated=Contato {0} atualizado
contato.deleted=Contato {0} removido
contato.not.found=Contato não encontrado com id {0}
contato.not.deleted=Contato não removido with id {0}
contato.id=Id
contato.nome=Nome
contato.nome.blank.error=Propiedade [Nome] da classe [Contato] não
pode ser vazio
contato.nome.nullable.error=Propriedade [Nome] da classe [Contato]
não pode ser nulo
contato.residencial=Residencial
contato.residencial.nullable.error=Propriedade [Residencial] da
classe [Contato] não pode ser nulo
contato.celular=Celular
contato.celular.nullable.error=Propriedade [Celular] da classe
[Contato] não pode ser nulo
contato.aniversario=Aniversário
contato.aniversario.nullable.error=Propriedade [Aniversario] da
classe [Contato] não pode ser nulo
contato.search.label=Busca
```

Figura 63 Mensagens I18n para a classe de Domínio Contato.

8.5.5 Visões

Após a finalização da internacionalização, com o objetivo de incorporar algumas funcionalidades AJAX na aplicação, os próximos passos consistem em realizar algumas alterações nas visões associadas ao controlador *ContatoController*. É importante relembrar que essas visões foram geradas automaticamente pelo *scaffolding*.

8.5.5.1 Visão create

Na visão create (arquivo *create.gsp*), que é responsável pelo cadastro de contatos, foram realizadas duas alterações. A primeira consiste na substituição da tag *<g:datePicker/>* pela tag *<richui:dateChooser/>* (componente AJAX do plugin *richui* instalado anteriormente). No contexto da aplicação **Agenda**, essa tag foi utilizada com o objetivo de permitir a utilização de um calendário dinâmico na entrada das datas de aniversário dos contatos (Figura 64). Trechos mais relevantes do código-fonte, relacionados a essa alteração, são apresentados a seguir.

Observações:

1. Pelo trecho de código pode-se observar que é necessário incluir a tag *<resource:dateChooser>*. Essa tag informa ao Grails que o componente AJAX *richui:dateChooser* será utilizado na visão.
2. Pelo trecho de código pode-se observar que o atributo *dataNasc* está vinculado ao componente AJAX *dateChooser*. Ou seja, a data escolhida será armazenada no atributo *dataNasc*.
3. Pela Figura 64 pode-se comprovar que a aplicação encontra-se internacionalizada para o *Locale* – Português do Brasil (*message bundle* discutido anteriormente nesta seção).

```
<head>
  …
<resource:dateChooser />
</head>
<body>
<richui:dateChooser name=”dataNasc” format=”dd.MM.yyyy” />
</body>
```

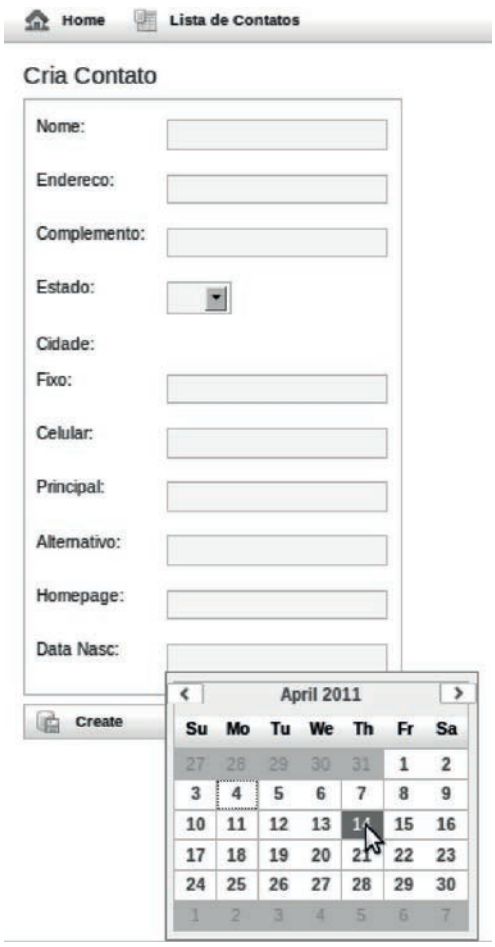


Figura 64 Formulário de cadastro de contatos.

A segunda alteração na visão create está relacionada à implementação da funcionalidade de *AJAX-Driven Selects*.[39] Ou seja, a funcionalidade relacionada ao seguinte comportamento: o conteúdo presente em uma caixa de seleção é dependente de outra caixa de seleção. Para implementar essa funcionalidade é necessário que uma caixa de seleção altere o conteúdo de outra caixa de seleção na mesma página web sem fazer nenhum *refresh* no browser. No contexto da aplicação **Agenda**, a seleção de cidades é dependente da seleção de um estado brasileiro. Ou seja, a caixa de seleção de cidades apenas apresenta cidades situadas no estado selecionado na outra caixa de seleção. Trechos mais relevantes do código-fonte da visão create, relacionados a essa alteração, são apresentados a seguir.

39 Disponível em: <http://www.grails.org/AJAX-Driven+SELECTs+in+GSP>. Acesso em: 23 jan. 2012.

```
<head>
<g:javascript library=”prototype” />
</head>
<body>
…
<g:select onchange=”${remoteFunction(controller:’contato’,action:’sel
ected’,update:’cidade’,
                                    params:’\’estado=\’ + this.value’
)}”
            from=”${Contato.estados()}”>
</g:select>
…
<div id=”cidade” name=”cidade”>
</div>
</body>
```

Observações:

1. Essa funcionalidade utiliza (tag *<g:javascript>*) os recursos AJAX providos pela biblioteca *prototype*.
2. É utilizada uma caixa de seleção (tag *<g:select>*) que ao ser selecionada (evento *onchange*) invoca assincronamente a ação *selected* do controlador *ContatoController*, passando alguns parâmetros. A lista de possíveis opções (atributo *from*) é o resultado da execução do método estados() da classe de domínio Contato (Figura 61). Esse método retorna o conjunto dos estados brasileiros.
3. O resultado da invocação assíncrona do controlador será refletido em , desde que o id desse trecho é “cidade”, que consiste no valor do atributo update da tag *<g:select>*. Pode-se observar na Figura 64 que desde que nenhum estado foi selecionado, o campo *cidade* encontra-se vazio.

Observa-se pela implementação de *selected* que esse método seleciona todas as cidades situadas no estado passado por parâmetro – Cidade. findAllByEstado(param) – e “renderiza” um g:select dinâmico que substituirá o trecho cujo id é igual a “cidade”. Ou seja, baseando-se no parâmetro estado, a ação *selected* seleciona as cidades e “produz” uma caixa de seleção dinâmica.

```
def selected = {
        def lst = new ArrayList();
        def param = params['estado']
        def cidades = Cidade.findAllByEstado(param)
        cidades.each { cidade -> lst.add(cidade) }
        render g.select(from:lst,name:'cidade',optionKey:'id')
}
```

A Figura 65 apresenta o formulário de cadastro de contatos. Inicialmente, nenhum estado se encontra selecionado. Porém, imediatamente após a seleção do Estado de São Paulo, a lista de cidades situadas nesse estado torna-se disponível na caixa de seleção Cidade.

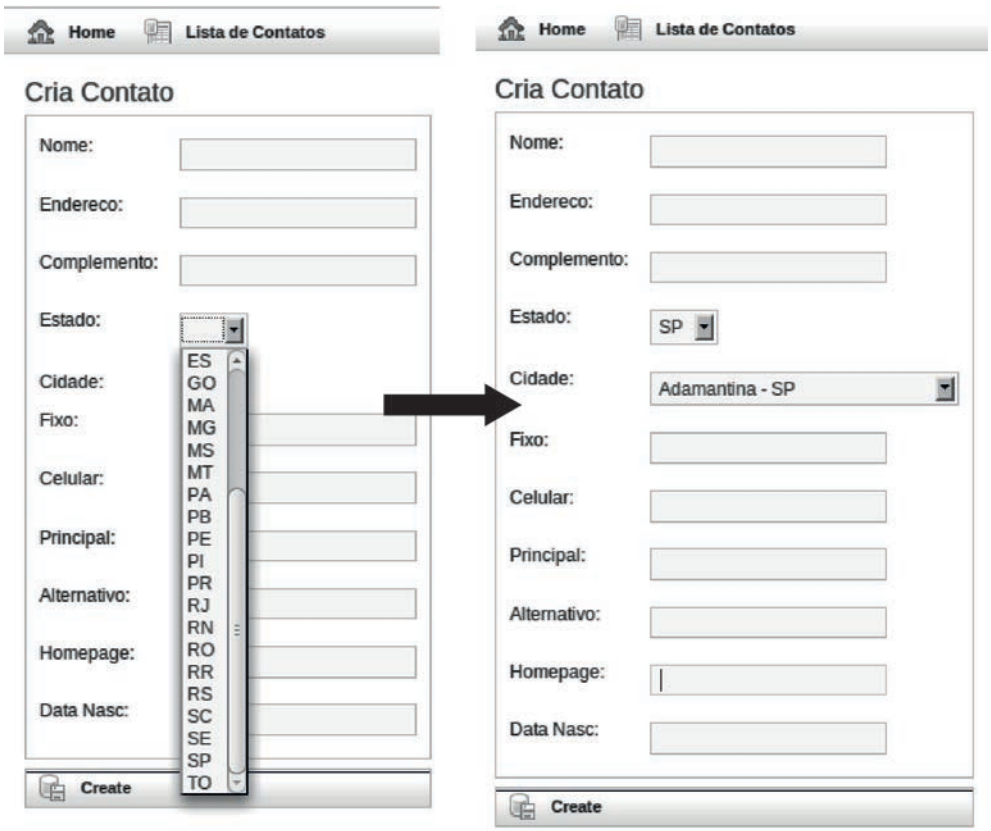


Figura 65 Funcionalidade AJAX – caixas de seleção dinâmicas.

8.5.5.2 Visão edit

Na visão edit (arquivo *edit.gsp*), que é responsável pela edição de contatos, as alterações são semelhantes às realizadas na visão create. Portanto, não serão discutidas.

8.5.5.3 Visão list

Na visão list (arquivo *list*.gsp), que é responsável pela apresentação da lista dos contatos armazenados, foram realizadas duas alterações.

A primeira alteração, objetivando diminuir a poluição visual, consiste em apresentar menos campos na tabela – apenas o id, o nome e a data de nascimento.

A segunda alteração está relacionada à implementação da funcionalidade de *autocomplete*. Essa funcionalidade ajuda muito os usuários finais no momento de selecionar uma opção. Além de ser visualmente mais agradável, esse recurso faz com que apenas apareçam as opções de acordo com o que é digitado no campo. Para a aplicação **Agenda**, a funcionalidade de *autocomplete* foi utilizada objetivando auxiliar a busca de contatos. Para tal será utilizado o componente AJAX *autocomplete* do plugin *richui*. Trechos mais relevantes do código-fonte da visão *list*, relacionados a essa alteração, são apresentados a seguir.

```
<head>
  <resource:autoComplete skin=”default” />
</head>
<body>
     <richui:autoComplete
        name=”search”
        action=”${createLinkTo(‘dir’: ‘contato/searchAJAX’)}”
        onItemSelect=”document.location.href = ‘${createLinkTo(dir:
‘contato/show’)}/’ + id;”
     />
</body>
```

Observações:

1. Pelo trecho de código, pode-se observar que é necessário incluir a tag *<resource:autoComplete>*. Essa tag informa ao Grails que o componente AJAX *richui:autoComplete* será utilizado na visão.
2. Pelo trecho de código, pode-se observar que o componente *richui:auto Complete* invoca assincronamente (sempre que ocorre uma edição no campo texto *search*) a ação *searchAJAX* do controlador *ContatoController*, que realiza uma busca nos contatos se baseando no que se encontra digitado no campo texto e retorna as opções possíveis. Ao selecionar um item (evento *onItemSelect)*, a ação show (e consequentemente a visão show) do controlador *ContatoController* será invocada. A visão show será discutida na próxima seção. A implementação de searchAJAX do controlador *ContatoController* encontra-se a seguir.

```
def searchAJAX = {
        def contatos = Contato.findAllByNomeLike("%${params.query}%")
// busca contatos
        //Cria resposta XML
        render(contentType: "text/xml") { // retorna os contatos
encontrados
            results() { contatos.each { contato ->
                    result(){
                        name(contato.nome)
                        id (contato.id)
                    }
                }
            }
        }
    }
```

A Figura 66 ilustra um exemplo do uso do *autocomplete* na busca de contatos. Para esse exemplo, deve-se supor que existem três contatos de cadastros: (1) Ana Barros, (2) Marcelo Santos e (3) Roberto Silva.

- A Figura 66(a) ilustra a situação em que o usuário digitou "A". Nesse caso, os três contatos estão presentes nas opções disponíveis desde que a sentença "A" se encontra presente nos nomes dos três contatos – Ana Barros, Marcelo Santos e Roberto Silva. A implementação de *searchAJAX* não leva em consideração a caixa (maiúscula ou minúscula) do texto digitado.

- A Figura 66(b) ilustra a situação em que o usuário digitou “An”. Nesse caso, apenas dois contatos estão presentes nas opções disponíveis desde que a sentença “An” se encontra presente nos nomes de dois contatos – Ana Barros e Marcelo Santos.
- Por fim, a Figura 66(c) ilustra a situação em que o usuário digitou “Ana”. Nesse caso, apenas um contato está presente nas opções disponíveis desde que a sentença “Ana” se encontra presente no nome de apenas um contato – Ana Barros. Ao selecionar essa opção, a aplicação apresenta as informações desse contato (Figura 67).

Figura 66 Funcionalidade AJAX – *autocomplete*.

8.5.5.4 Visão show

Na visão show (arquivo *show.gsp*), que é responsável pela apresentação das informações dos contatos, foram realizadas duas alterações.

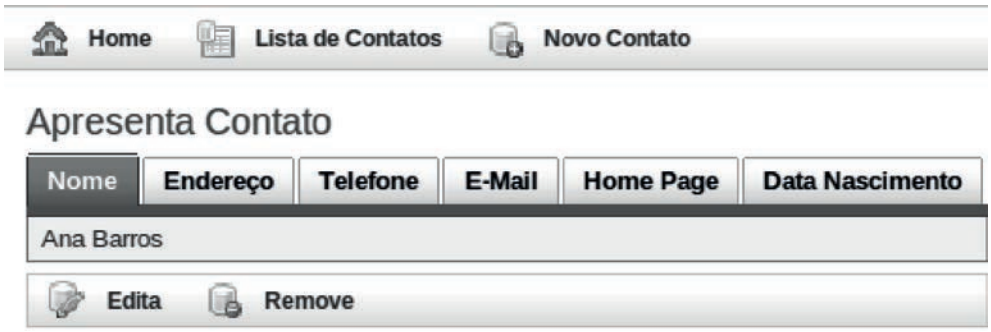
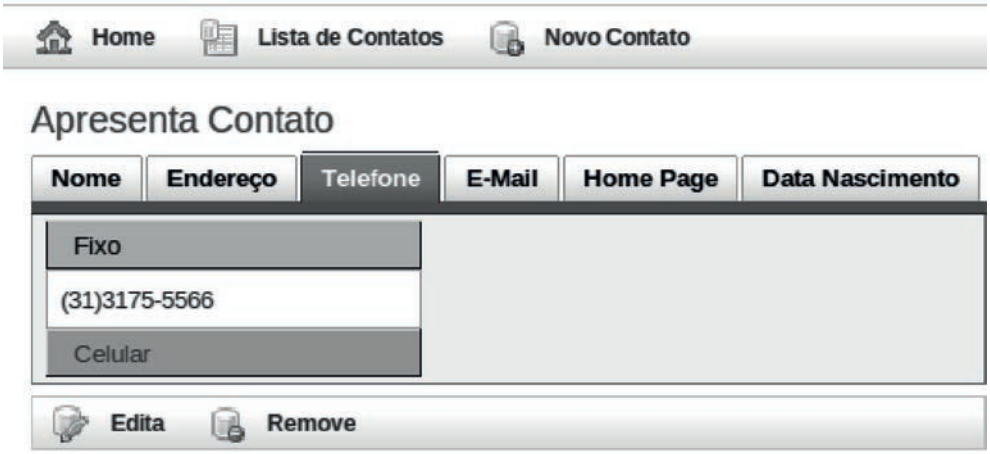


Figura 67 Visão show – abas.

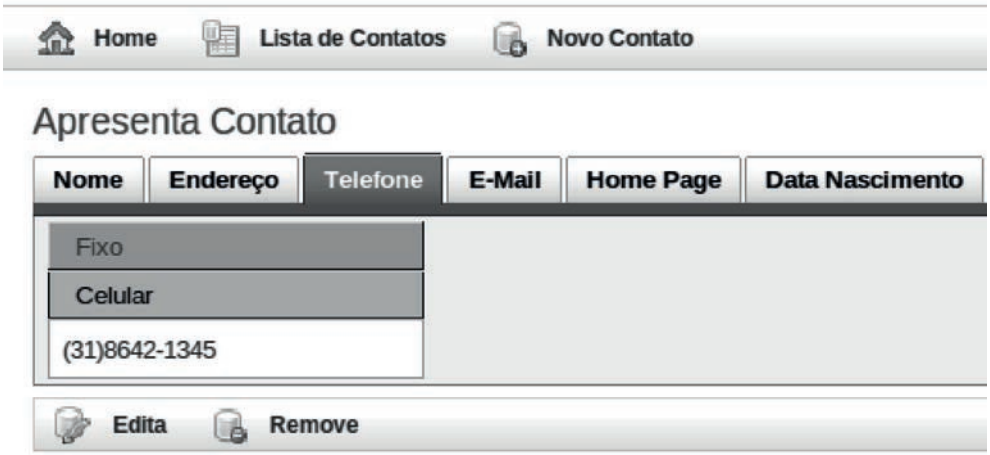
As duas alterações objetivam melhorar a estética da aplicação. A primeira consiste em utilizar visualizações em abas (Figura 67). Para tal será utilizado o componente AJAX *tabView* do plugin *richui*. A segunda consiste em utilizar menus *accordion*(). Desse modo, será utilizado o componente AJAX *accordion* do plugin *richui*. Trechos mais relevantes do código-fonte da visão *show*, relacionados a essas alterações, são apresentados a seguir.

```
<head>
<resource:tabView />
<resource:accordion skin=”default” />
</head>
<body>
<richui:tabView id=”tabView”>
      <richui:tabLabels>
        <richui:tabLabel selected=”true” title=”Nome” />
        <richui:tabLabel title=”Endereço” />
        <richui:tabLabel title=”Telefone” />
        <richui:tabLabel title=”E-Mail” />
        <richui:tabLabel title=”Home Page” />
        <richui:tabLabel title=”Data Nascimento” />
      </richui:tabLabels>

      <richui:tabContents>
          <richui:tabContent>
            ${fieldValue(bean: contatoInstance, field: “nome”)}
        </richui:tabContent>
<richui:tabContent>
          <richui:accordion>
            <richui:accordionItem id=”1” caption=”Fixo”>
```

```
  ${fieldValue(bean: contatoInstance, field: "fixo")}
            </richui:accordionItem>
            <richui:accordionItem caption="Celular">
                 ${fieldValue(bean: contatoInstance, field: "celular")}
            </richui:accordionItem>
          </richui:accordion>
        </richui:tabContent>

        </richui:tabContents>
</richui:tabView>
</body>
```



a) Acessando telefone → fixo



b) Acessando telefone → celular

Figura 68 Visão show – menu *accordion*.

Observações:

1. Pelo trecho de código, pode-se observar que é necessário incluir as tags *<resource:tabView>* e *<resource:accordion>*. Essas tags informam ao Grails que os componentes AJAX *richui:tabView* e *richui:accordion* serão utilizados na visão.
2. A tag *<richui:tabLabel>* define a legenda de cada aba presente na Figura 67.
3. Ao selecionar uma aba, seu conteúdo é apresentado. A tag *<richui:tabContent>* define o conteúdo de cada aba. O conteúdo da aba "Nome" é apresentado na Figura 67.
4. No caso da aba "Telefone", foi utilizado um menu *accordion*, conforme apresentada na Figura 68. O menu *accordion* é definido pelas tags *<richui:accordion>* e *<richui:accordionItem>*. As Figuras 68(a) e 68(b) apresentam as situações em que os itens "Fixo" e "Celular" do menu são selecionados.

8.5.5.5 Configuração do controlador padrão

Finalizando a implementação da aplicação **Agenda**, o arquivo URLMapping.groovy (diretório conf) foi alterado para definir o controlador/ação padrão. Usando a configuração apresentada, a URL – <http:/localhost:8080/Agenda> – direciona para a ação *index* do controlador *ContatoController*.

```
"/" {
            controller = "contato"
            action = "index"
 }
```

8.6 Automação de testes

Finalizando esta unidade, esta seção apresenta algumas características de Grails que facilitam a automação de testes. Relembrando: testes automatizados te dão segurança no momento da manutenção.

No entanto, antes do início da discussão sobre tais características, é fundamental relembrar a diferença entre testes unitários e de integração. *Testes*

unitários levam em consideração a unidade a ser verificada isoladamente. Não há conexões com bancos de dados ou qualquer outro tipo de componente: a unidade deve ser vista como um elemento isolado (não se comunica com ninguém).

Testes de integração, por sua vez, levam em consideração, como o próprio nome já diz, a integração da unidade a ser testada com componentes externos, como, por exemplo, bancos de dados ou outros serviços de natureza diversa. Testes de integração são, portanto, mais caros do ponto de vista computacional, visto que é necessário iniciar a aplicação para que estes possam ser executados.

Todos os testes se encontram no diretório *test/unit* (ou *integration*) presente na raiz do projeto Grails. Toda vez que é criada uma classe de domínio, controlador (e outros artefatos), testes automaticamente são incluídos no diretório *test/unit*. Da mesma forma que o GORM é fortemente baseado no framework hibernate, o arcabouço de testes presente em Grails é fortemente baseado no JUnit (discutido na Unidade 5).

Há três maneiras de se criar esses testes: (1) o Grails os cria automaticamente, (2) a classe de teste é criada manualmente pelos desenvolvedores e (3) a classe de teste é criada usando o comando *grails create-unit-test*.

Assim como diversos aspectos do Grails, aqui é necessário ater-se a algumas convenções. Toda classe de teste possui o sufixo Tests em seu nome. Sendo assim, os testes unitários para a classe de domínio *Usuario*, por exemplo, ficariam em *test/unit/UsuarioTests.groovy*.

O comando *grails create-unit-test* ou *grails create-integration-test* deve receber o nome do teste unitário ou de integração a ser gerado. Não é necessário incluir o *Tests* no final do arquivo, Grails o inclui automaticamente.

Testando classes de domínio. Ao lidar com linguagens dinâmicas como Groovy é frequente a necessidade de lidar com o seguinte problema: como testar uma classe que contém métodos e atributos que só serão injetados em tempo de execução? Funções como *save()*, *validate()* ou *constraints* só funcionam após injetados pelo *framework*.

Uma possibilidade é escrever testes de integração. O problema é que leva tempo até a aplicação ser iniciada – o que provavelmente irá reduzir a sua produtividade. O ideal é executar testes unitários, que por sua própria natureza são ordens de magnitude mais rápidas. A solução para o problema é usar *mock objects*.

Para ilustrar, tenha como base esta classe de domínio:

```
class Usuario {
    String nome
    String login
    static constraints = {
         nome(nullable:false, blank:false, maxSize:128, unique:true)
         login(nullable:false, blank:false, maxSize:16, unique:true)
    }
}
```

O teste unitário encontra-se na classe a seguir:

```
import grails.test.*
     class UsuarioTests  extends GrailsUnitTestCase {
        protected void setUp() {super.setUp()}
       protected void tearDown() {super.tearDown()}
       void testConstraints() {
          mockDomain Usuario
          def usuario = new Usuario()
          assertFalse usuario.validate() // usuário não validado pois
não incluiu campos obrigatórios
          def usuario_ok = new Usuario(nome:”Maria”, login: “maria”)
          assertTrue usuario_ok.validate()
       }
}
```

Vale a pena salientar o seguinte ponto: mesmo se tratando de um teste unitário, o teste exercita métodos que só existem em tempo de execução, no caso, o *validate*. Para isso, foi utilizado o método *mockDomain*, herdado de GrailsUnitTestCase. Este injeta na classe de domínio todos os métodos que uma classe desse tipo deve ter – como exemplos, os métodos de validação, *save()*, *delete()* etc. Assim, é possível testar facilmente a validação.

No caso de testes de integração, obviamente você não precisa do método *mockDomain*, pois as classes já estão prontas.

Testes unitários para controladores. É muito comum encontrarmos projetos nos quais apenas classes de domínio são testadas. Para testar seus controladores, você deve criar um teste tal como faria normalmente. A diferença é que esse teste não estenderá a classe *GrailsUnitTestCase*, e sim *ControllerUnitTestCase*.

Executando os testes. Com a aplicação funcionando, o próximo passo é executar seus testes. Para isso, deve-se usar o comando *grails test-app*, que executará todos os seus testes.

Para executar apenas alguns testes, basta passar como parâmetro os nomes dos testes excluindo o sufixo Tests, como exemplo *grails test-app Usuario*.

Para executar apenas testes unitários, execute *grails test-app unit* e para executar apenas os testes de integração, execute *grails test-app integration*.

Executados os seus testes, será criado o diretório *target/test-reports* em seu projeto, contendo o relatório de execução dos testes.

8.7 Estudos complementares

Para estudos complementares sobre o *framework* Grails abordado nesta unidade, o leitor interessado pode consultar a seguinte referência:

GRAILS. *Documentation Portuguese*. Disponível em: <http://www.grails.org/Documentation+Portuguese>. Acesso em: 23 jan. 2012. Consiste em um sítio web sempre atualizado com a documentação (livros, artigos, tutoriais etc.) sobre esse *framework*.

REFERÊNCIAS

ALTARAWNEH, H.; SHIEKH, A. A Theoretical Agile Process Framework for Web Applications Development in Small Software Firms. In: VI INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING RESEARCH, MANAGEMENT AND APPLICATIONS, IEEE Computer Society, 2008. p. 125-132.

AMBLER, S. Using Use-Cases. *Software Development*, v. 3, n. 6, p. 53-61, jul. 1995.

AOYAMA, M. Web-based Agile Software Development. *IEEE Computer*, v. 15, n. 6, p. 56-65, nov. 1998.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 2. ed. Boston: Addison-Wesley, 2003.

BECK, K. *Extreme Programming Explained*: Embrace Change. Boston: Addison-Wesley, 1999.

BECK, K.; ANDRES, C. *Extreme Programming Explained*: Embrace Change. 2. ed. Boston: Addison-Wesley, 2004.

BECK, K.; BEEDLE, M.; BENNEKUM, A.; COCKBURN, A.; CUNNINGHAM, W.; FOWLER, M.; GRENNING, J.; HIGHSMITH, J.; HUNT, A.; JEFFRIES, R.; KERN, J.; MARICK, B.; MARTIN, R. C.; MELLOR, S.; SCHWABER, K.; SUTHERLAND, J.; THOMAS, D. *Manifesto for Agile Software Development*. Disponível em: <http://agilemanifesto.org/>. Acesso em: 23 jan. 2012.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *UML*: Guia do Usuário. 2. ed. Rio de Janeiro: Campus, 2005.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. *Pattern-Oriented Software Architecture*: a System of Patterns. 1. ed. New Jersey: Wiley & Sons, 1996.

CACHERO, C.; KOCK, N.; GÓMEZ, J.; PASTOR, O. *Conceptual Navigation Analysis*: a Device and Platform Independent Navigation Specification. 2. ed. New Jersey: International Workshop on Web-Oriented Technology, 2002.

COAD, P.; LEFEBVRE, E.; DE LUCA, J. *Java Modeling in Color with UML*: Enterprise Components and Process. New Jersey: Prentice-Hall, 1999.

COCKBURN, A. *Writing Effective Use-Cases*. Boston: Addison-Wesley, 2001.

_____. *Agile Software Development*. Boston: Addison-Wesley, 2002.

DELAMARO, M. E.; CHAIM, M. L.; VINCENZI, A. M. R. Técnicas e Ferramentas de Teste de Software. In: MEIRA Jr., W.; CARVALHO, A. C. P. L. F. (Orgs.). *Atualizações em Informática*. Rio de Janeiro: PUC-Rio, 2010.

DIX, A.; FINLAY, J.; ABOWD, G.; BEALE, R. *Human Computer Interaction*. New Jersey: Prentice-Hall, 2004.

FOWLER, M. *UML Essencial*: um breve guia para a linguagem-padrão de modelagem de objetos. Porto Alegre: Bookman, 2005.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Padrões de Projeto*: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.

HEINEMAN, G. T.; COUNCIL, W. T. *Component-Based Software Engineering*: Putting the Pieces Together. 1. ed. Boston: Addison-Wesley, 2001.

KÖNIG, D. *Groovy in Action*. 1. ed. New York: Manning Publications Co., 2007.

KRASNER, G.; POPE, S. A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, v. 1, n. 3, p. 26-49, ago. 1988.

MESZAROS, G. *XUnit Test Patterns*: Refactoring Test Code. 1. ed. Boston: Addison-Wesley, 2007.

NIELSEN, J. *Designing Web Usability*. Indianapolis: New Riders Publishing, 2000.

PALMER, S.; FELSING, J. M. *A Practical Guide to Feature*: Driven Development. New Jersey: Prentice-Hall, 2002.

POWELL, T. A. *Web Design*. São Paulo: McGraw-Hill, 2000.

POWELL, T. A.; JONES, D.; CUTTS, D. *Web Site Engineering*: Beyond Web Page Design. New Jersey: Prentice-Hall, 1998.

PRESSMAN, R. S. Can Internet-Based Applications Be Engineered? *IEEE Software*, v. 15, n. 5, p. 104-110, set. 1998.

_____. *Engenharia de Software*. 6. ed. São Paulo: McGraw-Hill, 2006.

PRESSMAN, R. S.; LOWE, D. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

REEL, J. S. Critical Sucess Factors in Software Projects. *IEEE Software*, p. 18-23, maio 1999.

SCHWABER, K. *Agile Project Management with SCRUM*. Washington: Microsoft Press, 2004.

SCHWABER, K.; SUTHERLAND, J. *Scrum Guide*. Disponível em: <http://scrum.org>. Acesso em: 23 jan. 2012.

SHARP, H.; ROGERS, Y.; PREECE, J. *Interaction Design*: Beyond Human-Computer Interaction. 2. ed. Hoboken: Wiley & Sons, 2007.

SHAW, M.; GARLAN, D. *Software Architecture*: Perspectives on an Emerging Discipline. New Jersey: Prentice-Hall, 1996.

SOMMERVILLE, I. *Engenharia de Software*. 8. ed. São Paulo: Pearson Addison-Wesley, 2007.

SRIDHAR, M.; MANDYAM, N. *Effective Use of Data Models in Building Web Applications*. Disponível em: <http://www2002.org/CDROM/alternate/698/>. Acesso em: 23 jan. 2012.

SZYPERSKI, C. *Component Software*: Beyond Object-Oriented Programming. 2. ed. Boston: Addison-Wesley, 2002.

TELES, V. M. *Extreme Programming*. São Paulo: Novatec, 2004.

_____. *Programação em par*. Disponível em: <http://improveit.com.br/xp/praticas/programacao_par>. Acesso em: 23 jan. 2012.

WILLIAMS, L.; KESSLER, R. All I Really Need to Know about Pair Programming I Learned in Kindergarten. *Communications of the ACM*, v. 43, n. 5, p. 108-114, maio 2000.

WIRFS-BROCK, R.; WILKERSON, B.; WEINER, L. *Designing Object-Oriented Software*. New Jersey: Prentice-Hall, 1990.

YOO, J.; BIEBER, M. A Systematic Relationship Analysis for Modeling Information Domains. In: ROSSI, M.; SIAU, K. (Orgs.). *Information Modeling in the New Millennium*. Hershey: Idea Group, 2001.

ZIMMERMANN, H. OSI Reference Model: the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, v. 28, n. 4, p. 425-432, abr. 1980.

SOBRE O AUTOR

Delano Medeiros Beder

Delano Medeiros Beder, doutor pela Unicamp em 2001, é professor adjunto do Departamento de Computação da Universidade Federal de São Carlos (UFSCar). Tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software, atuando principalmente nos seguintes temas: técnicas de estruturação de software (padrões, arquiteturas, componentes de software etc.) e tolerância a falhas (tratamento de exceções, mecanismos de recuperação de erros e ações atômicas coordenadas), tendo publicado suas pesquisas em relevantes conferências e periódicos da Ciência da Computação. Recentemente foi revisor técnico da tradução brasileira de uma das principais referências bibliográficas da área de Engenharia Web – o livro *Web Engineering: a Practitioner's Approach*, de Roger Pressman e David Lowe.

Este livro foi impresso em março de 2013 pelo Departamento de Produção Gráfica – UFSCar.