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

GRENJ: UM FRAMEWORK OBTIDO POR UM PROCESSO ITERATIVO DE REENGENHARIA APLICANDO TDD

VINÍCIUS HUMBERTO SERAPILHA DURELLI

São Carlos/SP
Maio/2008

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

D955gr

Durelli, Vinícius Humberto Serapilha.
GRENJ: um framework obtido por um processo iterativo de reengenharia aplicando TDD / Vinícius Humberto Serapilha Durelli. -- São Carlos : UFSCar, 2008.
128 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2008.

1. Reengenharia de software. 2. Reengenharia orientada a objetos. 3. Frameworks. 4. Desenvolvimento guiado por testes I. Título.

CDD: 005.1 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

"GRENJ: um framework obtido por um processo iterativo de reengenharia aplicando TDD"

VINÍCIUS HUMBERTO SERAPILHA DURELLI

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

Membros da Banca:

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

Profa. Dra. Rosana Teresinha Vaccare Braga
(ICMC/USP)

Prof. Dr. Marcelo de Almeida Maia
(UFU)

São Carlos
Maio/2008

Dedico este trabalho à minha mãe, ao meu irmão e à minha avó materna.

Agradecimentos

Agradeço primeiramente a Deus por esta oportunidade valiosa.

À Professora Dra. Rosângela pela confiança, amizade e, principalmente, por conduzir e participar ativamente da realização deste trabalho. Obrigado por revisar minuciosamente tanto os documentos quanto as idéias apresentadas.

À Professora Dra. Rosana Braga pelo apóio e pelas inúmeras dúvidas esclarecidas. Muitíssimo obrigado por ter respondido meus e-mails sempre tão rapidamente.

À minha amiga Simone por ter me curado de, pelo menos, sete resfriados e da insolação em Florianópolis. Agradeço imensamente a ajuda com as figuras e os convites para tomar café exatamente quando eu precisava descansar um pouco. Aproveito para pedir desculpas por ter feito você passar esses últimos meses tendo que dizer tudo três vezes.

À minha mãe por sempre me incentivar a continuar estudando.

À CAPES pelo apoio financeiro.

"Sei que Deus não me dará nada que não possa lidar. Apenas gostaria que Ele não confiasse tanto em mim."
— Madre Teresa de Calcutá

"Experience is a hard teacher because she gives the test first, the lesson afterward.
— Provérbio Chinês

"TDD uses tests to motivate and inform design, refine and communicate intent, and smooth the troubled emotional waters of programming."
— Kent Beck

"The tests are a canary in a coal mine revealing by their distress the presence of evil design vapors."
— Kent Beck

Resumo

Os sistemas de software devem atender aos requisitos dos usuários e utilizar as tecnologias atuais. Dessa forma, esses sistemas permanecem em constante evolução e passam por várias atividades de manutenção a fim de se adequar tanto às necessidades dos usuários quanto às tecnologias. Quando há mudança de tecnologia, uma das formas de revitalizar esses sistemas é com a aplicação de reengenharia, classificada como manutenção preventiva. Este projeto de pesquisa apresenta o processo iterativo usado durante a realização da reengenharia do framework caixa-branca GREN (Gerenciamento de REcursos de Negócios), construído com base na linguagem de padrões GRN (Gerenciamento de Recursos de Negócios). O framework GREN foi implementado originalmente em Smalltalk, linguagem de programação muito difundida, principalmente, na comunidade acadêmica. No processo de reengenharia proposto, as linguagem Java™ e AspectJ, extensão da linguagem Java™ para programação orientada a aspectos, foram utilizadas para implementação da nova versão do framework, denominada GRENJ (Gerenciamento de REcursos de Negócios em Java™). O processo de reengenharia proposto pode ser utilizado na reengenharia de outros sistemas desenvolvidos com o paradigma OO, em que a mudança de linguagem de programação é desejada, pois é iterativo, incremental, utiliza padrões de engenharia reversa, desenvolvimento guiado por testes (*Test-Driven Development*, TDD) e refatoração. O framework GRENJ tem aproximadamente vinte e oito mil linhas de código fonte, das quais pouco mais de dez mil estão relacionadas aos testes criados durante o desenvolvimento. A documentação desse framework foi produzida utilizando a ferramenta *javadoc*. Um sistema hipotético que atende aos requisitos básicos de uma locadora de DVDs foi instanciado usando o GRENJ.

Abstract

Software systems must satisfy the users' requeriments and use current technologies. Thus, these systems are in constant evolution and several software maintenance tasks are carried out in order to adapt them to the user's requirements as well as to the current technologies. When there is a change in the technology used by these systems, an approach to their revitalization is reengineering, which is classified as a preventive maintenance. This research project presents the iterative process applied during the reengineering of the GREN framework, which is built based on the GRN pattern language. The GREN framework was originally implemented in Smalltalk, a very spread programming language, mainly within the academic community. In the proposed reengineering process the Java™language and AspectJ, which is an aspect-oriented Java™extension, were used during the framework implementation resulting from the reengineering process. This new framework is called GRENJ. The proposed process can be used to reengineer systems developed with OO paradigm, in order to change its implementation language to another OO language. That process is iterative, incremental, and applies reverse engineering patterns, test-driven development and refactoring. The GRENJ framework has approximately twenty-eight thousand source code lines, of which approximately ten thousand are related to tests created during the development. The GRENJ framework documentation was generated using the javadoc tool. A hypothetical system that satisfies the basics requirements of a rental video store was instantiated using the GRENJ framework.

Sumário

Lista de Figuras

Lista de Tabelas

Listagens

Lista de Abreviaturas

API: Interface de Programação de Aplicativos (*Application Programming Interface*)
ARA: Arcabouço de Reengenharia Ágil (Cagnin, 2005)
GREN: Framework para Gestão de Recursos de Negócio
GRENJ: Framework para Gestão de Recursos de Negócio implementado na linguagem Java™
GRN: Linguagem de Padrões para Gestão de Recursos de Negócios
GoF: *Gang of Four*, apelido atribuído aos autores Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides
HTML: acrônimo de *HyperText Markup Language*
IBM: *International Business Machines*
IDE: Ambiente de Desenvolvimento Integrado (*Integrated Development Environment*)
JDK: *Java SE Development Kit*
MIT: Instituto Tecnológico de Massachusetts (*Massachusetts Institute of Technology*)
MVC: *Model-View-Controller*
PDF: *Portable Document Format*
POA/AOP: Programação Orientada a Aspectos/*Aspect Oriented Programming*
TDD: Desenvolvimento Guiado por Testes (*Test-Driven Development*)
UML: Linguagem de Modelagem Unificada (*Unified Modeling Language*)
XP: acrônimo de *eXtreme Programming*

CAPÍTULO
1

Introdução

1.1 Contextualização

Reúso é o processo de construir software utilizando artefatos existentes (Krueger, 1992; Shiva e Shala, 2007). A reutilização de artefatos de software é uma das formas que a Engenharia de Software apresenta para aumento da produtividade e da qualidade do produto gerado. Existem diversas técnicas que possibilitam o reúso de artefatos de software em todas as fases do desenvolvimento de software. Diferentes tipos de artefatos podem ser considerados dependendo da técnica utilizada, como por exemplo, código, documentação, artefatos de análise e projeto, conhecimento, casos de teste, experiência adquirida em desenvolvimentos anteriores, entre outros.

O surgimento da programação orientada a objetos trouxe vários benefícios para o desenvolvimento de software, entre eles: encapsulamento, modularidade e polimorfismo. Considerando-se a reutilização de software, a principal vantagem em utilizar programação orientada a objetos é o reúso obtido tanto por meio de herança quanto por composição. No entanto, esses dois tipos de reúso ainda são de pequena escala, envolvendo somente elementos de menor granularidade.

Com o intuito de reusar não somente código, mas também projeto, surgiram os frameworks, que são uma técnica de reúso orientada a objetos e específica de domínio (Johnson, 1997b). Frameworks são soluções reutilizáveis, de todo ou parte do sistema, "semi completas", no sentido de que entidades abstratas são customizadas com o propósito de definir o comportamento específico (Johnson, 1997a). Eles proporcionam modularidade, reusabilidade, tanto de projeto quanto de código fonte, extensibilidade e inversão de controle (Fayad e Schmidt, 1997; Gamma et al., 1995;

Cunningham et al., 2005).

É desejável que os frameworks sejam mais flexíveis para permitir a criação de várias aplicações específicas. Geralmente, essa flexibilidade é alcançada por meio de padrões de projeto. Padrões são uma forma de captar e documentar a experiência contida em soluções de sucesso para problemas comuns do software (Fowler, 2003). Dessa forma, além de possibilitar a reutilização de soluções bem-sucedidas, proporcionam, também, um vocabulário comum de alto-nível de abstração, que pode ser empregado para facilitar a comunicação entre os desenvolvedores de software (Harrison et al., 2007). Existem padrões em vários níveis de abstração, por exemplo: análise (Fowler, 1996), projeto (Gamma et al., 1995), implementação (Beck, 2007), engenharia reversa e reengenharia (Demeyer et al., 2002).

Uma coleção estruturada de padrões, específicos de um domínio e organizados em árvores ou grafos, sendo que cada padrão leva a uma série de outros padrões, é denominada linguagem de padrões (Brugali e Sycara, 2000). Alguns trabalhos têm-se aproveitado das similaridades existentes entre frameworks e linguagens de padrões. Brugali e Sycara (2000) afirmam que linguagens de padrões podem ser utilizadas para gerar e documentar frameworks e Braga (2002b) propõe um processo para construção e instanciação de frameworks baseados em uma linguagem de padrões.

As técnicas de reúso facilitam a criação de aplicações, pois os desenvolvedores não necessitam partir do zero (*from scratch*) ao iniciar. Todavia, o software deve evoluir de acordo com as necessidades dos usuários, no entanto, modificações sempre ocorrem. Devido a essas modificações requeridas, a arquitetura do sistema se modifica em relação à original e as alterações se tornam cada vez mais complexas e custosas (Mens e Tourwe, 2004), além de cada vez mais propensas a introduzir falhas. Sistemas de software, normalmente, também são alterados a fim de se adaptarem às tecnologias mais recentes, o que implica em mudanças de plataforma, compilador, linguagem, etc. Este trabalho se encaixa nesse contexto, pois propõe a reengenharia iterativa de um framework caixa-branca com o propósito de implementá-lo em uma linguagem mais difundida e atual. Para tal, o processo de reengenharia que foi aplicado emprega padrões de engenharia reversa e algumas práticas ágeis.

A motivação para realização desta dissertação é comentada na Seção 1.2, os objetivos são discutidos na Seção 1.3 e a organização da dissertação é apresentada na Seção 1.4.

1.2 Motivação

Quando determinada linguagem torna-se menos difundida, gradualmente, a evolução das aplicações implementadas nessa linguagem é prejudicada. Pois, o número de pessoas com o conhecimento necessário para realizar modificações nessas aplica-

ções diminui. Posteriormente, a maioria dessas aplicações é migrada para linguagens mais atuais, a fim de que possam continuar passando por atividades de manutenção. A migração pode ser realizada de maneira automatizada, empregando ferramentas que realizam a conversão entre linguagens. Todavia, a conversão automatizada entre linguagens de programação é subestimada. É complexo alcançar a equivalência semântica necessária nesse tipo de conversão (Terekhov e Verhoef, 2000), principalmente quando as duas linguagens envolvidas têm características diferentes. Certas linguagens, como Smalltalk, possuem determinadas propriedades que tornam a conversão ainda mais complicada (Engelbrecht e Kourie, 2003).

Uma abordagem alternativa para efetuar a conversão entre linguagens é a reengenharia. Geralmente, reengenharia envolve engenharia reversa e engenharia avante ou reestruturação. A engenharia reversa é realizada para extração do projeto existente e para auxiliar na compreensão do sistema legado. A engenharia avante para realizar os ajustes necessários para conversão e implementação na linguagem escolhida. Atividades de reestruturação podem ser realizadas com o objetivo de aperfeiçoar a legibilidade do código fonte passando por reengenharia, facilitando assim a compreensão e extração do projeto.

O framework GREN (Gestão de REcursos de Negócios) (Braga, 2002b) é um framework caixa-branca implementado na linguagem Smalltalk, uma linguagem dinamicamente tipada e orientada a objetos. O GREN utiliza o banco de dados relacional MySQL (MySQL, 2007) para persistência dos dados. A evolução desse framework é prejudicada devido à ausência de pessoas aptas para realizar alterações e aprimoramentos, pois, Smalltalk já não é mais amplamente difundida nas comunidades acadêmica e profissional de Engenharia de Software. Além disso, a linguagem Smalltalk não é independente de plataforma e existe incompatibilidade entre as implementações disponíveis.

Java™, assim como Smalltalk, é uma linguagem orientada a objetos, interpretada e com gerenciamento automático de memória. Porém, diferentemente da linguagem Smalltalk, ela é estática e fortemente tipada, ou seja, as variáveis devem ser associadas a um tipo específico durante a declaração e há restrições sobre quais tipos podem ser envolvidos em determinadas operações. É possível obter informações sobre os tipos e seus relacionamentos até mesmo em tempo de execução, o que caracteriza a linguagem Java™como *fully runtime-typed* (Niemeyer e Knudsen, 2005). Além disso, talvez a principal vantagem da linguagem Java™é que, além de ser uma das linguagens contemporâneas mais difundidas, sua semelhança com as linguagens C/C++ facilita a sua compreensão e utilização.

Já foram realizados vários trabalhos utilizando o GREN, comprovando a sua eficácia (Silva et al., 2004; Cagnin, 2005). No entanto, se o GREN fosse implementado em uma linguagem mais difundida a sua evolução seria facilitada, além disso, isso

estimularia sua utilização em mais pesquisas. Assim, uma versão do framework implementada na linguagem Java™ apresenta todos os benefícios mencionados anteriormente. Adicionalmente, aplicações implementadas na linguagem Java™ podem ser mais facilmente portadas para Web, entre outras vantagens em relação à Smalltalk.

Usualmente, no desenvolvimento "tradicional" de software, também conhecido como *test-last* (Erdogmus et al., 2005), os testes são realizados somente após as atividades de análise, projeto e implementação – independentemente da linguagem empregada. Utilizando a prática denominada desenvolvimento guiado por testes (*Test-Driven Development*) ou, simplesmente, TDD (Beck, 2002), os testes são implementados antes da implementação total da funcionalidade. Desenvolvendo de acordo com essa prática, a funcionalidade a ser implementada é decomposta em uma lista de casos de teste. Esses casos de teste são implementados conforme avança-se no desenvolvimento do sistema e são considerados satisfeitos somente quando a funcionalidade avaliada por eles estiver devidamente implementada. TDD também envolve atividades de refatoração, que consiste em uma maneira disciplinada de aperfeiçoar o código existente. Nesse contexto, os testes produzidos auxiliam na detecção de possíveis problemas durante a realização das refatorações. Visto que testes são essenciais para garantir a qualidade dos sistemas, a implementação de testes desde o início, por meio da utilização de TDD, apóia a produção de sistemas com mais qualidade.

Este trabalho apresenta a reengenharia do framework GREN como exemplo de um processo de reengenharia que utiliza iteratividade e aplica TDD. O framework resultante do processo de reengenharia foi denominado Gestão de REcursos de Negócio em Java™ (GRENJ). Esse framework implementa a mesma funcionalidade do GREN, exceto que sua interface gráfica não foi construída. É importante salientar que nenhuma ferramenta de conversão foi empregada durante o processo.

1.3 Objetivos

Os principais objetivos deste trabalho são: apresentar um processo iterativo de reengenharia, descrito no Capítulo 5, e aplicação desse processo para reengenharia do framework GREN.

O processo de reengenharia iterativo utilizado é composto de dois tipos de atividades. O primeiro tipo é relacionado à extração do projeto do sistema legado que está passando por reengenharia. Assim, esse primeiro tipo caracteriza-se como atividade de engenharia reversa. O segundo aborda a implementação na linguagem escolhida, aprimoramentos e ajustes necessários para que, gradualmente, o sistema resultante do processo de reengenharia seja produzido, de forma que esse segundo tipo de atividade pode ser considerado de engenharia avante e reestruturação.

As atividades de engenharia reversa têm como principal fonte de informações so-

bre o sistema legado o respectivo código fonte e a documentação disponível. Durante a realização dessas atividades, alterna-se entre consultas ao código fonte e à documentação, obtendo-se informações de baixo e alto nível sobre o sistema legado, respectivamente. O reúso de conhecimento, documentado na forma de padrões, é importante para reduzir o tempo e custo de desenvolvimento, bem como produzir um produto de qualidade. Assim, a realização dessas análises é "guiada" por padrões de engenharia reversa, definidos por Demeyer et al. (2002).

Como o processo de reengenharia usado é iterativo, o sistema legado é dividido em parcelas. Cada uma dessas parcelas passa por atividades de engenharia reversa e atividades de engenharia avante. As atividades de engenharia avante são aplicadas utilizando TDD. No contexto do processo iterativo proposto, as informações sobre cada parcela do sistema legado são decompostas em uma lista de casos de teste e implementadas aplicando TDD.

O processo descrito anteriormente foi aplicado para reengenharia do framework GREN. O GRENJ, resultante desse processo, é equivalente ao GREN, porém, implementado nas linguagens Java™ e AspectJ. O GRENJ utiliza o banco de dados relacional MySQL para persistência das informações e, como o GREN, é composto de duas camadas: de persistência e de negócios. Na camada de negócios encontram-se as implementações dos padrões da GRN (Braga et al., 1999; Braga, 2002b). A documentação associada ao GRENJ foi produzida com o auxílio da ferramenta *javadoc* (Sun Microsystems, Inc., 2007).

1.4 Organização

Este trabalho está organizado em sete Capítulos, incluindo este, e três Apêndices, além das referências bibliográficas. O Capítulo 2 reúne os principais conceitos que foram utilizados para que o objetivo deste projeto fosse alcançado. O Capítulo 3 tem enfoque na linguagem de padrões GRN e no framework GREN, além de apresentar informações sobre o GREN-Wizard e o *"cookbook"* do GREN.

O Capítulo 4 descreve os conceitos fundamentais do TDD e as ferramentas que apóiam o desenvolvimento aplicando essa prática. São apresentadas, especificamente, as ferramentas utilizadas durante a implementação do framework GRENJ: o JUnit (JUnit, 2007), que foi usado para facilitar a implementação e automatizar a execução dos testes, e a biblioteca EasyMock (EasyMock, 2008) que facilitou a geração dinâmica de objetos *mock*. Nesse Capítulo também são destacadas as definições de objetos *stub* e *mock*, bem como um exemplo de aplicação de TDD.

No Capítulo 5, o processo de reengenharia iterativo é apresentado, bem como a utilização dos padrões de Demeyer et al. (2002) no contexto do processo de reengenharia. A reengenharia do framework GREN e implementação do framework GRENJ

usando o processo proposto, também são discutidas.

No Capítulo 6, descreve-se a abordagem recomendada para instanciação de uma aplicação, pertencente ao domínio de gestão de recursos de negócios, utilizando o framework GRENJ. A abordagem é exemplificada por meio da instanciação de uma aplicação que satisfaz os requisitos propostos para uma locadora fictícia de DVDs. O Capítulo 7 apresenta as conclusões deste trabalho, as contribuições e limitações do mesmo, bem como sugestões de trabalhos futuros.

O Apêndice A apresenta uma introdução ao framework JUnit, ferramenta que foi empregada para apoiar a criação e automatização dos testes produzidos durante o desenvolvimento do framework GRENJ. O Apêndice B contém uma introdução à biblioteca usada para geração dinâmica de objetos *mock* durante a implementação dos testes do GRENJ. Por fim, no Apêndice C o *protocolo de reengenharia*, documento que foi preenchido para cada classe do GREN que passou por reengenharia, é descrito.

CAPÍTULO
2

Resenha Bibliográfica

2.1 Considerações Iniciais

Neste Capítulo são abordados os conceitos fundamentais, necessários para compreensão do trabalho, juntamente com algumas pesquisas que serviram de embasamento. Assim, a Seção 2.2 define reengenharia, engenharia reversa e reestruturação, descrevendo seus propósitos. Na Seção 2.3 o conceito de padrões é apresentado, padrões de projeto e de reengenharia são discutidos mais detalhadamente, dado que são mais relevantes para o contexto do trabalho.

A Seção 2.4 e a Seção 2.5 apresentam linguagens de padrões e frameworks de software orientados a objetos, respectivamente. Na Seção 2.6 é abordada a refatoração. A Seção 2.7 apresenta o conceito de aspectos, seus benefícios e a linguagem orientada a aspectos AspectJ e na Seção 2.8 são apresentadas as considerações finais.

2.2 Reengenharia, Engenharia Reversa e Reestruturação

Sistemas de software têm crescente importância na sociedade contemporânea. Essa dependência aparece em vários setores (Demeyer et al., 2005a): indústria, transporte, comércio, etc. Requisitos e ambientes, ambos em constantes mudanças, exercem influência sobre os sistemas de software. Lehman (1996) revisa algumas leis fundamentais relacionadas à evolução do software. Demeyer et al. (2005b) afirmam que as duas com maior impacto são:

Lei da Mudança Contínua: um sistema de software utilizado em ambientes reais deve mudar, senão se tornará progressivamente menos útil nesse ambiente.

Lei da Complexidade Crescente: conforme um sistema de software evolui, sua complexidade aumenta, assim "recursos extras" são necessários para preservar e simplificar sua estrutura.

Alguns sistemas de software são de vital importância para as organizações que os utilizam, sendo que não podem ser simplesmente descartados (Bianchi et al., 2003; Demeyer et al., 2002). Esse tipo de sistema é denominado sistema legado (*legacy system*). De acordo com a primeira lei de Lehman, alterações são inevitáveis, e, de acordo com a segunda lei, conforme o sistema de software é alterado sua estrutura se afasta da projetada originalmente e esforços adicionais precisam ser feitos para impedir essa degradação. Devido às alterações, a documentação do software quando disponível, geralmente não é confiável pois não registra todas as modificações realizadas nele (Demeyer et al., 2005b). Na maioria dos sistemas de software legados pode-se afirmar que o código é a única fonte de informações confiável (Müller et al., 2000). Sistemas valiosos para as organizações, que foram submetidos a várias alterações e tiveram sua estrutura original corrompida, são candidatos à reengenharia.

Não é a idade de um sistema de software que determina sua categorização como legado, o fator determinante é o número de vezes que o sistema foi alterado ou adaptado sem passar por reengenharia (Demeyer et al., 2002).

A reengenharia preocupa-se em *reestruturar* sistemas, com o objetivo de eliminar alguns problemas e ainda facilitar futuras alterações (Demeyer et al., 2002). Segundo Chikofsky e Cross II (1990) reengenharia consiste em examinar e alterar um sistema de software para reconstruí-lo em uma nova forma. Normalmente, reengenharia inclui atividades de engenharia reversa, seguida por atividades de engenharia avante ou reestruturação.

Müller et al. (2000) afirmam que a engenharia reversa pode ser definida como o processo de análise de um sistema legado com o objetivo de identificar seus componentes e o relacionamento entre eles, gerando modelos de mais alto-nível de abstração. Engenharia avante é o processo que transforma abstrações de mais alto-nível em modelos de baixo nível, cada vez mais dependentes de implementação. Engenharia reversa não implica em modificações no sistema legado, é um processo de análise, usado para recuperar modelos de mais alto-nível de abstração que os disponíveis, para alcançar a compreensão necessária para outras atividades.

A reestruturação procura abordar o problema da complexidade do software por meio de aprimoramentos incrementais da estrutura interna (Mens e Tourwe, 2004), envolvendo alterações de representações no mesmo nível de abstração. No contexto de sistemas orientados a objetos, o termo refatoração, difundido por Opdyke (1992) e Fowler et al. (1999), é empregado para descrever determinados tipos de reestruturações. O conceito de refatoração é apresentado na Seção 2.6.

2.3 Padrões

Geralmente, engenheiros de software são sobrecarregados com muitas decisões durante o desenvolvimento de software. Leva tempo para se tornar proficiente quando se trata de projeto orientado a objetos, deve-se encontrar as classes e definir suas interfaces, hierarquias de herança e estabelecer os relacionamentos entre as classes (Gamma et al., 1995). Desenvolvedores de software têm uma forte tendência de reutilizar soluções que atuaram bem no passado e conforme obtêm mais experiência, seu repertório aumenta, tornando-os mais proficientes (Beck et al., 1996). Antes do conceito de padrões ser introduzido no desenvolvimento de software, tal proficiência adquirida era restrita à experiência pessoal, ou seja, era raramente compartilhada entre desenvolvedores.

É possível observar padrões em estruturas do dia-a-dia como: construções (arquitetura), em organizações de pessoas, na natureza e no tráfico de veículos (Coplien, 1996). Originalmente a idéia de padrões partiu do campo da arquitetura, com Christopher Alexander. Mesmo considerando o fato de que ele se refere a padrões em edificações e cidades, os conceitos são igualmente válidos para padrões de projeto orientados a objetos (Gamma et al., 1995). Exposto de maneira mais genérica, segundo Grand (1998), as idéias apresentadas nos livros de Alexander são aplicáveis em vários campos além da arquitetura, incluindo o software. As soluções podem ser expressas em forma de objetos e interfaces ao invés de muros e portas.

A essência dos padrões é comunicar uma solução para determinado problema em um contexto (Gamma et al., 1995). De acordo com Shalloway e Trott (2001), padrões existem em todos os níveis de abstração: análise, projeto, implementação, etc. Padrões descrevem um problema, que pode ocorrer em alguma das fases do desenvolvimento de software. Juntamente com o problema, apresenta uma solução genérica em um contexto particular.

Segundo Gamma et al. (1995), um padrão de projeto denomina, abstrai e identifica pontos-chave de uma estrutura comum de projeto; são úteis quando empregados com o objetivo de produzir um projeto orientado a objetos reutilizável e flexível. Podem ser considerados como uma maneira de encapsular a experiência de programadores, de forma que seja compreensível para outros programadores (Agerbo e Cornils, 1998).

2.3.1 Formato dos Padrões

Padrões são comumente apresentados em um formato específico, com o propósito de introduzir o problema ao leitor, descrever o contexto no qual o problema pode ocorrer, analisar o problema e, posteriormente, apresentar e elucidar a solução (Coplien, 1996). Quando se escreve padrões, normalmente se escreve em algum formato padronizado (Fowler, 2003). Dentre os diversos formatos destacam-se o formato de *Por-*

tland, o formato *GoF*[1] e o formato de *Appleton*. Somente o formato GoF é apresentado aqui, sendo ele dividido nas seguintes seções (Gamma et al., 1995):

Nome do Padrão e Classificação: o nome deve ser claro e conciso, a denominação claramente acentua a idéia chave envolvendo padrões; tal idéia é a criação de um vocabulário de projeto mais alto-nível. A classificação dos padrões de projeto será discutida posteriormente, na Seção 2.3.2.

Intenção: tem o intuito de, resumidamente, informar o que o padrão de projeto deve fazer, sua razão, intenção e para qual assunto de projeto o padrão propõe uma solução.

Também conhecido como: informa o nome ou possivelmente a lista de outros nomes atribuídos ao padrão.

Motivação: traz um cenário que ilustra um problema de projeto e como as classes e estruturas de objetos pertencentes ao padrão devem se comportar para solucionar o problema.

Aplicabilidade: descreve as situações onde o padrão de projeto pode ser considerado uma solução plausível. Além disso, pode se necessário, informar quais exemplos de projeto pobre a aplicação do padrão pode evitar.

Estrutura: fornece uma representação gráfica dos participantes do padrão. Tanto modelos de classe e modelos de seqüência, entre outros, podem ser utilizados.

Participantes: as classes e possivelmente os objetos participantes do padrão; bem como suas responsabilidades.

Colaborações: como os participantes colaboram para realizar suas responsabilidades.

Conseqüências: essa seção se encarrega de informar sobre os custos e benefícios da utilização do padrão sendo considerado.

Implementação: ressalta as armadilhas, dicas ou técnicas que deve se conhecer quando o padrão for implementado.

Exemplo de Código: fragmento de código ilustrando uma possível implementação do padrão.

Usos Conhecidos: exemplos de sistemas *reais* onde o padrão foi empregado.

[1] GoF significa Gang of Four; apelido atribuído aos autores do livro "Design Patterns"(Gamma et al., 1995).

Padrões Relacionados: informa quais padrões de projeto são intimamente relacionados ao padrão subjacente, as diferenças e com quais outros padrões pode se empregar.

Fowler (2003) cita que somente dois componentes são indispensáveis na descrição de padrões: o *como* e o *quando*. O *como* deve descrever, de forma geral, uma maneira de implementar o padrão; o *quando* tem o propósito de expor alternativas e quando usá-las. Não há um consenso sobre isso. Diferentes autores fazem diferentes escolhas de projeto, com isso, suas descrições do mesmo padrão podem variar (Wirfs-Brock, 2006).

2.3.2 Tipos de Padrões

Idiomas são padrões pertencentes a um nível mais baixo de abstração em relação a padrões de projeto. Mais especificamente, idiomas definem técnicas de programação, específicas de linguagem, que descrevem como implementar determinados pontos usando características da linguagem subjacente. Além de padrões de projeto e idiomas, existem padrões em vários níveis de abstração, como por exemplo: análise (Fowler, 1996), engenharia reversa e reengenharia (conforme descrito na Seção 2.3.3) (Demeyer et al., 2002).

Adicionalmente à classificação por nível de abstração, Gamma et al. (1995) propõem que padrões de projeto sejam classificados por meio de dois critérios: propósito e escopo. A Tabela 2.1 apresenta essa classificação e definições de cada uma das categorias propostas. Todavia, é importante ressaltar que padrões pertencentes à diferentes categorias podem ser utilizados em conjunto. Por exemplo, *Composite*, um padrão de projeto estrutural, é normalmente utilizado com *Iterator* ou *Visitor*, ambos categorizados como comportamentais. Padrões também podem ser classificados em coleções, catálogos e linguagens (Seção 2.4) (Brugali e Sycara, 2000; Braga, 2002b).

Um dos principais benefícios do uso de padrões é a inserção de um vocabulário de mais alto-nível entre os engenheiros de software (projetistas, analistas, etc.) envolvidos no sistema. Isso permite que projetistas com experiência moderada compreendam e discutam projetos de software sem explicações adicionais (Fowler, 2003). Padrões de projeto também auxiliam na documentação dos sistemas e na compreensão de sistemas existentes, de princípios fundamentais e técnicas básicas, como por exemplo: herança e polimorfismo (Gamma et al., 1995; Shalloway e Trott, 2001). Entretanto, segundo Fowler (2003), não é a quantidade de padrões presentes em um sistema de software o fator determinante para a sua qualidade, é importante que os padrões apropriados tenham sido empregados.

Tabela 2.1: Classificação dos padrões de projeto (Gamma et al., 1995)

CLASSIFICAÇÃO			
PROPÓSITO		ESCOPO	
Criação	São padrões relacionados à criação (ou instanciação) de objetos.	**Classe**	Tratam da relação entre classes e suas subclasses, ilustram uma configuração estática, definida em tempo de compilação.
Estrutural	Tratam da composição de classes ou objetos.		
Comportamental	Caracterizam a forma como as classes ou objetos interagem e distribuem as responsabilidades.	**Objeto**	Abordam a relação entre objetos, podem ser alterados em tempo de execução.

2.3.3 Padrões de Reengenharia

Projetos de reengenharia, normalmente, deparam-se com alguns problemas típicos, sendo difícil utilizar ferramenta ou técnica específica para resolução de todos problemas. Um modo de registrar a experiência obtida na reengenharia de projetos é com a utilização de padrões. Ducasse et al. (1999b) citam que padrões de reengenharia enfocam o processo de transformar um sistema legado em um novo sistema refatorado (o conceito de refatoração é abordado na Seção 2.6), enquanto que padrões de projeto apresentam uma solução para um problema recorrente de projeto. Os problemas abordados por padrões de reengenharia são mais amplos que aqueles solucionados por padrões de projeto, pois tanto o contexto do ambiente de negócios quanto o contexto do sistema de software devem ser abordados, além de considerar fatores como o orçamento da organização (Stevens e Pooley, 1998).

Os padrões de reengenharia definidos por Ducasse et al. (1999a) são de mais baixo-nível, no sentido que tratam diretamente com o código-fonte. Porém existem padrões de reengenharia de mais alto-nível, que abrangem toda a estratégia para se abordar sistemas legados, como por exemplo os de Stevens e Pooley (1998). Demeyer et al. (2002) apresentam vários padrões de engenharia reversa, que abrangem desde o contato inicial com o sistema até a preparação para reengenharia e padrões de reengenharia que apóiam a realização do processo.

2.4 Linguagens de Padrões

Linguagens de padrões consistem de uma coleção estruturada de padrões, que refletem e documentam experiência. São específicas de domínio, uma linguagem de padrões organiza o conhecimento sobre um domínio em um conjunto de padrões, ambos específicos do domínio subjacente (Braga e Masiero, 2002c).

As linguagens de padrões podem ser organizadas como árvores ou grafos. Cada padrão leva a uma série de outros padrões pertencentes à linguagem (Brugali e Sycara, 2000). Podem ser empregadas como guias durante o processo de desenvolvimento de aplicações do mesmo domínio, fornecendo soluções alternativas e orientação sobre os assuntos que devem ser explorados, bem como a ordem que isso deve ser feito (Braga e Masiero, 2002c).

A Seção 3.2 apresenta a Linguagem de Padrões para Gestão de Recursos de Negócio que será usada neste trabalho.

2.5 Frameworks de Software Orientados a Objetos

Neste trabalho, frameworks de software orientados a objetos, por motivos de simplicidade, são referidos simplesmente como frameworks. Reúso é o processo de construir um novo sistema de software utilizando os artefatos existentes (Krueger, 1992). Esses artefatos não se restringem somente a código, mas a qualquer trabalho realizado durante o ciclo de vida do software. Abordagens "tradicionais" de reúso de código são (Fach, 2001): bibliotecas de componentes reutilizáveis e funções matemáticas. Frameworks são uma técnica de reúso orientada a objetos (Johnson, 1997b). São específicos de domínio, ou seja, são soluções "semi-completas", voltadas para domínio ao qual foram projetadas (Fayad e Schmidt, 1997).

Segundo Braga e Masiero (2002a) frameworks proporcionam o reúso de grandes estruturas de software, voltadas para um domínio particular, permitindo customizações ou extensões para criação de aplicações específicas. Frameworks proporcionam, além de reúso de código, reúso de projeto (Johnson, 1997b). Um framework pode ser definido como um conjunto de classes de software – abstratas e concretas – que fazem parte de um grande projeto abstrato, cujo objetivo é fornecer uma solução para uma família de problemas relacionados. Em outras palavras, um projeto genérico – em relação ao domínio – que descreve como a parte comum da família de aplicações pode ser decomposta em um conjunto de objetos e suas interações (Johnson, 1997a).

Johnson (1997b,a) define frameworks aproveitando-se de duas definições, uma para estrutura e outra para o propósito:

Estrutura: projeto reutilizável de todo ou parte do sistema, é representado por meio de um conjunto de classes abstratas e a maneira que suas instâncias interagem.

Propósito: servir de "esqueleto de aplicação", tal esqueleto pode ser ajustado por um desenvolvedor de aplicações com o objetivo de criar aplicações específicas.

Um framework é formado por classes concretas e abstratas, responsáveis por mapear as decisões do domínio. As classes concretas definem o comportamento comum

– fixo – para a família de aplicações do domínio. As classes abstratas devem ser "especializadas", mapeando o comportamento variável desejado. As partes que não variam em uma família de aplicações são denominadas pontos-fixos (em inglês, *frozen spots*). Os pontos-fixos ditam a arquitetura (Braga, 2002b; Gamma et al., 1995) e caracterizam partes comuns de todas as aplicações do domínio abordado (Cunningham et al., 2005). Geralmente, são representados por classes e métodos concretos. Os locais que devem proporcionar flexibilidade para introdução de comportamentos específicos são conhecidos como pontos-variáveis (*hot spots*) ou ganchos (*hooks*). Normalmente são representados como classes abstratas com métodos abstratos ou interfaces (Cunningham et al., 2005).

Durante a reutilização de frameworks ocorre a "inversão de controle" (Johnson, 1997b; Gamma et al., 1995; Fayad e Schmidt, 1997), também conhecido como "Princípio de Hollywood" (Larman, 2004). Quando bibliotecas de classes são reutilizadas, o fluxo de controle fica por conta do programador (desenvolvedor de aplicações) que deve invocar os métodos, isto é, codifica-se na aplicação as chamadas dos métodos. Com frameworks, o desenvolvedor de aplicações fornece o código que deve ser chamado pelo framework. Dessa forma, o framework fica responsável pelo fluxo de controle do programa, reduzindo a quantidade de decisões que devem ser tomadas pelo desenvolvedor de aplicações (Gamma et al., 1995).

Outros dois conceitos importantes, relacionados aos frameworks, são métodos gabarito (*template methods*) e métodos gancho (*hook methods*). Métodos gabarito são as partes comuns a uma família de aplicações, representados por métodos concretos (Cunningham et al., 2004, 2006). Métodos gancho são locais onde há divergência entre as aplicações pertencentes ao domínio subjacente, geralmente são métodos abstratos. Os métodos gabarito invocam o comportamento variável, definido nos métodos gancho.

2.5.1 Categorias de Frameworks

As duas técnicas mais comuns para reutilizar funcionalidade em sistemas orientados a objetos são: herança e composição. Reúso por meio de criação de subclasses é conhecido como reúso caixa-branca (*white-box reuse*), já na composição, o reúso ocorre por meio da "montagem" de objetos, esse tipo de reúso é conhecido como reúso caixa-preta (*black-box reuse*), porque detalhes internos dos objetos sendo compostos não são visíveis (Gamma et al., 1995).

Os frameworks enquadram-se nas seguintes categorias: frameworks caixa-branca (*white-box frameworks*), frameworks caixa-preta (*black-box frameworks*) ou frameworks caixa-cinza (*gray-box frameworks*) (Fayad e Schmidt, 1997; Johnson, 1997a).

Em frameworks caixa-branca o reúso ocorre por herança, ou seja, o usuário deve criar subclasses das classes abstratas do framework (Braga, 2002b). Projetistas de

aplicações precisam conhecer a arquitetura desse tipo de framework a fim de ajustá-lo para uma aplicação concreta. É importante ressaltar que uma desvantagem desse tipo de framework é que o usuário final precisa de um sólido conhecimento sobre a sua arquitetura, que implica em uma curva íngreme de aprendizado e aumenta a probabilidade de erros (Parsons et al., 1999; Braga, 2002b). Em frameworks caixa-preta, o reúso é por composição: a aplicação desejada é obtida pela combinação de várias classes concretas pertencentes ao framework. O usuário não necessita conhecer a arquitetura interna do framework, somente os pontos variáveis. Assim, a utilização de frameworks caixa-preta é mais fácil quando comparada ao uso de frameworks caixa-branca (Parsons et al., 1999). Frameworks caixa-cinza, entretanto, consistem de uma mistura de caixa-preta e caixa-branca, o reúso é obtido por herança, interfaces de definição e ligação dinâmica (Braga, 2002b). Parsons et al. (1999) ressaltam que na prática existem poucos frameworks puramente caixa-branca ou caixa-preta e os pontos variáveis podem ser desenvolvidos usando tanto a abordagem de caixa-branca quanto a de caixa-preta.

Segundo Johnson (1997b), um framework caixa-branca é mais fácil de projetar, dado que não há necessidade de prever todas as alternativas possíveis de implementação, como ocorre nos frameworks caixa-preta. Frameworks caixa-preta são mais fáceis de usar, visto que não é necessário fornecer uma implementação completa, basta selecioná-la.

2.5.2 Frameworks e Outras Formas de Reúso

Gamma et al. (1995), destacam as seguintes diferenças entre frameworks e padrões de projeto: frameworks são menos abstratos que padrões de projeto; padrões de projeto são "elementos arquiteturais" menores e são menos especializados que frameworks, porque os frameworks são concebidos para um domínio de aplicação particular, enquanto que os padrões de projeto podem ser usados em quase qualquer espécie de aplicação. Usualmente, padrões de projeto são utilizados na implementação de frameworks, dado que eles propõem soluções que introduzem a flexibilidade necessária nos pontos-variáveis. Padrões de projeto podem auxiliar na compreensão do funcionamento e da estrutura interna dos frameworks, pois cada padrão carrega consigo sua semântica.

Linguagens de padrões e frameworks são específicos de domínio. Johnson (1992) afirma que a documentação de frameworks exige uma descrição do propósito do framework, deve também detalhar a sua utilização e descrever detalhes de projeto. Linguagens de padrões podem ser utilizadas para documentar frameworks e também podem atuar como "guia" durante a instanciação do framework. Nesse sentido, a linguagem de padrões comunica os padrões e subpadrões, bem como o papel de cada classe dos padrões (Brugali e Sycara, 2000; Johnson, 1992). A disponibilidade de

uma linguagem de padrões e do respectivo framework – "inspirado" na linguagem de padrões – possibilitam que aplicações não necessitem ser projetadas "partindo do zero", já que o framework oferece implementações de cada um dos padrões da linguagem (Braga, 2002b; Brugali e Sycara, 2000).

Frameworks são mais abstratos que a maioria dos sistemas de software, implicando que os interessados devem compreender complexas hierarquias de classes e colaborações entre os objetos, somente assim, podem reutilizá-los eficientemente. Ou seja, diferentemente das bibliotecas de classes, para se reutilizar frameworks (instanciação), são necessários conhecimentos mais consistentes sobre sua implementação. Frameworks exigem uma documentação mais elaborada e mais treino por parte dos interessados na sua instanciação (Johnson, 1997b; Srinivasan, 1999; Johnson, 1992).

Frameworks são implementados usando linguagens orientadas a objetos, assim tiram proveito de três características da programação orientada a objetos (Johnson, 1997a): abstração de dados, polimorfismo e herança. De forma geral, as vantagens proporcionadas aos desenvolvedores pelos frameworks derivam da modularidade, reusabilidade e inversão de controle (Fayad e Schmidt, 1997).

2.6 Refatoração

A refatoração (*refactoring*), de acordo com Fowler et al. (1999), surgiu na comunidade de programadores Smalltalk. É o processo de alterar um sistema de software, melhorando a sua estrutura interna, de forma que o comportamento externo[2] do código não seja alterado. A refatoração consiste, entre outras coisas, em redistribuir as classes, variáveis e métodos na hierarquia de classes, com o objetivo de facilitar futuras atividades de desenvolvimento ou de manutenção (Opdyke, 1992; Fowler et al., 1999; Mens e Tourwe, 2004; Demeyer et al., 2004).

No contexto da reengenharia, a refatoração é empregada para converter o código legado em um código mais modular e estruturado ou até com o objetivo de migrá-lo para uma nova linguagem de programação (Mens e Tourwe, 2004). Neste trabalho, ela é usada com o objetivo de apoiar o projeto iterativo do framework. Geralmente, o desenvolvimento de um framework envolve várias etapas e mudanças estruturais (Opdyke, 1992). A refatoração, com transformações que conservam o comportamento externo, atua melhorando o projeto existente, a compreensão e facilitando o reúso.

À medida que o código é alterado, gradualmente se torna mais difícil entendê-lo. Normalmente, uma aplicação mal projetada necessita de mais código; isso porque faz a mesma coisa em vários lugares; dessa forma, para se aprimorar o projeto deve-se eliminar o código duplicado (Fowler et al., 1999). Eliminar a duplicação é um prin-

[2]Também denominado comportamento observável (Beck, 2002).

cípio simples que guia a bons projetos (Fowler, 2001). Os profissionais responsáveis pela manutenção de sistemas vêem claros benefícios na eliminação de código duplicado (Demeyer, 2005), pois a compreensão do funcionamento interno do software fica facilitada. No contexto de construção e evolução de frameworks, a refatoração colabora para tornar o projeto mais flexível. Cortes et al. (2003) propõem uma abordagem para apoiar a evolução de frameworks, usando refatoração e regras de unificação (*unification rules*).

A maioria das refatorações introduz indireção, em outras palavras, tende a dividir objetos de maior granularidade em objetos menores e métodos longos são transformados em vários métodos menores (Fowler et al., 1999). Usualmente, os programadores não apreciam introduzir mais indireção aos seus programas, alegando que isso pode resultar em programas mais lentos (Demeyer, 2005). Não obstante, a indireção tem algumas vantagens (Fowler et al., 1999):

Explicar intenção e implementação separadamente: o nome de cada método, variável ou classe fornece a oportunidade de explicar sua intenção. A implementação de classes ou métodos explicam como a intenção é realizada.

Isolar a mudança: facilita a introdução de funcionalidade.

Codificar a lógica condicional: alterando a lógica condicional por mensagens polimórficas evita-se duplicação de código e aumenta-se a flexibilidade.

Além disso, segundo Demeyer (2005), considerando a tecnologia atual dos processadores e compiladores, funções virtuais (linguagem C++) são tão rápidas quanto os trechos de código equivalentes em lógica condicional.

A maioria das *Integrated Development Environments* (IDE's) automatizam algumas refatorações, mas nenhuma delas auxilia na determinação de qual trecho de código deve ser refatorado e nem quais refatorações devem ser aplicadas. O desenvolvedor determina onde o código deve ser refatorado por meio da análise de *"bad smells"* (Fowler et al., 1999). Porém, Mäntylä et al. (2004) consideram a avaliação do código utilizando o conceito de *"bad smells"* subjetiva, variando de acordo com a experiência do desenvolvedor. Entretanto, de acordo com Fowler et al. (1999), nenhum critério exato pode ser utilizado para determinar quando o código deve ser refatorado ou não.

2.6.1 Testes e Refatoração

É consenso que, a fim de realizar atividades de refatoração com segurança, é indispensável a presença de testes automatizados (Fowler et al., 1999; Demeyer et al., 2002; Beck, 2002). Todavia, é importante observar que não é somente o código relacionado à funcionalidade que pode passar por refatorações, o código relacionado aos

testes também deve ser refatorado (Meszaros, 2007). Porém, a refatoração do código dos testes envolve algumas complicações, pois, nesse caso, não há testes que forneçam *feedback* sobre a introdução de erros (Deursen et al., 2001; Meszaros, 2007; Guerra e Fernandes, 2007).

2.7 Aspectos

A programação orientada a objetos oferece uma maneira de encapsular os dados e o comportamento em uma única entidade e reutilização de código por meio do relacionamento de herança e polimorfismo. Mesmo com todos esses benefícios, alguns interesses (*concerns*) não podem ser adequadamente modularizados.

Um interesse é um requisito que deve ser considerado com o propósito de satisfazer metas do sistema. Há duas categorias de interesses (Laddad, 2003) descritas na Tabela 2.2. Exemplos comuns de interesses transversais que afetam vários subsistemas são: gerenciamento de transações (Laddad, 2003), manipulação de exceções (Lippert e Lopes, 2000; Filho et al., 2006), persistência de dados (Soares et al., 2002; Rashid e Chitchyan, 2003) e segurança (Murphy e Schwanninger, 2006).

Tabela 2.2: Categorias de interesses

CATEGORIA	DESCRIÇÃO
Interesses fundamentais ou interesses do negócio (*core concerns*)	Captam a funcionalidade central de um módulo. Formam a lógica de negócio.
Interesses transversais (*crosscutting concerns*)	Captam requisitos de nível de sistema, aqueles que permeiam vários módulos.

A programação orientada a objetos não é conveniente para tratar de interesses transversais, pois, geralmente, são implementados como entidades ou um conjunto de entidades, com dados e comportamentos pertinentes. Entretanto, chamadas explícitas dos serviços devem ser feitas, causando entrelaçamento de código no lado do cliente. Por exemplo, uma entidade β, que necessite de autenticação e persistência, precisa invocar o comportamento das respectivas entidades, responsáveis por mapear tal comportamento. Assim, o código em β fica entrelaçado com o código de outros módulos, rompendo assim a independência entre as abstrações (módulos). De forma simples, interesses transversais tendem a se "misturar" na implementação dos interesses de negócio, como ilustrado na Figura 2.1.

Metodologias como programação generativa (*generative programming*), metaprogramação (*meta-programming*), programação orientada a sujeitos (*subject-oriented programming*) e programação adaptativa (*adaptive programming*) surgiram como possí-

veis abordagens para modularização de interesses transversais (Laddad, 2003). Também com o objetivo de separar, claramente, a implementação de interesses transversais surgiu a programação orientada a aspectos (*aspect oriented programming*) ou POA[3]. Trata-se de um progresso para modularização de interesses transversais, baseada em um novo elemento de programação, os aspectos (*aspects*) (Elrad et al., 2001; Miller, 2001).

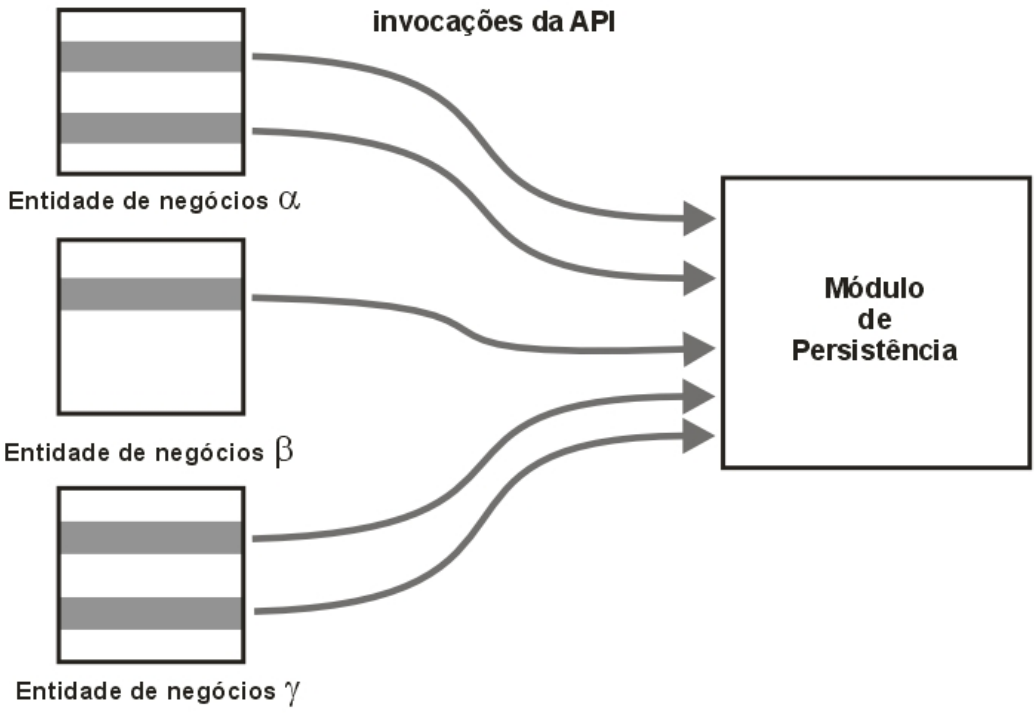


Figura 2.1: Implementação de interesse transversal utilizando programação orientada a objetos, adaptado de Laddad (2003)

Utilizando a programação orientada a aspectos, programadores estão aptos a mapear esses interesses, que permeiam várias partes do sistema, em um único bloco de código que é aplicado de maneira uniforme através do programa. A POA é construída sobre tecnologias existentes. Semelhante às classes, aspectos são entidades "tipadas" que contém funcionalidade, mas diferentemente delas, aspectos têm a intenção de captar interesses transversais que deverão ser injetados em outros tipos (Viega e Voas, 2000). Aspectos, conceitualmente, são separados de objetos; aspectos podem observar objetos e reagir de acordo com o comportamento deles (Viega e Voas, 2000). A implementação do interesse ilustrado na Figura 2.1 (persistência) utilizando POA é apresentada na Figura 2.2.

[3]POA é acrônimo de Programação Orientada a Aspectos.

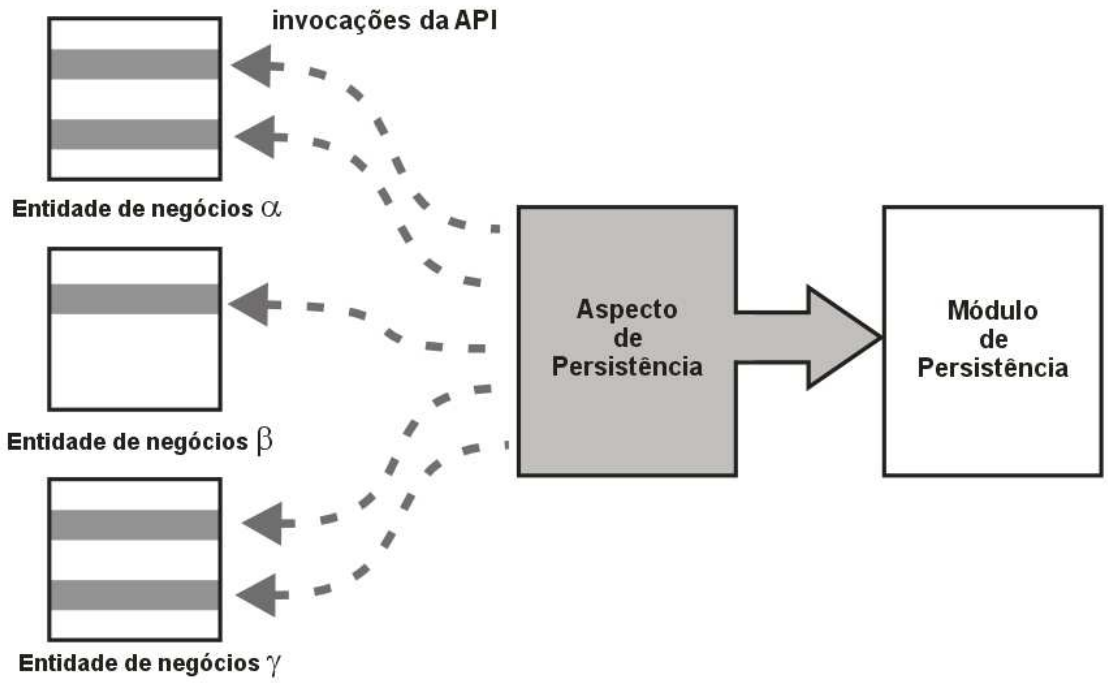


Figura 2.2: Implementação de interesse transversal utilizando POA, adaptado de Laddad (2003)

Necessita-se definir como os aspectos devem entrecortar (*crosscut*) o sistema, pois os aspectos não podem entrecortar os objetos arbitrariamente (Viega e Voas, 2000). A funcionalidade suplementar fornecida pelos aspectos pode ser introduzida somente em pontos bem definidos, tais pontos são conhecidos como pontos de junção (*join points*) (Viega e Voas, 2000; Murphy e Schwanninger, 2006). Pontos de junção descrevem os "ganchos" onde aprimoramentos devem ser acrescentados (Elrad et al., 2001). Exemplos de pontos de junção são: invocação de métodos, instanciação de objetos, execução de um método, o evento de atribuir um valor a determinada propriedade de um objeto (variável de instância) e o evento de manipular uma exceção. Também deve haver um meio de identificar os pontos de junção (Elrad et al., 2001). Em algumas linguagens orientadas a aspectos, os pontos de junção são representados por um conjunto de pontos de junção (*pointcut*). Após o programador ter definido os pontos nos quais o aspecto irá adicionar comportamento, esse deve ser definido. Comportamentos transversais (*advices*), também conhecidos como adendos, consistem de um conjunto de pontos de junção e um corpo, que contém o comportamento que deverá ser adicionado cada vez que um ponto de junção do conjunto de pontos de junção ocorrer. Aspectos encapsulam especificações de ponto de junção e os comportamentos transversais que devem ser adicionados aos respectivos pontos de junção (Elrad et al., 2001). Outro elemento essencial da POA é o combinador (*aspect*

weaver). O combinador é uma entidade semelhante a um compilador. Sua função é "compor" o sistema final por meio da composição dos módulos de negócio e dos módulos transversais. Esse processo é denominado combinação (*weaving*) (Laddad, 2003).

A POA não têm o intuito de substituir outras tecnologias como a programação orientada a objetos e a programação procedural, por exemplo. Pelo contrário, a POA foi construída sobre as tecnologias existentes (Elrad et al., 2001) para suprir algumas deficiências dessas tecnologias (Miller, 2001).

Várias pesquisas que avaliam a aplicabilidade e vantagens da programação orientada a aspectos têm sido conduzidas. Hannemann e Kiczales (2002) afirmam que alguns padrões GoF implementados utilizando a POA apresentaram melhorias em relação a modularidade, Hanenberg (2003) propõe idiomas para construção de frameworks, também utilizando a POA. Pesquisas sobre refatorações com o intuito de mudar de paradigma – da programação orientada a objetos para POA – têm sido desenvolvidas por Monteiro e Fernandes (2005a) e Kulesza et al. (2005), entre outros. Monteiro e Fernandes (2005b) propõem um catálogo de refatorações.

Várias linguagens já possuem extensão para programação orientada a aspectos, por exemplo: C (Coady et al., 2001), C++ (Lohmann et al., 2004; Mahrenholz et al., 2002), o ambiente Squeak/Smalltalk (Hirschfeld, 2001) e Java™ (Laddad, 2003), entre outras. Nesta Seção é apresentada uma extensão da linguagem Java™que introduz orientação a aspectos, denominada AspectJ. Assim, qualquer programa válido em Java™é válido em AspectJ, ou seja, AspectJ desfruta de todos os benefícios proporcionados pela plataforma Java™ (Laddad, 2003). Utilizando AspectJ, os interesses de negócio são implementados usando a linguagem Java™"padrão" e os interesses transversais são implementados com o auxílio de alguns mecanismos da AspectJ.

A AspectJ suporta a implementação de interesses transversais dinâmicos (*dynamic crosscutting*) e interesses transversais estáticos (*static crosscutting*). O primeiro possibilita que certos "comportamentos" sejam adicionados em pontos bem definidos dos módulos que implementam os interesses de negócio. Mais especificamente, adiciona implementações ou substitui as existentes, alterando o fluxo de controle do programa transversalmente. O segundo permite alterar a estrutura dos módulos. Por exemplo, permite adicionar métodos e dados em uma classe.

As linguagens de programação orientadas a aspectos têm três elementos essenciais: os pontos de junção, um meio de especificar o conjunto de pontos de junção e também uma forma de especificar o comportamento que deve ser adicionado nos pontos de junção, ou seja, os interesses transversais (Kiczales et al., 2001).

Os pontos de junção são pontos "identificáveis" durante a execução do programa, como invocações de métodos, acesso à variáveis de instância e invocação de construtores. Pontos de junção são os locais onde os comportamentos transversais são

combinados (Laddad, 2003).

Conjunto de pontos de junção ou, simplesmente, conjunto de junção selecionam pontos de junção e obtêm informações sobre seus respectivos contextos. Um conjunto de junção pode ser derivado da combinação de outros conjuntos de junção. Tal combinação pode empregar os operadores lógicos e (em inglês, *and*; sintaxe: **&&**), ou (em inglês, *or*; sintaxe: **||**) e negação (em inglês, *not*; sintaxe: **!**). Comportamento transversal define o código que deve ser acrescentado quando um determinado conjunto de junção é alcançado. A unidade que encapsula conjuntos de junção e comportamentos transversais é denominada aspecto.

2.8 Considerações Finais

Este Capítulo apresentou alguns dos conceitos necessários para a realização deste trabalho. O conceito de padrões foi citado, principalmente para padrões de projeto e de reengenharia, pois são mais relevantes no contexto deste trabalho que tem por objetivo a reengenharia de um framework orientado a objetos. Linguagens de padrões e frameworks foram apresentados, bem como o relacionamento entre eles, já que o framework que passará pelo processo de reengenharia foi construído com base em uma linguagem de padrões de análise.

O Capítulo também apresentou a programação orientada a aspectos, que tem o objetivo de modularizar adequadamente interesses transversais. A linguagem AspectJ, uma extensão orientada a aspectos da linguagem Java^{TM}foi apresentada, dado que ela também foi utilizada na implementação do framework GRENJ.

O Capítulo seguinte apresenta o framework GREN, no qual o trabalho tem enfoque, e a linguagem de padrões em que o mesmo foi baseado, descrevendo informações necessárias para compreensão do processo de reengenharia apresentado posteriormente.

CAPÍTULO
3

GRN e GREN

3.1 Considerações Iniciais

Frameworks de software orientados a objetos são difíceis de projetar, pois, normalmente, são sistemas de software que requerem maior flexibilidade e exigem conhecimento do seu domínio. Abordagens para facilitar a criação, instanciação e documentação de frameworks são propostas por Brugali e Sycara (2000) e Braga e Masiero (2002a,c), tais abordagens são baseadas na utilização de linguagens de padrões. Neste Capítulo é apresentada a Linguagem de Padrões para Gestão de Recursos de Negócios (GRN) na qual o trabalho é enfocado, bem como uma sucinta descrição de seus padrões. A linguagem de padrões subjacente foi utilizada na implementação de um framework (GREN – Gestão de REcursos de Negócios), que também é apresentado neste Capítulo. Para facilitar ainda mais o reúso do framework desenvolvido, possibilitando que aplicações sejam geradas com base no conhecimento dos padrões da linguagem de padrões GRN, um *wizard* foi desenvolvido para automatizar a instanciação de aplicações por meio do GREN (Braga, 2002b). Essa ferramenta, denominada GREN-Wizard (Braga, 2002b), também é descrita neste Capítulo.

O Capítulo está organizado da seguinte forma: a Seção 3.2 apresenta informações relevantes sobre a linguagem de padrões GRN, na Seção 3.3 é descrito o framework GREN implementado com base na linguagem de padrões GRN e a Seção 3.4 descreve o *wizard* desenvolvido. Por fim, na Seção 3.5 são apresentadas as considerações finais deste Capítulo.

3.2 Linguagem de Padrões Para Gestão de Recursos de Negócios

A Linguagem de Padrões para Gestão de Recursos de Negócio (GRN) (Braga et al., 1999), é resultado da experiência prática no desenvolvimento de sistemas de gestão de recursos de negócio para pequenas e médias empresas (Braga, 2002b). A GRN se situa no nível de análise, é composta de quinze padrões de análise, sendo alguns extensões ou aplicações de padrões recorrentes propostos na literatura.

A GRN é útil para aplicações nas quais seja necessário registrar:

- Transações de aluguel (bem ou serviço)
- Comercialização (transferência de propriedade ou de bem)
- Manutenção (reparo ou conservação de produto)

Aluguel, no contexto da GRN, compreende-se como a utilização temporária de um bem ou serviço; comercialização compreende-se como a transferência de propriedade ou bem; manutenção enfoca o reparo ou conservação de produtos, utilizando mão-de-obra e peças para execução (Braga et al., 1999). Os padrões da GRN são agrupados de acordo com seus respectivos propósitos e são denotados usando a *Unified Modeling Language* (UML) (Fowler, 2005). A solução proposta por cada padrão é ilustrada com um diagrama de classes. Elementos adicionais são usados nesses diagramas a fim de facilitar a distinção entre alguns tipos de operações. Por exemplo, o marcador *"?"* é usado para denotar operações que modificam o estado interno do sistema e o prefixo *"!"* para as operações de saída, ou seja, operações que geram saídas do sistema sem alterar o estado interno. O prefixo **"*"** é empregado nos métodos em que a mensagem é enviada para uma coleção de objetos (*container*), ao invés de uma única instância. Além disso, o nome dos padrões é escrito utilizando capitalização de maiúsculas (Braga et al., 1999).

A Figura 3.1 ilustra o grafo de fluxo de aplicação dos padrões. Tal grafo estabelece uma possível ordem de aplicação dos padrões que compõe a linguagem, ilustrando quais são os principais padrões – linha reforçada – e os padrões opcionais da linguagem (Braga, 2002b).

Os padrões da GRN são agrupados por propósito.

- Relacionados com à identificação do recurso de negócio (Seção 3.2.1);
- Relacionado às transações com os recursos (Seção 3.2.2);
- Relacionados aos detalhes das transações (Seção 3.2.3).

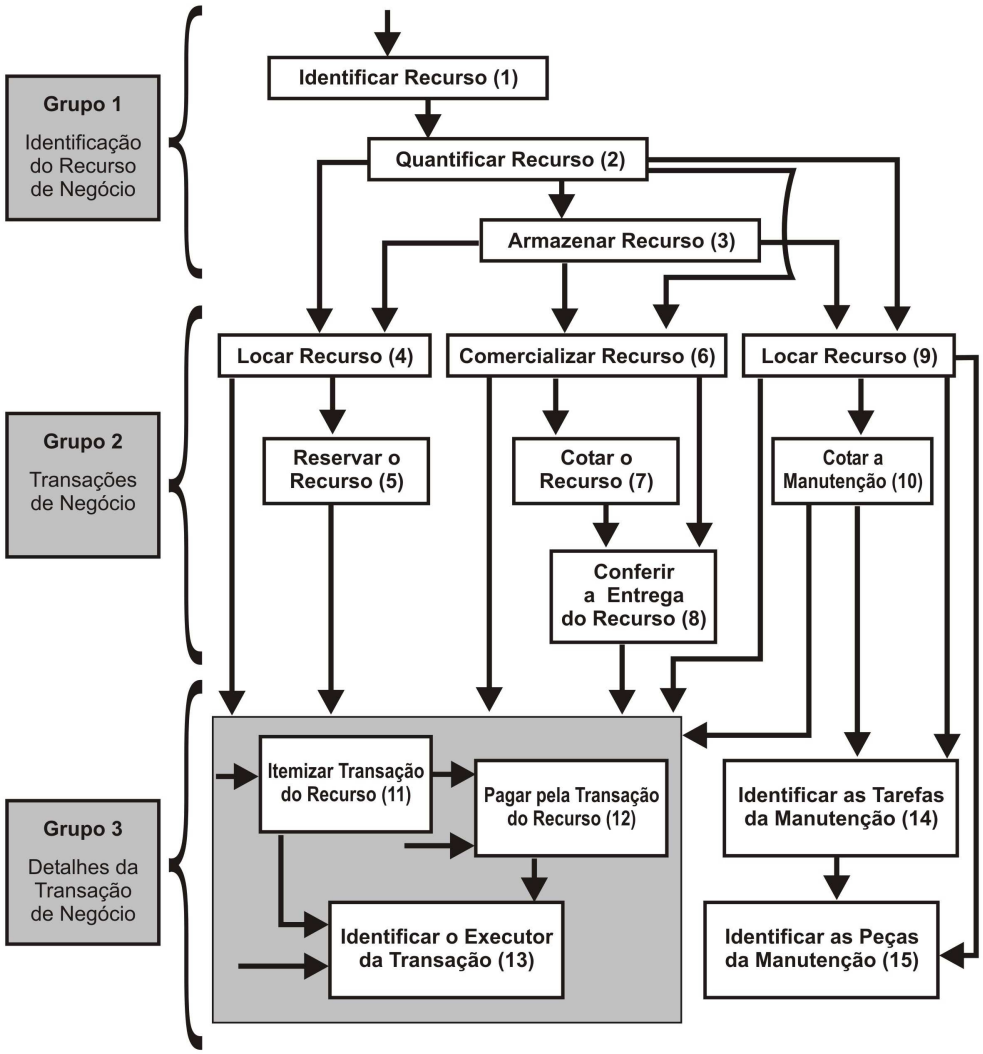


Figura 3.1: Grafo de fluxo de aplicação da GRN (Braga, 2002b)

3.2.1 Grupo Relacionado com a Identificação do Recurso de Negócio

Esse grupo é formado por três padrões: IDENTIFICAR O RECURSO, QUANTIFICAR O RECURSO e ARMAZENAR O RECURSO.

O padrão IDENTIFICAR O RECURSO, ilustrado na Figura 3.2, propõe uma solução para representar os recursos de negócio envolvidos nas transações e processados pelo sistema (Braga, 2002a).

QUANTIFICAR O RECURSO aborda a forma de quantificação do recurso. Em algumas aplicações é necessário controle sobre instâncias específicas do recurso, que são negociadas separadamente. Em outras aplicações os recursos são tratados em lotes e existem ainda aplicações em que o recurso é único. QUANTIFICAR O RECURSO é

formado por quatro subpadrões, cada um deles voltados para uma forma de quantificação:

RECURSO SIMPLES: quando o recurso é único.

RECURSO MENSURÁVEL: situações onde o recurso é tratado em quantidades específicas.

RECURSO INSTANCIÁVEL: quando é necessário distinguir entre instâncias dos recursos.

RECURSO EM LOTES: quando o recurso é tratado em lotes.

No mesmo grupo encontra-se o padrão ARMAZENAR O RECURSO. Em muitas aplicações é essencial controlar o armazenamento dos recursos, facilitando a recuperação quando necessária. ARMAZENAR O RECURSO trata das considerações envolvidas.

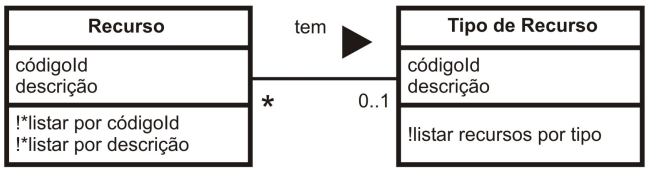


Figura 3.2: Padrão IDENTIFICAR O RECURSO; extraído de (Braga, 2002a)

3.2.2 Grupo Relacionado às Transações com os Recursos

Este grupo, composto por sete padrões, contêm os relacionados à manipulação dos recursos de negócio (Braga, 2002a): LOCAR O RECURSO, RESERVAR O RECURSO, COMERCIALIZAR O RECURSO, COTAR O RECURSO, CONFERIR A ENTREGA DO RECURSO, MANTER O RECURSO e COTAR A MANUTENÇÃO.

O padrão LOCAR O RECURSO cuida das questões relacionadas aos aluguéis de recursos. RESERVAR O RECURSO destina-se ao gerenciamento de reservas antes da locação. COMERCIALIZAR O RECURSO resolve o problema de como uma aplicação deve gerenciar o comércio de recursos e pode ser visualizado na Figura 3.3.

O padrão COTAR O RECURSO trata o gerenciamento das cotações, realizadas antes do comércio efetivo do recurso.

O padrão CONFERIR A ENTREGA DO RECURSO propõe uma solução para aplicações que necessitam conferir a entrega em relação à comercialização (Braga, 2002a). MANTER O RECURSO é o padrão que aborda a manutenção ou conserto de recursos de negócio, como pode ser visto na Figura 3.4.

Por fim, no grupo relacionado às transações com os recursos, tem-se o padrão COTAR A MANUTENÇÃO, que oferecer uma estimativa de custo aos clientes, antes que a manutenção seja autorizada.

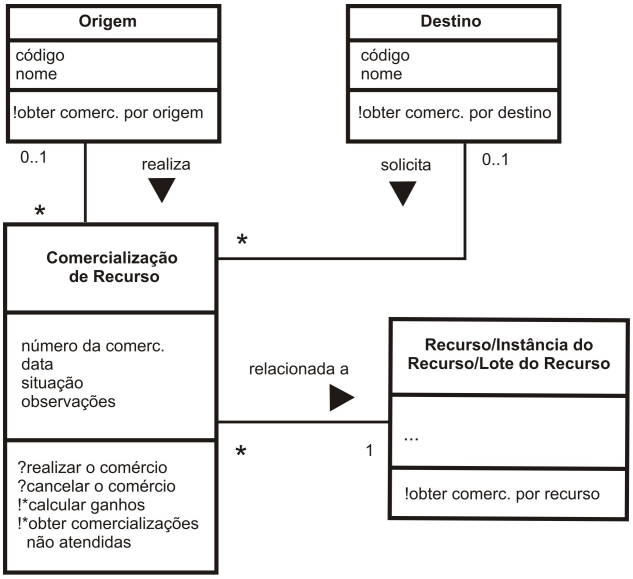


Figura 3.3: Padrão COMERCIALIZAR O RECURSO; extraído de (Braga, 2002a)

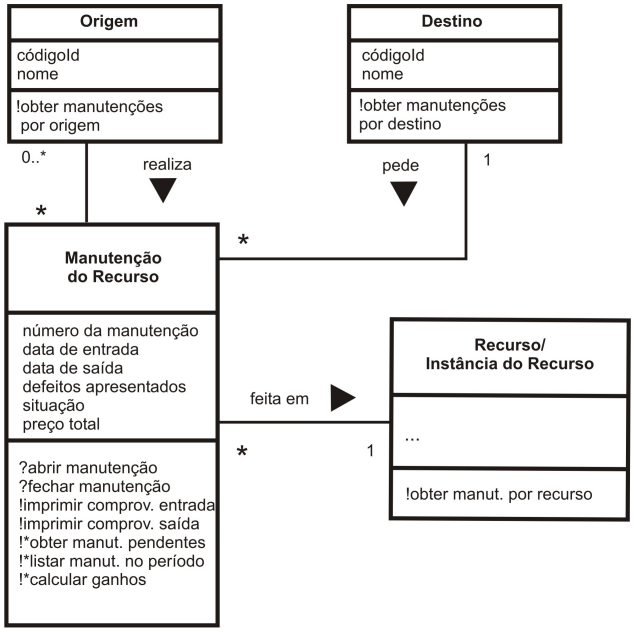


Figura 3.4: Padrão MANTER O RECURSO; extraído de (Braga, 2002a)

3.2.3 Grupo Relacionado com Detalhes das Transações

Este grupo contêm cinco padrões que cuidam dos detalhes envolvidos nas transações. Os três primeiros padrões: ITEMIZAR TRANSAÇÃO DO RECURSO, PAGAR PELA TRANSAÇÃO e IDENTIFICAR EXECUTOR DA TRANSAÇÃO são aplicáveis à quaisquer transações da Seção 3.2.2. Entretanto, os outros dois padrões do grupo, IDENTIFICAR AS TAREFAS DA MANUTENÇÃO e IDENTIFICAR AS PEÇAS DA MANUTENÇÃO, são aplicáveis às transações contidas nos padrões MANTER O RECURSO e COTAR A MANUTENÇÃO.

Há certas situações em que mais de um recurso deve ser gerenciado em uma única transação. Por exemplo, no caso de uma locadora, provavelmente um cliente aluga mais de um DVD por visita. O padrão ITEMIZAR TRANSAÇÃO DO RECURSO traz uma solução para sistemas que precisam desse tipo de gerenciamento. A Figura 3.5 ilustra o padrão ITEMIZAR TRANSAÇÃO DO RECURSO.

PAGAR PELA TRANSAÇÃO DO RECURSO trata das despesas envolvidas na maioria das transações. IDENTIFICAR O EXECUTOR DA TRANSAÇÃO trata da identificação da pessoa ou entidade responsável pela execução da transação.

Tarefas de manutenção geralmente envolvem trabalhos de mão-de-obra de diferentes pessoas e substituição de peças danificadas. Em casos assim, deve-se especificar as peças envolvidas na manutenção. Para tal propósito deve-se utilizar o padrão IDENTIFICAR AS TAREFAS DA MANUTENÇÃO.

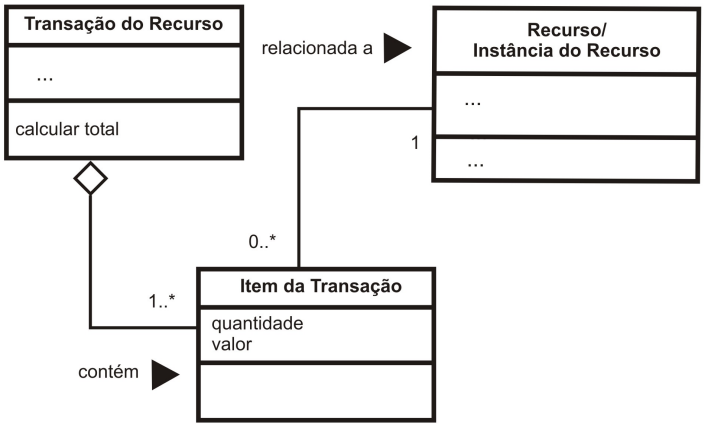


Figura 3.5: Padrão ITEMIZAR TRANSAÇÃO DO RECURSO; extraído de (Braga, 2002a)

3.3 GREN

O GREN (Gestão de REcursos de Negócios) é um framework caixa-branca, dado que seu reúso acontece por meio de herança. Foi desenvolvido tendo a GRN como base e facilita a criação de aplicações no domínio de gestão de recursos de negócios.

O GREN foi implementado na linguagem Smalltalk[1], com banco de dados relacional MySQL (MySQL, 2007). A arquitetura do GREN é composta de três camadas: de persistência, de negócios e a de interface gráfica com o usuário. A camada de persistência contêm as classes relacionadas à conexão com o banco e persistência dos objetos de forma geral. A camada de negócios possui implementações dos padrões da GRN (Braga, 2002b). Na camada de interface gráfica com o usuário encontram-se os formulários, janelas, menus, caixas de diálogos, entre outros responsáveis pela interação com o usuário e pela entrada e exibição dos dados.

Essa separação em camadas proporciona que aplicações sejam instanciadas de três modos (Braga, 2002b): instanciadas por meio de herança a partir da camada de interface gráfica do GREN, por meio da utilização de outra camada de interface gráfica reutilizando a camada de negócios e camadas abaixo dela ou por meio do GREN-Wizard (apresentado na Seção 3.4). O processo de instanciação, utilizando a linguagem de padrões, é abordado em detalhes por Braga e Masiero (2002c); usando o GREN-Wizard por Braga (2002b).

Frameworks, como mencionado, proporcionam o reúso de grandes estruturas de código, comuns a maioria das aplicações do domínio ao qual eles pertencem. Todavia, existem divergências entre as aplicações do domínio. Tais divergências precisam ser identificadas e o framework precisa propiciar a flexibilidade necessária para acomodar as variabilidades. A GRN foi a principal fonte para identificação dos pontos-variáveis do GREN e padrões de projeto foram empregados para alcançar a flexibilidade almejada. Por exemplo, o padrão QUANTIFICAR O RECURSO (Seção 3.2.1) aborda quatro maneiras distintas de se realizar a quantificação do recurso e a flexibilidade necessária foi alcançada com a utilização do padrão de projeto *Strategy* (Gamma et al., 1995), encapsulando o comportamento variável (Braga, 2002b). A camada de persistência é implementada utilizando o padrão *Persistence Layer* e a camada de negócios se comunica com a de persistência para persistir um objeto.

A GRN foi utilizada para implementar o GREN de forma gradual e seqüencial, dado que cada padrão pode ser considerado como uma unidade funcional (Braga e Masiero, 2002a; Braga, 2002b). A implementação de cada padrão deu origem a uma parcela da funcionalidade implementada pelo framework GREN.

A elaboração da documentação do GREN originou o "*Cookbook* do GREN", que contém algoritmos para serem utilizados em conjunto com as tabelas criadas para mapear a GRN ao GREN, possibilitando o uso disciplinado dessas tabelas (Braga, 2002b).

[1]Mais especificamente, o ambiente de desenvolvimento foi o VisualWorks Non-Commercial 5i.4.

3.4 GREN-Wizard

Obter o reúso por meio de frameworks não é trivial. Desenvolvedores interessados em reutilizar frameworks caixa-branca precisam compreender complexos relacionamentos entre objetos e hierarquias de classes, deve-se criar subclasses das classes apropriadas e fornecer o comportamento variável. A "chave" para se alcançar o reúso utilizando frameworks reside em uma documentação adequada, facilitando assim a instanciação.

O GREN-Wizard[2] foi criado com o intuito de apoiar o processo de instanciação do framework GREN. O GREN-Wizard "contorna" o problema da complexidade – derivada da instanciação de frameworks – proporcionando que aplicações sejam geradas apenas com o conhecimento da GRN. Mais especificamente, o GREN-Wizard tem sua interface gráfica com o usuário baseada nos padrões da GRN e na ordem em que eles podem ser aplicados. O engenheiro de aplicações informa os padrões que serão usados e quais variantes são mais adequados para a aplicação específica sendo construída (Braga, 2002b). A GRN é utilizada para produzir o modelo de análise, e tal modelo é usado para "alimentar" as tabelas do GREN-Wizard e produzir o código Smalltalk que adapta o GREN à aplicação específica. Adicionalmente, o GREN-Wizard gera as tabelas necessárias na base de dados MySQL.

3.5 Considerações Finais

Este Capítulo apresentou a GRN, destando seu propósito e sua utilidade. Além disso, os quinze padrões de análise que compõem a linguagem de padrões foram descritos. Para familiarizar o leitor uma possível ordem de aplicação desses padrões é a apresentada na Figura 3.1. Apresentou também o GREN, framework caixa-branca criado com base na GRN, e o GREN-Wizard, ferramenta criada com o objetivo de automatizar a instanciação do framework caixa-branca, por meio de um modelo de interação com o desenvolvedor de aplicações possibilitando que aplicações sejam criadas somente com o conhecimento da GRN.

Tanto a GRN quando o "Cookbook do GREN" foram usados como fontes de informações para apoiar o processo de reengenharia do GREN. O Capítulo seguinte apresenta a prática, de fundamental importância no processo iterativo de reengenharia, denominada desenvolvimento guiado por testes (*Test-Driven Development*).

[2]Implementado em Smalltalk, usando o ambiente VisualWorks Non-Commercial 5i.4.

CAPÍTULO
4

Desenvolvimento Guiado Por Testes

4.1 Considerações Iniciais

O desenvolvimento de sistemas requer a presença de testes para aperfeiçoar a sua qualidade e confiabilidade (Myers et al., 2004). Quando o desenvolvimento segue abordagens "tradicionais", a criação de testes é uma das últimas atividades a serem realizadas (Erdogmus et al., 2005). A criação de testes antes da implementação da funcionalidade e a utilização desses nas atividades de análise e projeto auxilia no desenvolvimento de sistemas mais bem estruturados, com menos problemas e mais flexíveis (Beck, 2002; Jeffries e Melnik, 2007). A funcionalidade é testada desde o início da concepção do sistema. Essa prática tornou-se popular com o surgimento de *Extreme Programming* (XP) (Beck, 2000) e é conhecida por vários nomes, entre eles: desenvolvimento guiado por testes (*Test-Driven Development*, TDD), *test-first programming*, *test-driven design* e *test-first design* (Janzen e Saiedian, 2005). Neste trabalho será referida como TDD.

Devido à importância dessa prática no contexto do processo de reengenharia, este Capítulo apresenta os conceitos fundamentais na Seção 4.2. A Seção 4.3 ilustra o ciclo do TDD e a Seção 4.4 descreve os estudos que comparam a abordagem proposta pelo TDD (*test-first*) com a abordagem considerada "tradicional" de desenvolvimento, *test-last*. Na Seção 4.5 o conceito de *test doubles* é apresentado e na Seção 4.6 são descritas as ferramentas que podem ser utilizadas tanto para automatização dos testes quanto para geração dinâmica de *test doubles*. A Seção 4.7 apresenta um exemplo da aplicação do TDD para implementação da seqüência de Fibonacci e as considerações finais são discutidas na Seção 4.8.

4.2 Definição

TDD, assim como refatoração (Fowler et al., 1999), foi concebido em ambiente altamente interativo de desenvolvimento de sistemas utilizando Smalltalk, no qual os desenvolvedores, a partir de um projeto preliminar não muito abrangente, codificavam e testavam o sistema em ciclos bem definidos (Wirfs-Brock, 2007). Juntamente com refatoração e programação em pares, TDD figura entre as práticas fundamentais da *Extreme Programming* (XP) (Beck, 2000; Müller e Hagner, 2002; Janzen e Saiedian, 2005; Melis et al., 2006; Müller e Höfer, 2007). Desenvolvendo de acordo com essa prática, os testes são implementados antes da implementação da funcionalidade pretendida (Martin, 2007).

Os casos de teste criados para exercitar a funcionalidade são agrupados em uma lista de casos de teste ou simplesmente lista de testes (Beck, 2002; Koskela, 2007). Os testes são implementados à medida que se avança no desenvolvimento do sistema e são considerados satisfeitos quando a funcionalidade avaliada por eles estiver corretamente implementada. Desse modo, no TDD tanto o código dos testes quanto o código da funcionalidade avaliada pelos testes, são produzidos em rápidas e pequenas iterações.

Mesmo desenvolvendo dessa maneira, *bad smells* (Fowler et al., 1999) podem ser introduzidos durante a codificação da funcionalidade e dos testes (Deursen et al., 2001; Guerra e Fernandes, 2007). Assim, refatorações são realizadas a fim de aprimorar, incrementalmente, a estrutura do código existente, dado que nenhum projeto preliminar exaustivo é realizado. Beck (2002) e Koskela (2007) afirmam que refatorar é essencial para que novas funções possam ser facilmente introduzidas. Os testes criados atuam como testes de regressão, que podem ser executados durante todo o desenvolvimento e, principalmente, após as refatorações, com o propósito de detectar se, inadvertidamente, houve a introdução de efeitos colaterais (*side effects*) (Meszaros, 2007).

Apesar da denominação, TDD não é uma técnica de teste, é uma técnica de análise e projeto (Beck, 2002; Janzen e Saiedian, 2005; Jeffries e Melnik, 2007). É considerada uma técnica de análise, pois, durante a criação dos testes, decisões são tomadas sobre o que será implementado, definindo o escopo da funcionalidade. É considerada uma técnica de projeto iterativo visto que, conforme o teste é implementado, decisões relacionadas à interface (API) do sistema são tomadas. Por exemplo: nome da classe, nome do método, quantidade de parâmetros, tipos de retorno e possíveis exceções que o método pode lançar. O conceito de implementar o teste como se o código sendo testado existisse é denominado *programar por intenção* (Koskela, 2007).

O TDD é mais comumente empregado no contexto de metodologias ágeis, pois, as decisões sobre a arquitetura do sistema sendo desenvolvido são tomadas iterativa-

mente, à medida que se avança no desenvolvimento.

4.3 Ciclo do TDD

TDD proporciona aos desenvolvedores um ciclo que se repete a cada teste selecionado da lista de casos de teste (Beck, 2002). Conforme pode ser observado na Figura 4.1, inicialmente, seleciona-se um teste da lista de casos de teste e, em seguida, esse teste é implementado. Geralmente, o teste recém-implementado não compila, pois pode utilizar métodos e classes que ainda não foram implementados, ou seja, foram somente concebidos enquanto se implementava o teste *programando por intenção* (Koskela, 2007). Posteriormente, para que o teste compile, implementam-se os elementos (classes ou métodos) referenciados por ele. Em seguida, todos os testes são executados, e o teste recém-criado não é satisfeito, dado que a funcionalidade ainda não foi implementada; as classes necessárias ou métodos foram implementados como *stubs* (Seção 4.5). Após implementada a funcionalidade avaliada pelo teste recém-criado, executam-se, novamente, todos os testes. O teste recém-criado agora é satisfeito e refatorações são realizadas caso exista código duplicado ou outro *bad smell* (Fowler et al., 1999) no código da funcionalidade ou do teste. Executam-se novamente os testes, agora com o propósito de certificar que as refatorações não afetaram o comportamento observável. Esse ciclo se repete até que todos os testes da lista de casos de testes tenham sido implementados.

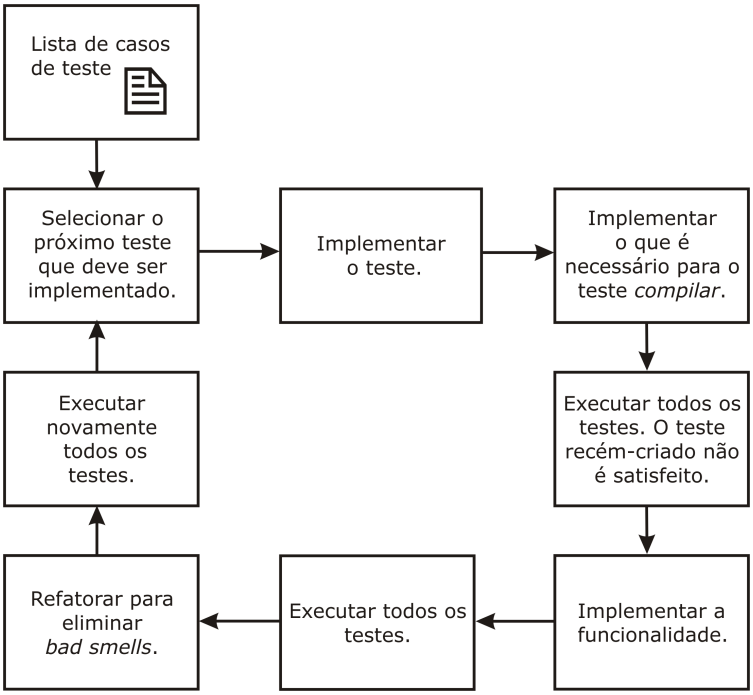


Figura 4.1: Ciclo do TDD

4.4 Contrastando as Abordagens *Test-First* e *Test-Last*

Abordagens em que a criação dos testes ocorre somente após a implementação da funcionalidade são consideradas "tradicionais" e denominadas *test-last* (Erdogmus et al., 2005), abordagens que empregam TDD são denominadas *test-first*. A Figura 4.2 exibe dois diagramas de fluxo contrastando o desenvolvimento utilizando as abordagens *test-last* e TDD (representando abordagens *test-first*). Conforme pode ser observado no TDD avança-se incrementalmente, teste a teste, até que a funcionalidade tenha sido implementada. Em contraste, nas abordagens *test-last*, quando se utiliza modelo cascata ou iterativo, todos os testes são criados ao final da implementação da funcionalidade ou de uma parcela da funcionalidade, respectivamente. A Tabela 4.1 relaciona, sucintamente, as características do desenvolvimento utilizando TDD e do desenvolvimento com abordagens *test-last*.

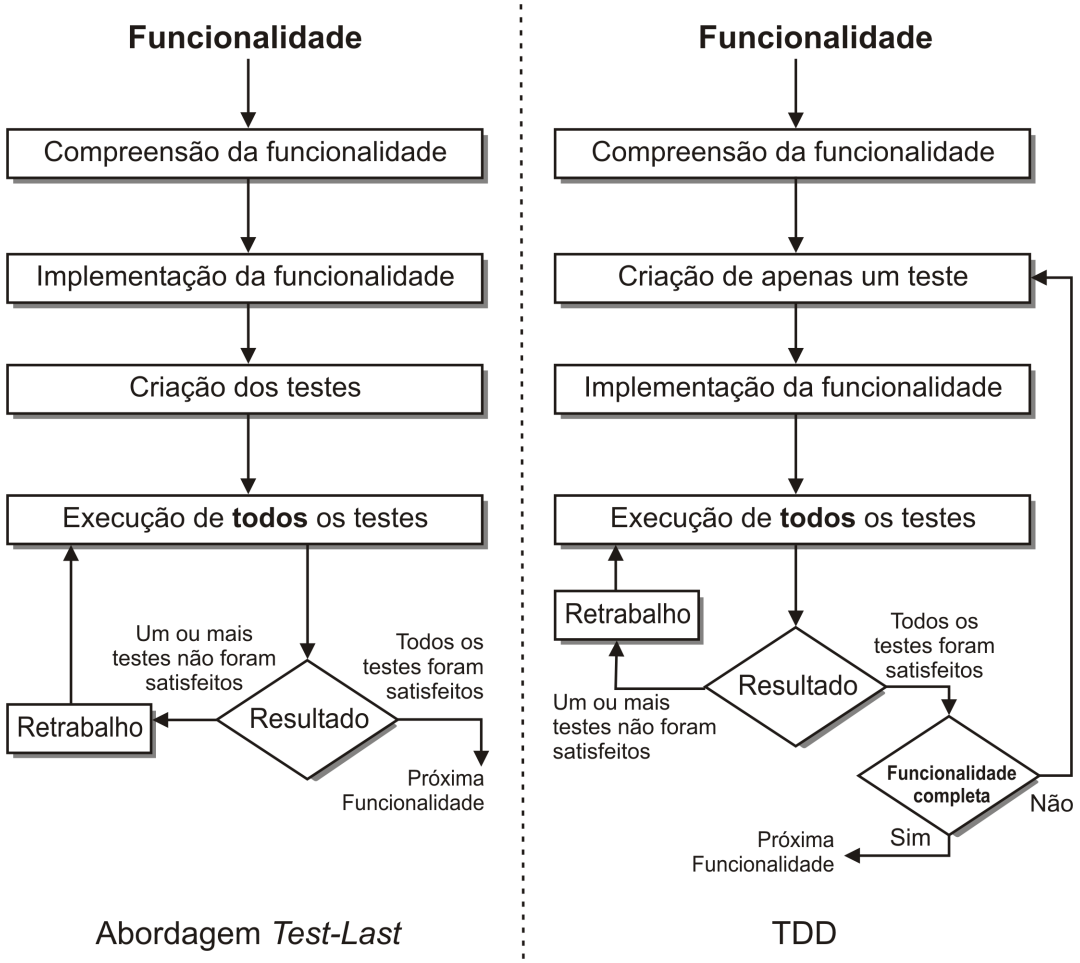


Figura 4.2: Desenvolvimento nas abordagens *test-last* e TDD, respectivamente; adaptado de Erdogmus et al. (2005) e Huang e Holcombe (2006)

Tabela 4.1: Comparação entre as abordagens *test-last* e TDD

Características		
	Test-First	Test-Last
Quem cria os testes?	programador	programador
Quando os testes são criados?	Antes da implementação da funcionalidade.	Depois da implementação da(s) funcionalidade(s).
Quando os testes são executados?	Enquanto o código das funcionalidades é criado. Mais freqüentemente.	Depois que o código das funcionalidades foi criado. Menos freqüentemente.

4.4.1 Estudos Contrastando as duas Abordagens

Alguns estudos analisam os benefícios ou possíveis desvantagens do TDD em relação à abordagem mais tradicional, a *test-last*. Williams et al. (2003) examinam a eficácia do TDD por meio da comparação do número de problemas encontrados em um sistema legado e em um desenvolvido utilizando TDD. Esse estudo foi conduzido na *International Business Machines* (IBM) e nenhum dos desenvolvedores tinha experiência prévia com TDD. Após a criação de um projeto, especificado em UML, todos os testes foram implementados e, conforme se avançava na implementação do sistema, os testes eram satisfeitos. Foi observado que o sistema resultante apresentou quarenta por cento menos problemas que o sistema legado; todavia, nenhuma informação adicional é fornecida quanto à natureza desses problemas. É importante mencionar que a implementação de todos os testes de uma só vez, como realizado no estudo anteriormente mencionado, não é a abordagem proposta por Beck (2002) e Koskela (2007) que conduzem o desenvolvimento teste a teste conforme ilustrado na Figura 4.2.

No experimento realizado por Müller e Hagner (2002) estudantes foram encarregados de concluir a implementação da principal classe de uma biblioteca gráfica que, quando atribuída a esses estudantes, continha somente a declaração da maioria dos métodos e alguns comentários. Todas as outras classes e métodos dessa biblioteca gráfica foram atribuídas aos estudantes devidamente implementadas, ficando a cargo deles somente a compreensão dos mesmos. Os estudantes foram divididos em dois grupos, um utilizou TDD durante a implementação e o outro empregou uma abordagem *test-last*. Ao contrário do esperado pelos condutores do experimento, o grupo utilizando TDD não foi mais produtivo. No entanto, os estudantes desse grupo compreenderam o comportamento dos métodos existentes de forma mais eficiente que o outro grupo. Müller e Hagner (2002) atribuem isso à estratégia de criação constante de testes, pois, à medida que as falhas são indicadas pelos testes, o estudante deve

corrigi-las, compreendendo assim como utilizar o método corretamente.

Erdogmus et al. (2005) conduziram um experimento envolvendo um grupo formado, inicialmente, por trinta e cinco estudantes. No entanto, somente vinte e quatro conseguiram terminar a implementação de um aplicativo para controlar a pontuação de partidas de boliche (*bowling score keeper*); dos quais onze utilizaram TDD durante o desenvolvimento e o restante seguiu uma abordagem *test-last*. De acordo com Erdogmus et al. (2005) os onze estudantes do grupo empregando TDD implementaram mais testes que os participantes do outro grupo. Erdogmus et al. (2005) relatam que os estudantes que implementaram mais testes foram mais produtivos e justificam esse efeito como a seguir: a abordagem de se implementar testes antes da implementação da funcionalidade encoraja uma decomposição mais bem elaborada do sistema e reduz o escopo das tarefas que devem ser realizadas.

Janzen e Saiedian (2006) realizaram um experimento evolvendo três grupos de estudantes. O projeto designado aos grupos foi um sistema de *pretty printer* para *HyperText Markup Language* (HTML). Inicialmente, dois grupos desenvolveriam usando TDD e o grupo restante deveria empregar uma abordagem *test-last* e iterativa. Porém, somente um grupo desenvolveu conforme estabelecido (grupo 1). O outro (grupo 2), que também deveria proceder dessa forma, implementou os testes após a implementação da funcionalidade e, além disso, os testes não foram implementados pelo mesmo estudante que implementou a funcionalidade. Por fim, o grupo que deveria desenvolver de maneira *test-last*, devido a restrições de tempo, não implementou nenhum teste (grupo 3). Foi observado que o grupo 1 implementou o dobro das funções (12) que os grupos 2 e 3 (6 e 5, respectivamente). De acordo com Janzen e Saiedian (2006) os resultados indicam que a implementação dos testes antes da implementação da funcionalidade pode ter uma correlação positiva com a produtividade alcançada.

A maioria dos estudos acima mencionados foi realizado com indivíduos sem experiência em TDD, geralmente estudantes. Müller e Höfer (2007) realizaram um estudo para determinar a influência da experiência, ou a sua falta, no desenvolvimento empregando TDD. Sete profissionais com experiência prévia, que variava de um a seis anos, em TDD e onze estudantes inexperientes participaram da implementação de um sistema de elevador. Observou-se, durante o estudo de caso, que os profissionais seguiram mais apropriadamente o "ciclo" do TDD, Figura 4.1, implementando os testes antes da funcionalidade, executando o teste recém-criado antes de se implementar a funcionalidade – mesmo sabendo que o teste não será satisfeito – e refatorando ao final do "ciclo" quando necessário. Sumarizando, os resultados do estudo são:

- Os profissionais seguiram mais adequadamente o ciclo do TDD em relação à maioria dos estudantes envolvidos. Entretanto, nenhum membro dos dois grupos alcançou cem por cento de conformidade durante todo o estudo.

- Os testes criados pelos profissionais têm mais cobertura (*block coverage*) que os testes criados pelos estudantes.

Müller e Höfer (2007) afirmam que, de acordo com esses resultados, estudos que pretendem investigar os benefícios e desvantagens do TDD não podem ser generalizados caso empreguem estudantes com pouca experiência na prática.

4.5 *Test Doubles*

A maioria das classes de um sistema não é utilizada isoladamente, elas colaboram com outras classes para satisfazer um determinado comportamento. O comportamento de uma classe é definido em termos dos métodos que ela invoca e resultados retornados pelos métodos definidos nessa classe (Freeman et al., 2004).

Ao empregar TDD, o desenvolvedor deve enfocar a criação de testes relacionados unicamente a uma classe. Entretanto, como mencionado, as classes colaboram entre si. As colaboradoras são descobertos conforme se avança no desenvolvimento da classe. Aplicando TDD, muito freqüentemente, o desenvolvedor necessita testar a colaboração da classe em desenvolvimento com seus colaboradores, na maioria das vezes antes mesmo desses colaboradores existirem ou estarem completamente implementados. Isso torna a criação dos testes mais complexa. Tais colaboradores podem ser substituídos por *test doubles* (Meszaros, 2007), implementações alternativas de uma interface ou classe que substituem os colaboradores durante os testes (Koskela, 2007). Outros motivos para introdução de *test doubles* durante a realização dos testes são:

- Não existe nenhuma implementação do colaborador disponível, isto é, não há nenhuma classe que implementa a interface necessária ou o colaborador foi concebido enquanto o desenvolvedor *programava por intenção*.
- A implementação real do colaborador apresenta problemas de desempenho, ou seja, demanda muitos recursos computacionais.
- A classe colaboradora é difícil de ser instanciada e configurada para realização dos testes.
- A utilização da implementação real do colaborador produz efeitos colaterais (*side effects*).

Existem vários tipos de *test doubles*, no entanto, a terminologia utilizada atualmente para defini-los é confusa e inconsistente, são usados diferentes termos para

comunicar o mesmo conceito. A terminologia utilizada neste trabalho é a mesma empregada por Meszaros (2007), e os tipos de *test doubles* mais relevantes no contexto deste trabalho são:

Stub: são simples implementações de uma interface. Normalmente os métodos de um objeto *stub* retornam constantes.

Mock: são mais complexos, em termos de implementação, que os objetos *stub*, pois, possuem maneiras de verificar as "colaborações" que ocorreram durante a execução dos testes (Thomas e Hunt, 2002; Fowler, 2007). Um exemplo de colaboração é a invocação de métodos de um colaborador direto. Tipicamente, um objeto *mock* pode conter, além da capacidade de verificar a ocorrência de colaborações, a funcionalidade de um objeto *stub*. Possibilitando, quando necessário, retornar constantes em resposta às colaborações com a classe sendo testada.

Objetos *stub*, usualmente, são produzidos sem o auxílio de nenhuma ferramenta. O mesmo é aplicável para objetos *mock*, que também podem ser produzidos sem o apoio de nenhuma ferramenta. Contudo, devido a sua complexidade, na maioria das vezes, frameworks ou bibliotecas são utilizados para geração dinâmica desse tipo de objeto. Exemplos de bibliotecas utilizadas para geração de objetos *mock* são: jMock[1], rMock[2] e EasyMock (apresentado na Subseção 4.6.2).

4.6 Ferramentas de Apoio ao TDD

Atualmente existem várias ferramentas disponíveis para facilitar a utilização do TDD. Duas categorias de ferramentas utilizadas neste trabalho são descritas a seguir. Na Subseção 4.6.1 descreve-se a família de frameworks *xUnit*, utilizada para facilitar a criação e automatização de testes, e o JUnit (Louridas, 2005; JUnit, 2007) que foi usado neste trabalho; a ferramenta empregada para geração de objetos *mock* durante o desenvolvimento do framework GRENJ é examinada na Subseção 4.6.2.

4.6.1 Família *xUnit*

Para automatização do TDD a família de ferramentas mais difundida é a *xUnit*, que são frameworks que automatizam a execução dos testes e podem ser utilizados juntamente com vários IDEs. O framework precursor, denominado SUnit, foi desenvolvido na linguagem Smalltalk por Kent Beck. Posteriormente, com base no projeto do SUnit[3], o framework JUnit foi desenvolvido por Erich Gamma e Kent Beck, empregando a

[1] *http://www.jmock.org/*
[2] *http://rmock.sourceforge.net/*
[3] *http://sunit.sf.net*

linguagem Java™; atualmente várias linguagens de programação possuem uma versão desse framework[4]. Os frameworks dessa família, de acordo com Meszaros (2007), foram projetados com os seguintes objetivos:

- tornar a criação de testes possível sem que o desenvolvedor necessite aprender uma nova linguagem de programação;
- facilitar a execução de um ou mais testes com o mínimo de esforço possível.

Neste trabalho, o framework JUnit versão 4.1 é utilizado para automatização dos testes. A Figura 4.3 ilustra o framework JUnit sendo utilizando em conjunto com o IDE Eclipse. A exibição de uma barra vermelha indica que algum teste não foi satisfeito e a barra verde indica que todos os testes foram satisfeitos. Isso deu origem ao mnemônico usado para definir, sucintamente, o desenvolvimento utilizando TDD: *"vermelho-verde-refatorar"* (*red-green-refactor*) (Beck, 2002; Koskela, 2007). O Apêndice A apresenta uma introdução ao framework JUnit.

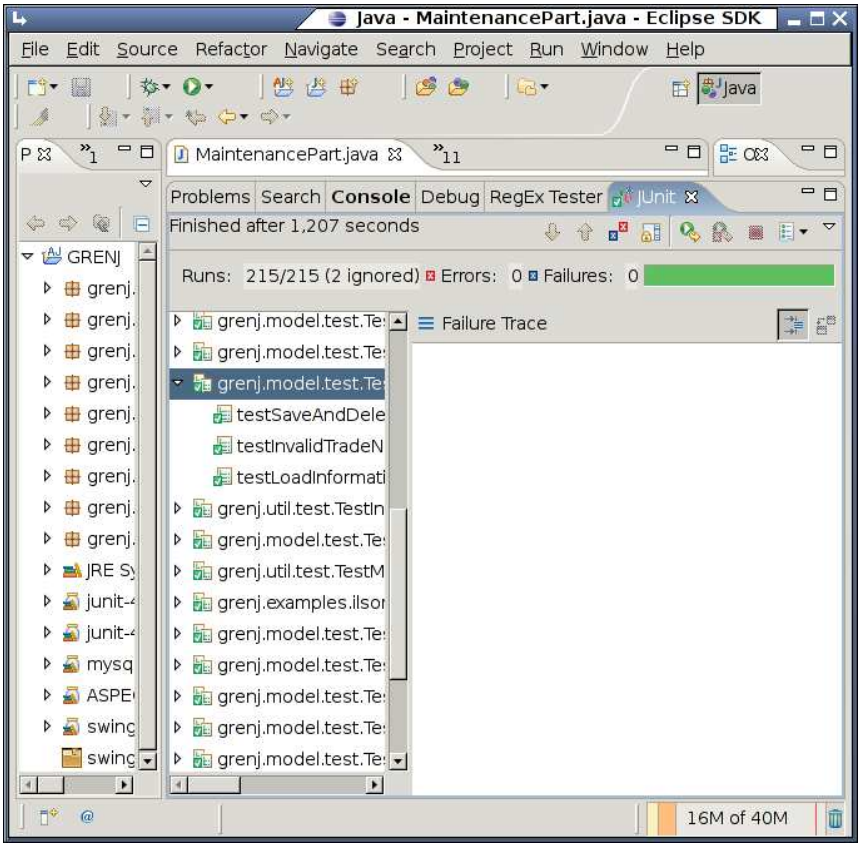


Figura 4.3: Utilizando o framework JUnit juntamente com o IDE Eclipse

[4]Algumas das linguagens que fazem parte da familia *xUnit*: ColdFusion, C#, C++, Delphi, Haskell, JavaScript, Visual Basic, Python, Perl, PHP, Lingo, LISP, Objective-C, PL/SQL, PowerBuilder e Ruby.

4.6.2 EasyMock

A geração de objetos *mock*, conforme mencionado na Seção 4.5, pode ser facilitada por meio da utilização de ferramentas. Durante a implementação do framework GRENJ utilizou-se a biblioteca EasyMock, versão 2.3. Essa biblioteca é distribuída sob os termos da licença *Massachusetts Institute of Technology* (MIT)[5]. Utilizando essa biblioteca é possível criar objetos *mock* a partir de interfaces Java™e, utilizando uma extensão da biblioteca, também a partir de classes. O EasyMock utiliza várias classes da API de *reflection* da linguagem Java™, pacote `java.lang.reflect`, para implementação dessa funcionalidade. No Apêndice B é fornecida uma introdução ao EasyMock. Alguns benefícios da utilização do EasyMock são:

- não há necessidade de se implementar os objetos *mock* manualmente;
- os objetos *mock* dinamicamente criados pela biblioteca são *refactoring-safe*, ou seja, o código dos testes não necessita ser alterado quando refatorações, que alteram o nome de um método ou a ordem de seus parâmetros, são realizadas na interface usada para geração do objeto *mock*.

Uma desvantagem da biblioteca EasyMock é que ela, a partir da versão 2.2, é compatível somente com o Java SE Development Kit (JDK) 1.5 ou versões posteriores.

4.7 Implementando a Seqüência de Fibonacci Aplicando TDD

Exemplifica-se a seguir, utilizando a linguagem Java™e o framework JUnit, a implementação da seqüência de Fibonacci utilizando TDD. Na matemática, os números de Fibonacci são uma seqüência definida pela fórmula abaixo:

$$F(n) = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ F(n-1) + F(n-2) & \text{outros casos.} \end{cases}$$

A seqüência inicia com 0 e 1 e tem a propriedade de que cada número subseqüente é a soma de dois números que o precedem. Aplicando TDD, inicialmente divide-se a funcionalidade a ser implementada em um conjunto de testes. Esses casos de teste são mantidos em uma lista e novos testes podem ser adicionados conforme se avança no desenvolvimento. A seqüência de Fibonacci poderia ser decomposta nos casos de teste da Tabela 4.2.

Após a criação da lista de casos de teste, seleciona-se um deles. Tal escolha é subjetiva, porém, geralmente, escolhem-se aqueles que podem ajudar o desenvolvedor a conhecer melhor o problema sendo abordado. Neste exemplo, considerando que

[5]A licença pode ser encontrada em *http://www.easymock.org/License.html*)

a seqüência de Fibonacci é um problema que pode ser resolvido utilizando recursividade, inicia-se com os casos de teste que avaliam os casos básicos da recursão: *fibonacci(0) = 0* e *fibonacci(1) = 1*. Durante a implementação do teste o desenvolvedor *programa por intenção*, tomando decisões relacionadas à API da funcionalidade que será posteriormente implementada. Por exemplo, enquanto implementa o teste, o desenvolvedor deve decidir o nome da classe que irá encapsular a funcionalidade, o nome do método que implementará a funcionalidade e se o método será de instância ou estático bem como os parâmetros que o método receberá. Erdogmus et al. (2005) ressaltam que essas decisões, tomadas durante a implementação dos casos de teste, são menos abstratas e mais relacionadas à implementação (*low-level design*). A implementação do caso de teste *fibonacci(0) = 0* é exibida na Listagem 4.1. Explicações relacionadas à criação dos testes usando o framework JUnit serão omitidas neste exemplo.

Tabela 4.2: Lista de casos de teste para seqüência de Fibonacci

Lista de Casos de Teste
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2

Listagem 4.1: Implementação do caso de teste *fibonacci(0) = 0*

```
1  import static org.junit.Assert.*;
2  import org.junit.Test;
3
4  public class TesteFibonacci {
5
6      @Test
7      public void testeFibonacci() {
8          assertEquals( 0, Calculadora.fibonacci( 0 ) );
9      }
10
11 }
```

O teste da Listagem 4.1 especifica que dado a constante `0` como parâmetro o método `fibonacci` da classe `Calculadora` deve retornar `0` como resultado. No entando, esse teste ainda não pode ser executado, pois, a classe `Calculadora` e o método estático `fibonacci` foram somente concebidos enquanto se programava por intenção e, sendo assim, ainda não estão implementados. Deve-se implementar os elementos ne-

cessários para que o teste compile, nesse caso: uma classe denominada `Calculadora` com um método estático denominado `fibonacci` (Listagem 4.2).

Listagem 4.2: Implementação dos elementos necessários para que o teste da Listagem 4.1 compile: classe `Calculadora` e método estático `fibonacci`

```
1 public class Calculadora {
2
3    public static int fibonacci( int num ) {
4       return 0;
5    }
6
7 }
```

Nesta etapa, enquanto implementa-se os elementos necessários, pode-se empregar uma técnica denominada *Fake It* (Beck, 2002; Koskela, 2007). Essa técnica consiste em, inicialmente, retornar constantes e, gradualmente, substitui-las por variáveis, até que a funcionalidade tenha sido implementada corretamente. Utilizando essa técnica retorna-se a constante `0` do método `fibonacci` para que o teste da Listagem 4.1 seja satisfeito.

A próxima etapa é refatorar o código eliminando *bad smells* (Fowler et al., 1999) tanto do código relacionado aos testes quando do código que implementa a funcionalidade. Todavia, como nenhum deles possui código duplicado ou outro tipo de *bad smell* nenhuma refatoração é realizada.

Prossegue-se com a seleção e implementação de outro caso de teste, mais especificamente: *fibonacci(1) = 1*. A implementação do teste é apresentada na Listagem 4.3; somente as linhas de código relevantes ao exemplo são apresentadas.

Listagem 4.3: Implementação do segundo caso de teste, *fibonacci(1) = 1*

```
6 @Test
7 public void testeFibonacci() {
8    assertEquals( 0, Calculadora.fibonacci( 0 ) );
9    assertEquals( 1, Calculadora.fibonacci( 1 ) );
10 }
```

O método `fibonacci` da classe `Calculadora` ainda não implementa a funcionalidade adequadamente. O segundo teste especifica que dado a constante `1` como parâmetro o método `fibonacci` deve retornar a constante `1` como resultado. Contudo, o método `fibonacci` retorna somente a constante `0` (Listagem 4.2, linha 4). De forma que o teste, quando executado, não é satisfeito.

Utilizando TDD implementa-se a funcionalidade avaliada pelos testes da maneira mais simples possível e todos os testes devem continuar sendo satisfeitos (Beck, 2002;

Martin, 2007; Jeffries e Melnik, 2007). Assim, o método `fibonacci` é alterado de forma que satisfaça os dois testes já implementados. Essas alterações são exibidas na Listagem 4.4.

Listagem 4.4: Alterações no método `fibonacci` a fim de satisfazer os dois testes implementados

```
3 public static int fibonacci( int num ) {
4    if ( num == 0 ) return 0;
5    return 1;
6 }
```

Após os ajustes da Listagem 4.4, os testes são executados e ambos são satisfeitos. Entretanto, a implementação do segundo teste introduziu código duplicado. Como pode ser observado na Listagem 4.3, as duas instruções utilizadas para avaliar o resultado são semelhantes (linhas 8 e 9), somente os parâmetros fornecidos são diferentes; `0` na linha 8 e `1` na linha 9. Desse modo, o código dos testes é refatorado conforme a Listagem 4.5. Após as refatorações a implementação dos casos de teste restantes é facilitada. A implementação do próximo caso de teste é apresentada na Listagem 4.6. Executam-se novamente os testes e todos são satisfeitos, inclusive o teste recém-implementado, pois, a constante retornada pelo método `fibonacci` é igual ao resultado esperado para esse teste.

Listagem 4.5: Código dos testes após refatoração

```
6 @Test
7 public void testeFibonacci() {
8    //array que armazena: {resultado esperado, num}
9    int[][] casos = new int[][] { {0, 0},{1, 1} };
10
11   for( int i = 0; i < casos.length; i++ ) {
12      assertEquals( casos[ i ][ 0 ],
13      Calculadora.fibonacci( casos[ i ][ 1 ] ) );
14   }
15 }
```

Listagem 4.6: Alterações necessárias para implementação do caso de teste *fibonacci(2) = 1*

```
8 //array que armazena: {resultado esperado, num}
9 int[][] casos = new int[][] { {0, 0},{1, 1}, {1, 2}};
```

A implementação do teste antecedente não adicionou nenhum *bad smell*, assim

segue-se com a implementação do próximo caso de teste, ilustrado no trecho de código da Listagem 4.7. Executam-se todos os testes e o recém-adicionado não é satisfeito, visto que espera-se 2 como resultado e a implementação do método `fibonacci` retorna somente a constante `1` para parâmetros diferentes de `0`; como pode ser observado nas linhas 4 e 5 da Listagem 4.4. Utilizando novamente a técnica *Fake It* (Beck, 2002; Koskela, 2007), pode se satisfazer os testes existentes ajustando a implementação da Listagem 4.4 de acordo com o exibido na Listagem 4.8.

Listagem 4.7: Alterações necessárias para implementação do caso de teste *fibonacci(3) = 2*

```
8 //array que armazena: {resultado esperado, num}
9 int[][] casos = new int[][] { {0, 0},{1, 1}, {1, 2}, {2, 3} };
```

Listagem 4.8: Implementação da funcionalidade utilizando a técnica *Fake It* (Beck, 2002; Koskela, 2007)

```
3 public static int fibonacci( int num ) {
4    if ( num == 0 ) return 0;
5    if ( num <= 2 ) return 1;
6    return 2;
7 }
```

Como pode ser observado o desenvolvedor poderia continuar aplicando a técnica *Fake It*, no entanto, isso não levaria a implementação apropriada da funcionalidade. Dessa forma, gradualmente, o desenvolvedor substitui as constantes do método `fibonacci` por variáveis ou métodos. A seqüência de Fibonacci então é implementada como a seguir:

(i) substitui-se a constante 2 pela expressão `1 + 1`, executam-se os testes e todos continuam sendo satisfeitos (Listagem 4.9).

Listagem 4.9: Substituindo constantes por variáveis, primeiro passo

```
3 public static int fibonacci( int num ) {
4    if ( num == 0 ) return 0;
5    if ( num <= 2 ) return 1;
6    return 1 + 1;
7 }
```

(ii) a primeira constante `1` (Listagem 4.9, linha 6) é alterada para uma invocação recursiva do método estático `fibonacci` (Listagem 4.10, linha 6), executam-se

todos os testes e eles continuam sendo satisfeitos, sinalizando que o comportamento externo do método não foi alterado.

Listagem 4.10: Substituindo constantes por variáveis, segundo passo

```
3 public static int fibonacci( int num ) {
4    if ( num == 0 ) return 0;
5    if ( num <= 2 ) return 1;
6    return fibonacci( num - 1 ) + 1;
7 }
```

(iii) por fim, altera-se a segunda constante `1` por uma chamada recursiva (Listagem 4.11 linha 6), tornando a segunda instrução `if` menos abrangente (Listagem 4.11 linha 5). É importante ressaltar que a ordem dessas alterações é importante, caso a ordem tivesse sido invertida, após a alteração da instrução `if` os testes não seriam satisfeitos. Depois de cada uma dessas alterações os testes são executados a fim de detectar a introdução de problemas ou se o comportamento observável do método foi alterado. Como todos os testes da lista de casos de testes foram implementados e satisfeitos pode se considerar que a funcionalidade encontra-se implementada. A implementação da seqüência de Fibonacci utilizando testes é exibida na Listagem 4.11.

Listagem 4.11: Implementação da funcionalidade seqüência de Fibonacci

```
3 public static int fibonacci( int num ) {
4    if ( num == 0 ) return 0;
5    if ( num == 1 )return 1;
6    return fibonacci( num - 1 ) + fibonacci( num - 2 );
7 }
```

4.8 Considerações Finais

O custo associado à remoção de problemas nos sistemas aumenta de acordo com a demora para descobri-los. Nas abordagens tradicionais de desenvolvimento, a criação de testes é uma das últimas atividades realizadas. Assim, a detecção de problemas, inseridos durante a análise, projeto e implementação do sistema, é tardia e a remoção desses problemas mais dispendiosa. Além disso, os sistemas são, normalmente, implementados sem considerar a posterior criação dos testes, resultando em baixa testabilidade. Nos sistemas desenvolvidos dessa forma torna-se complexo isolar parte da funcionalidade, exercitá-la e verificar os resultados, comprometendo tanto a realização dos testes quanto a qualidade do produto.

O desenvolvimento utilizando TDD requer a implementação de mais código em comparação às abordagens *test-last*, visto que além do código da funcionalidade, o desenvolvedor precisa implementar também os testes. Porém, devido a implementação dos testes antes da implementação da funcionalidade, a testabilidade dos elementos que implementam a funcionalidade é considerada desde o início do desenvolvimento. Por meio dos testes obtém-se rápido *feedback* sobre as decisões de análise, projeto e implementação. No TDD, o conjunto de testes cresce conforme o desenvolvimento avança e, como testes são essenciais para garantir a qualidade dos sistemas, a criação de testes desde o início adiciona qualidade aos sistemas produzidos. Esses testes podem atuar como testes de regressão, evitando a introdução de problemas, quando futuras alterações forem necessárias.

Conforme pôde ser observado no exemplo apresentado na Seção 4.7, o TDD possibilita que a funcionalidade almejada seja implementada iterativamente aplicando técnicas como *Fake It*, o que facilita a implementação de funções complexas.

Este Capítulo apresentou conceitos relacionados ao TDD, necessários para compreensão do processo de reengenharia descrito neste trabalho. Também foram apresentados estudos que avaliam os benefícios ou possíveis desvantagens da utilização da abordagem proposta pelo TDD em relação à abordagem mais tradicional de desenvolvimento, *test-last*. Foi exposto o conceito de *test double* e as características de principalmente dois tipos: objetos *stub* e objetos *mock*, visto que esses dois tipos foram amplamente utilizados durante a implementação do framework GRENJ utilizando TDD.

Durante a implementação do GRENJ empregou-se o JUnit para automatização dos testes, os objetos *stub* foram implementados sem o auxílio de nenhuma ferramenta, enquanto que para a geração dinâmica dos objetos *mock* foi utilizada a biblioteca EasyMock. O framework JUnit e a biblioteca EasyMock foram apresentados.

O próximo Capítulo detalha o processo de reengenharia realizado e o papel fundamental do TDD nesse contexto.

CAPÍTULO
5

Um Processo de Reengenharia Iterativo

5.1 Considerações Iniciais

Há diferentes tipos de processos estabelecidos para realização da engenharia avante (*forward engineering*), entretanto, não há um modelo amplamente estabelecido para realização de atividades de reengenharia (Mens e Tourwe, 2004). Há consenso de que reengenharia envolve atividades de engenharia reversa e de engenharia avante ou de reestruturação (Chikofsky e Cross II, 1990). Mens e Tourwe (2004) afirmam que a melhor abordagem, dada a ausência de processos estabelecidos, é a utilização de conhecimento adquirido em atividades anteriores de reengenharia, documentado na forma de padrões que podem ser consultados e novamente aplicados.

O processo de reengenharia mais conhecido é o que segue o modelo de processo seqüencial linear, porém com o surgimento dos métodos ágeis, há interesse que as práticas neles existentes sejam também utilizadas na reengenharia de software. Como comentado anteriormente, a reutilização de experiência e de práticas de sucesso possibilitam que um software passe pelo processo de reengenharia com maior segurança, qualidade e menor esforço para a sua realização.

Este Capítulo descreve um processo iterativo de reengenharia que foi utilizado para a obtenção do framework GRENJ. Para o entendimento do software legado, framework GREN (Braga, 2002b), na etapa de engenharia reversa foram aplicados alguns padrões definidos por Demeyer et al. (2002). Na etapa de engenharia avante foi utilizado TDD, uma prática de análise e projeto (Cockburn, 2006; Jeffries e Melnik, 2007)

e que abrange atividades de refatoração.

Nenhuma ferramenta de conversão automatizada de código Smalltalk para código Java™ foi utilizada. É complexo alcançar a equivalência semântica necessária nesse tipo de conversão envolvendo linguagens dinamicamente tipadas, como por exemplo Smalltalk, e linguagens fortemente tipadas, como por exemplo Java™. Segundo Terekhov e Verhoef (2000) a tecnologia utilizada para conversão entre linguagens não é apropriada, podendo comprometer o framework resultante. Além disso, as características singulares da linguagem Smalltalk tornam a conversão, para outra linguagem de alto nível, mais complexa que o usual (Engelbrecht e Kourie, 2003).

A organização deste Capítulo é da seguinte forma: na Seção 5.2 é comentada uma justificativa para a utilização da linguagem Java™ na implementação do framework GRENJ; na Seção 5.3 é apresentado o processo de reengenharia utilizado, como uma proposta para realização de reengenharia iterativa. A Seção 5.4 descreve a reengenharia do framework GREN utilizando o processo proposto; a Seção 5.5 ressalta os benefícios provenientes da utilização de testes automatizados durante a implementação do GRENJ; na Seção 5.6 é comentada a convenção de codificação utilizada durante a implementação do framework GRENJ. Na Seção 5.7 são mostrados os aperfeiçoamentos introduzidos no framework GRENJ e a Seção 5.8 apresenta as considerações finais deste Capítulo.

5.2 Motivações para Utilização da Linguagem Java™

Smalltalk é uma linguagem dinâmica e puramente orientada a objetos, na qual, ao contrário de linguagens como Java™ e C++, os tipos primitivos (por exemplo, inteiros) também são objetos. Embora seja uma linguagem fortemente orientada a objetos, Smalltalk já não é tão difundida nas comunidades acadêmica e profissional de Engenharia de Software.

Entre as similaridades existentes entre a linguagem Java™ e Smalltalk pode se mencionar: são orientadas a objetos, possuem gerenciamento de memória automático e são interpretadas[1]. Programas desenvolvidos na linguagem Java™, usualmente, possuem melhor desempenho e são mais robustos em relação àqueles desenvolvidos na linguagem Smalltalk, dado que a tecnologia Java™ inclui um verificador de *bytecode* (*bytecode verifier*) que garante a integridade do código Java™ compilado, reduzindo o número de verificações em tempo de execução (Niemeyer e Knudsen, 2005).

Diante das considerações mencionadas anteriormente, a linguagem Java™ foi escolhida para ser utilizada na implementação da "versão" do framework GREN resultante do processo iterativo de reengenharia e que é denominado Gerenciamento de

[1] De acordo com Niemeyer e Knudsen (2005) Java™ é uma linguagem tanto compilada quanto interpretada, visto que código Java™ após compilado é transformado em *bytecode*.

REcursos de Negócios em Java™(GRENJ).

5.3 O Processo Iterativo de Reengenharia

O processo iterativo de reengenharia é composto de dois tipos de atividades. O primeiro tipo aborda a extração do projeto existente e o segundo está relacionado à implementação e ao aprimoramento do projeto extraído. Assim, o primeiro caracteriza-se como atividade de engenharia reversa e o segundo como atividade de engenharia avante e reestruturação. Durante a descrição deste processo de reengenharia, o termo solução legada é utilizado para descrever o sistema legado que está passando por reengenharia e o termo solução alvo é empregado para representar o sistema resultante do processo de reengenharia.

Usualmente, as soluções legadas são sistemas complexos. Desse modo, o processo iterativo de reengenharia proposto considera "parcelas gerenciáveis" das soluções legadas e não o sistema como um todo (Demeyer et al., 2002). As soluções legadas podem ser divididas em "parcelas" de granularidade grossa (*coarse-grained*), como por exemplo camadas e pacotes, ou de granularidade fina (*fine-grained*) como classes.

Após a divisão da solução legada, cada uma das "parcelas" é abordada em uma iteração do processo. Nessa iteração realizam-se atividades de engenharia reversa e avante. Assim, a solução legada passa iterativamente por reengenharia conforme a solução alvo é implementada. A engenharia reversa emprega padrões definidos por Demeyer et al. (2002) para a aquisição e compreensão de informações sobre as "parcelas" da solução legada que estão sendo examinadas. Durante as atividades de engenharia avante, TDD é utilizado para implementação das informações relacionadas à "parcela" da solução legada tratada durante a engenharia reversa. Nesse processo, as "parcelas" da solução legada são equivalentes às "parcelas" da solução alvo, agora implementadas na linguagem escolhida e com os testes referentes. Uma visão geral do processo é exibida na Figura 5.1.

Uma característica das soluções legadas, de acordo com (Demeyer et al., 2002), é que a maioria delas não tem documentação disponível ou a existente não é fidedigna, pois deixou de refletir o estado atual da funcionalidade implementada. Para tratar esse tipo de problema, de maneira iterativa, dois padrões de engenharia reversa propostos por (Demeyer et al., 2002) são utilizados. Com a aplicação desses padrões, as informações sobre a solução legada são obtidas e validadas utilizando tanto o código fonte quanto a documentação disponível. Isso evita que possíveis informações inconsistentes, encontradas na documentação, sejam consideradas e, conseqüentemente, implementadas na solução alvo. Na Subseção seguinte, as atividades de engenharia reversa, realizadas iterativamente e apoiadas por padrões, são descritas.

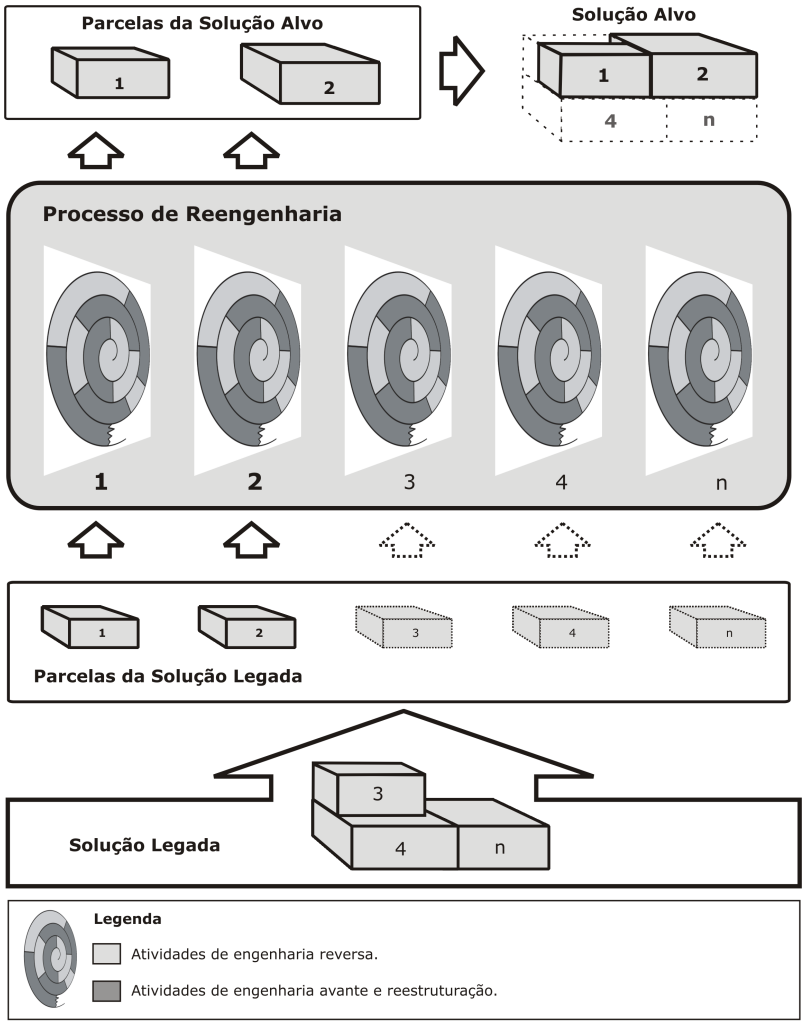


Figura 5.1: Visão geral do processo iterativo de reengenharia

5.3.1 Atividades de Engenharia Reversa Utilizando Padrões

As atividades de engenharia reversa realizadas têm como principal fonte de informações o código da solução legada e a documentação disponível. Dessa forma, alterna-se entre consultas à documentação e ao código fonte da solução legada com o propósito de compreender e validar as informações sobre a "parcela" da solução legada que está passando por reengenharia. Para realização dessas duas atividades são usados os padrões: *Read All the Code in One Hour* e *Skim the Documentation* (Ducasse et al., 1999a; Demeyer et al., 2002). É importante ressaltar que esses padrões são recomendados para reengenharia realizada de maneira "não iterativa", porém são também pertinentes para processos iterativos como comentam Demeyer et al. (2002).

O objetivo do padrão *Read All the Code in One Hour* é avaliar a qualidade e a complexidade, entre outros atributos, do código fonte. Tal avaliação é realizada por

meio de uma intensiva revisão do código fonte da solução legada. Há uma importante diferença entre as revisões de código tradicionais e essa realizada no contexto do processo de reengenharia. A primeira, geralmente, procura detectar possíveis problemas existentes no código fonte. Enquanto que a segunda tem o objetivo de extrair uma impressão inicial da qualidade do código fonte da solução legada e informações sobre como a funcionalidade está implementada.

O padrão originalmente sugere que todo o código da solução legada seja lido em uma hora. No entanto, levando em consideração as características iterativas deste processo, somente os trechos de código relacionados à "parcela" da solução legada que está passando por reengenharia são lidos. Dessa forma, há diminuição na quantidade de relacionamentos entre classes que devem ser compreendidos em cada iteração. Contudo, Demeyer et al. (2002) ressaltam que um dos pontos negativos de se aplicar o padrão *Read All the Code in One Hour*, é que as informações obtidas são de baixo nível de abstração. Conseqüentemente, deve se complementar o conhecimento obtido consultando representações mais abstratas do sistema – como, por exemplo, diagramas de classes, seqüência, entre outros, se existirem.

O padrão *Skim the Documentation* (Demeyer et al., 2002) é aplicado a fim de avaliar a relevância da documentação disponível, podendo ser aplicado antes ou depois do padrão *Read All the Code in One Hour*. No contexto deste processo de reengenharia, ele é aplicado com o propósito de selecionar as partes da documentação que podem conter informações relevantes relacionadas à "parcela" da solução legada que está passando por reengenharia. A aplicação do padrão *Skim the Documentation* resulta na obtenção de informações, em alto nível de abstração, sobre a funcionalidade da parcela da solução legada. Tais informações servem para validar e complementar as informações, de baixo nível de abstração, obtidas com a utilização do padrão *Read All the Code in One Hour*.

As atividades que devem ser realizadas durante a engenharia reversa estão ilustradas no diagrama de atividades da Figura 5.2. Os padrões *Read All The Code in One Hour* e *Skim the Documentation* são aplicados iterativamente com o propósito de se adquirir informações sobre as "parcelas" da solução legada (cada "parcela" é considerada em uma iteração). Neste processo, não há uma ordem específica para aplicação desses dois padrões e, após a aplicação, as informações obtidas são verificadas e sintetizadas. Essa verificação é necessária dado que a documentação pode estar desatualizada. A iteratividade nesse momento possibilita ao desenvolvedor decidir quando as informações são suficientes para realização das atividades de engenharia avante. O ícone existente em *Atividades de engenharia avante* indica que essa será detalhada posteriormente, na Seção 5.3.3.

Algumas "parcelas" da solução legada podem ser difíceis de serem compreendidas se somente o código fonte e a documentação forem analisados. Essa dificuldade é

maior, principalmente, se a documentação não contém um diagrama de classes, em nível de análise ou de projeto, ou qualquer representação mais abstrata da funcionalidade implementada. A Subseção a seguir descreve como diagramas de classes das "parcelas" mais complexas da solução legada podem ser produzidos.

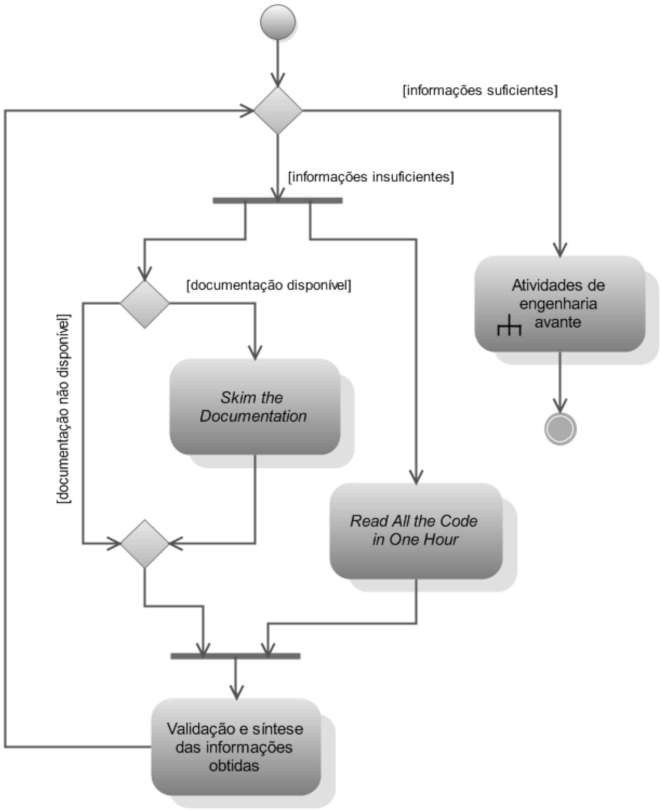


Figura 5.2: Atividades de engenharia reversa

5.3.2 Construção do Diagrama de Classes de Parcelas Complexas da Solução Legada

Algumas "parcelas" da solução legada podem ser mais facilmente compreendidas por meio de diagramas de classes. Para a elaboração de tais diagramas pode-se utilizar o padrão de engenharia reversa *Speculate About Design* (Demeyer et al., 2002). Esse padrão permite que um diagrama de classes hipotético, construído com base em suposições de como a funcionalidade de uma "parcela" da solução legada encontra-se implementada, seja refinado. A partir desse diagrama hipotético, inicialmente abstrato e sem detalhes de implementação, acrescentam-se classes, métodos e atributos, tornando dessa forma o diagrama cada vez mais próximo do que está implementado na solução legada.

No caso de soluções legadas que possuem diagramas de classes como parte da documentação, é necessário verificar se tais diagramas são significativos em relação ao código fonte existente. Essa verificação pode ser realizada também com a aplicação do padrão *Speculate About Design*. Demeyer et al. (2002) recomendam que os diagramas de classes em nível de análise, caso estejam disponíveis, sejam iterativamente refinados. Dessa forma, somente as informações relevantes para propósitos de reengenharia são adicionadas ao diagrama.

Embora haja uma seção denominada *Solução*, que descreve os passos necessários para utilização do padrão, esses passos foram alterados para a aplicação no contexto de reengenharia aqui apresentado. Essa alteração tornou a aplicação do padrão mais flexível, de forma que ele é aplicado até que o desenvolvedor esteja satisfeito com o diagrama de classes produzido. Os passos agora utilizados são:

1. O desenvolvedor produz um diagrama de classes com base na sua compreensão da "parcela" da solução legada e de como supõe que a funcionalidade esteja implementada.

2. Os nomes das classes, métodos e atributos desse diagrama de classes são enumerados. Posteriormente, o desenvolvedor procura por esses nomes no código fonte da parcela que está sendo considerada na iteração. Adicionalmente, os nomes podem ser classificados de acordo com a probabilidade que eles têm de aparecer no código fonte da solução legada. Os elementos com maior probabilidade são pesquisados primeiro.

3. O desenvolvedor mantém no diagrama as classes, métodos e atributos que foram encontrados no código fonte e elimina aqueles que não foram encontrados. Nesse passo é possível reclassificar os elementos remanescentes em relação às suas probabilidades de ocorrência.

4. O diagrama, com base nos erros e acertos, é adaptado da seguinte forma:

   (a) renomear os elementos do diagrama (classes, métodos e atributos) quando seus nomes, no código fonte, forem diferentes dos "nomes hipotéticos".

   (b) remodelar os elementos que não correspondem à implementação. Por exemplo: pode ser necessário transformar uma operação em uma classe e um atributo em uma operação.

   (c) adicionar elementos que não aparecem no diagrama hipotético, mas que existem no código fonte.

5. Os passos de 2 a 4 são repetidos até que o diagrama esteja satisfatório.

De acordo com Demeyer et al. (2002) a aplicação do padrão *Speculate About Design* é apropriada para sistemas que possuem mais de cem classes. No entanto, a utilização desse padrão tem as seguintes desvantagens:

Requer experiência: o desenvolvedor deve ter conhecimento dos idiomas usados na linguagem em que a solução legada está implementada, bem como dos padrões de projeto e dos algoritmos, para que possa reconhecer a ocorrência desses elementos no código fonte.

Consome muito tempo: para se obter um diagrama de classes satisfatório da parcela da solução legada que está sendo considerada.

5.3.3 Engenharia Avante Utilizando TDD e Refatoração

As informações, adquiridas anteriormente, sobre uma "parcela" da solução legada são empregadas para implementação de uma "parcela" equivalente na solução alvo. Conforme o processo de reengenharia avança, a implementação da solução alvo necessita ser aprimorada, modificada e adaptada para introdução de funcionalidade e para acomodar as novas "parcelas" que são, iterativamente, adicionadas. Essas alterações tornam o código da solução alvo mais complexo, além dos problemas que podem ser inadvertidamente introduzidos, visto que é difícil antecipar os efeitos que alterações em camadas arquiteturais ou classes podem causar. Quando as camadas ou classes possuem um alto nível de acoplamento entre si é difícil realizar alterações sem produzir um *problema de regressão*. Um problema de regressão ocorre quando alguma funcionalidade, anteriormente implementada corretamente, deixa de desempenhar o comportamento esperado devido às alterações realizadas (Koskela, 2007; Meszaros, 2007). Duas práticas podem ser utilizadas para atenuar a inserção de problemas durante a realização de alterações: TDD apresentado no Capítulo 4 e refatoração comentada na Seção 2.6.

Conforme descrito no Capítulo 4, o primeiro passo para aplicação do TDD é decompor a funcionalidade em uma lista de casos de teste. Assim, as informações obtidas durante as atividades de engenharia reversa são, de maneira subjetiva, transformadas em uma lista de casos de teste. Após a criação dessa lista, emprega-se TDD para implementação da solução alvo na linguagem escolhida.

Os testes criados durante a implementação utilizando TDD possibilitam que refatorações sejam realizadas, pois, detectam a introdução de problemas conseqüentes das alterações realizadas durante as refatorações; quando os testes existentes são executados após a realização de refatorações eles são denominados testes de regressão (Seção 4.2). Os testes de regressão e atividades de refatoração permitem que "parcelas" sejam adicionadas e que o projeto existente da solução alvo seja aprimorado conforme avança-se. As atividades de engenharia avante são ilustradas no dia-

grama de atividades da Figura 5.3, que detalha a atividade *"Atividades de engenharia avante"* exibida na Figura 5.2.

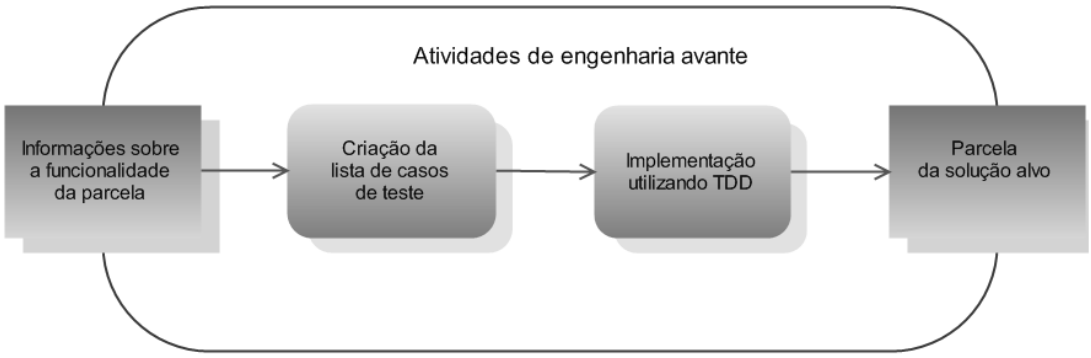


Figura 5.3: Atividades de engenharia avante

A implementação das "parcelas" da solução legada usando TDD possibilita que pouco ou nenhum projeto preliminar necessite ser realizado antes do início da implementação. Isso evita que o desenvolvedor tenha que considerar como algumas características da implementação da solução legada serão implementadas na linguagem escolhida para a solução alvo. Com TDD o desenvolvedor obtém rápido *feedback* sobre as adaptações e escolhas realizadas durante a implementação da solução alvo, possibilitando avaliar, sem a criação de um projeto preliminar abrangente, quais alternativas são mais apropriadas para cada circunstância em que as linguagens envolvidas têm características diferentes. Além disso, *programar por intenção*, conceito aplicado durante o desenvolvimento usando TDD (Seção 4.2), auxilia na implementação de uma API mais compreensível desde o início da implementação da solução alvo, pois, *programando por intenção* enfoca-se nas interfaces públicas das classes (Koskela, 2007). A Seção 5.4 apresenta um exemplo da utilização desse conceito durante a reengenharia do framework GREN.

Durante a implementação da solução alvo, refatorações podem ser realizadas com os seguintes propósitos: eliminar *bad smells* (Fowler et al., 1999) ou introduzir padrões de projeto. Refatorações consideradas de baixo nível são empregadas no primeiro caso. Alguns exemplos desse tipo de refatoração são: *Extract Method*, *Replace Temp with Query* e *Consolidate Conditional Expression* (Fowler et al., 1999). Algumas refatorações de baixo nível resultam na introdução de padrões de projeto, conforme exibido na Tabela 5.1. Porém, normalmente, padrões de projeto são inseridos por meio da aplicação de refatorações compostas (*composite refactorings*) que consistem de várias refatorações de baixo nível (Kerievsky, 2004). A introdução de padrões durante a implementação da solução alvo torna o código mais flexível e mais fácil de ser compreendido, facilitando futuras alterações.

Tabela 5.1: Refatorações e os respectivos padrões introduzidos

REFATORAÇÃO	PADRÃO
Form Template Method[1]	*Template Method*[2]
Introduce Null Object[1]	*Null Object*[3]
Replace Constructor with Factory Method[1]	*Factory Method*[2]
Replace Type Code with State/Strategy[1]	*State*[2] ou *Strategy*[2]

1→ Fowler et al. (1999); 2→ Gamma et al. (1995); 3→ Woolf (1996)

5.4 Reengenharia do Framework GREN

O processo iterativo de reengenharia apresentado foi utilizado na reengenharia do framework GREN para obter o framework GRENJ. A arquitetura do framework GREN é mostrada na Figura 5.4. Somente as camadas de negócios e de persistência passaram por reengenharia. O GREN é composto por aproximadamente trinta mil linhas de código e os padrões da GRN encontram-se implementados na camada de negócios do framework (Braga, 2002b). Durante o processo de reengenharia cada padrão foi considerado como uma "parcela".

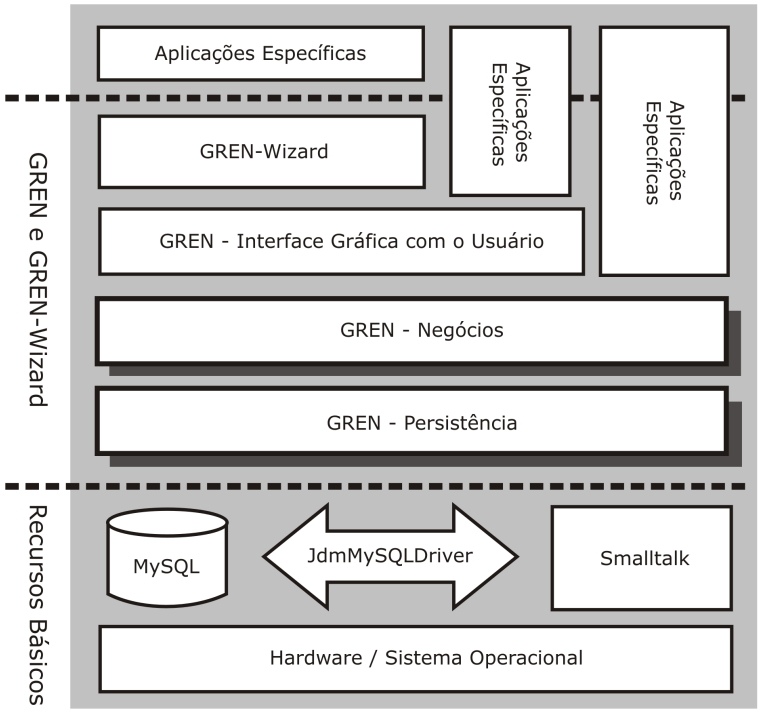


Figura 5.4: Arquitetura do framework GREN; adaptado de (Braga, 2002b)

As Subseções a seguir apresentam como as atividades de engenharia reversa e

engenharia avante foram conduzidas durante a realização da reengenharia do framework GREN, bem como as dificuldades enfrentadas.

5.4.1 Realização das Atividades de Engenharia Reversa

As atividades de engenharia reversa envolvem consultar a documentação disponível e o código fonte do sistema passando por reengenharia. No caso do GRENJ, a principal documentação é o *"cookbook"* (Braga e Masiero, 2002b), que contém o mapeamento entre o modelo de domínio e a implementação do framework. O processo de instanciação recomendado para criação de aplicações por meio do framework GREN e os métodos utilizados para configuração dessas aplicações – métodos gancho (Seção 2.5) – também são documentados pelo *"cookbook"*. Outra fonte de informações é a linguagem de padrões GRN, que fornece diagramas de classes, em nível de análise, de seus padrões; implementados no GREN. Esses dois documentos foram utilizados como fontes de informações durante o processo de reengenharia. Contudo, a principal fonte de informações foi o código fonte do framework GREN.

Para cada classe que passou por reengenharia, um documento denominado *protocolo de reengenharia* (Apêndice C) foi produzido. O *protocolo de reengenharia* documenta tanto a versão em Smalltalk de todos os métodos de uma classe quanto a versão em Java™obtida ao final da reengenharia da classe. Essas informações foram freqüentemente consultadas, durante a reengenharia do framework GREN, com o propósito de manter a consistência entre as conversões. Os *protocolos de reengenharia* também contêm informações sobre os problemas encontrados em cada classe do framework GREN, observações realizadas durante a reengenharia da classe e informações obtidas durante a leitura do seu código fonte, como por exemplo: nome dos atributos e métodos pertencentes a cada classe.

Geralmente, a reengenharia de um padrão começou com uma aplicação do padrão *Read All the Code in One Hour*. Essa aplicação do padrão visa somente avaliar a qualidade do código fonte do padrão que está sendo considerado, identificando os possíveis problemas e trechos de código mais complexos. Tais informações, após a leitura do código, são documentadas no *protocolo de reengenharia* apropriado, isto é, da classe com o código problemático ou complexo. Posteriormente, aplica-se o padrão *Skim the Documentation* com o objetivo de avaliar as partes relevantes do *"cookbook"* e da GRN, ou seja, as partes desses dois documentos que apresentam informações sobre o padrão que está passando por reengenharia. Em seguida, aplica-se novamente o padrão *Read All the Code in One Hour*, dessa vez com o propósito de obter uma descrição textual da funcionalidade implementada pelo padrão sendo considerado. Com base nessas informações, extraídas durante a última leitura do código, o padrão *Skim the Documentation* é aplicado, agora com o propósito de verificar se a documentação encontra-se "sincronizada" com a atual implementação, isto é, se

a funcionalidade extraída durante a leitura do código fonte corresponde à descrição da funcionalidade documentada pelo *"cookbook"* e, de maneira mais abstrata, pela GRN.

Os dois padrões são aplicados iterativamente até que se obtenha a descrição da funcionalidade implementada. Na Subseção seguinte, a aplicação desses padrões é exemplificada. Para tal descreve-se a realização das atividades de engenharia reversa durante a reengenharia do padrão IDENTIFICAR AS TAREFAS DA MANUTENÇÃO (Braga, 2002a).

5.4.2 Atividades de Engenharia Reversa: Padrão IDENTIFICAR AS TAREFAS DA MANUTENÇÃO

O padrão IDENTIFICAR AS TAREFAS DA MANUTENÇÃO, pertencente ao grupo relacionado aos detalhes das transações, é aplicado quando há necessidade de identificar e especificar os trabalhos realizados durante a manutenção dos recursos.

A reengenharia da "parcela" correspondente a esse padrão foi iniciada com a aplicação do padrão *Read All the Code in One Hour*. O código relacionado às classes participantes desse padrão foi lido. Para facilitar a identificação das classes que participam do padrão pode-se empregar as informações fornecidas na seção *Participantes* da GRN. No caso do padrão IDENTIFICAR AS TAREFAS DA MANUTENÇÃO, os nomes dessas classes são: *Tarefa de Manutenção*, *Manutenção do Recurso*, *Cotação da Manutenção* e *Executor da Manutenção*. O *"cookbook"* também documenta qual classe do GREN implementa cada participante.

O *browser* do ambiente Smalltalk facilita a aplicação do padrão *Read All the Code in One Hour*, pois, disponibiliza opções para busca e visualização de classes, métodos definidos na classe, métodos herdados, entre outros.

A leitura do código começou pela classe `MaintenanceTask`, que representa as tarefas a serem realizadas para que a manutenção seja concluída. Devido à simplicidade dessa classe, trechos complexos ou possíveis problemas não foram encontrados. Assim, nenhuma observação desse tipo é anotada no protocolo de reengenharia.

No GREN, as classes com informações que podem ser persistidas no banco de dados implementam os seguintes métodos: `insertionFieldClause`, `insertionValueClause`, `updateSetClause` e `updateWhereClause`. Esses métodos são declarados na classe `PersistentObject`, apresentados na Tabela 5.2. A recorrência dessa "construção", caracterizada pela aplicação do padrão *Persistence Layer* (Yoder et al., 1998), resultou em um idioma próprio do framework para persistência das informações, e facilitou a identificação das classes que têm informações persistidas no banco de dados. Visto que, durante a leitura do código da classe `MaintenanceTask`, esses métodos foram encontrados, pôde-se inferir que os atributos utilizados no método

`insertionFieldClause` correspondem às informações que devem ser persistidas. Adiciona-se essa informação à descrição da funcionalidade.

Tabela 5.2: Métodos da classe `PersistentObject`

NOME DO MÉTODO	DESCRIÇÃO
`insertionFieldClause`	Retorna a cláusula que contém os nomes das colunas, para inserção na base de dados.
`insertionValueClause`	Retorna a cláusula que contém os nomes dos métodos que retornam os valores dos atributos a serem inseridos em suas respectivas colunas na base de dados.
`updateSetClause`	Retorna a cláusula de atualização da tabela, que contém os nomes das colunas e os respectivos métodos que retornam os valores dos atributos.
`updateWhereClause`	Retorna a cláusula usada quando um registro é atualizado.

Após a leitura de parte do código fonte da classe, as informações sobre os atributos e o nome da superclasse, por exemplo, são documentadas no respectivo *protocolo de reengenharia* (Apêndice C). Obteve-se a seguinte descrição parcial sobre a funcionalidade implementada pelo padrão:

> "⇒ A classe `MaintenanceTask` mantém informações sobre o problema que deve ser resolvido, a descrição da mão-de-obra necessária para resolução do problema, a quantidade de horas gastas, o custo, o executor e a transação (manutenção ou cotação).
> ⇒ Essas informações sobre a tarefa de manutenção devem ser persistidas."

A classe `MaintenanceTask` possui uma variável de referência que pode representar tanto uma manutenção quanto uma cotação de manutenção. Assim, o próximo passo deveria ser a leitura do código de ambas as classes. Porém, essas classes já passaram por reengenharia, pois, participam de padrões que foram abordados anteriormente. A classe que representa as manutenções dos recursos compõe o nono padrão da GRN: MANTER O RECURSO. A classe que representa cotações pertence ao décimo padrão da GRN: COTAR A MANUTENÇÃO. A outra classe participante do padrão IDENTIFICAR AS TAREFAS DA MANUTENÇÃO é denominada, na GRN, *Executor da Manutenção*. Todavia, nenhuma classe com esse nome foi encontrada durante a leitura do código fonte das classes que implementam o padrão.

Prosseguiu-se, então, com a aplicação do padrão *Skim the Documentation* para determinar as partes da documentação que contêm informações relevantes relacionadas ao padrão que está passando por reengenharia, validar a descrição parcial obtida e

elucidar as dúvidas que surgiram durante a primeira leitura do código do padrão.

Os diagramas de classes em nível de análise fornecidos pela GRN podem ser consultados para se adquirir uma visão geral, de alto nível, do relacionamento entre as classes participantes dos padrões. A Figura 5.5 ilustra o diagrama de classes em nível de análise do padrão IDENTIFICAR AS TAREFAS DA MANUTENÇÃO documentado pela GRN. Durante a aplicação do padrão *Skim the Documentation*, esse diagrama foi consultado e confrontado com as informações obtidas por meio da leitura do código. Todas as associações existentes entre as classes do diagrama da Figura 5.5 e suas respectivas multiplicidades são anotadas para serem verificadas durante a próxima leitura do código fonte.

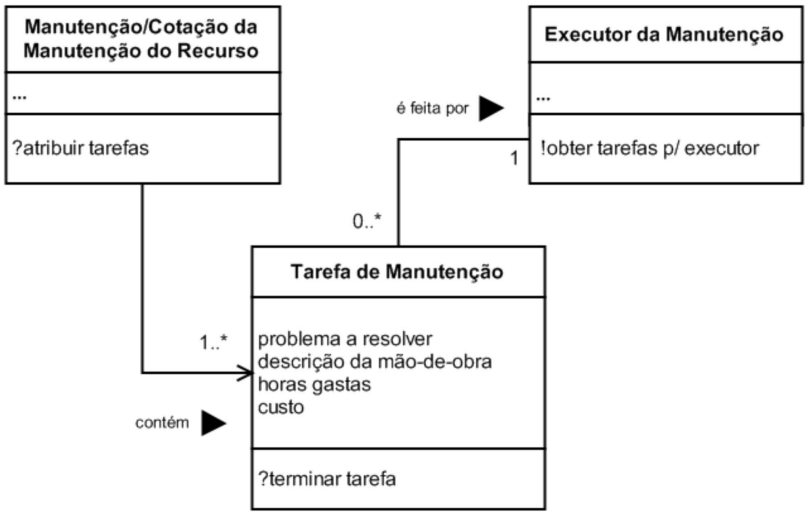


Figura 5.5: Padrão IDENTIFICAR AS TAREFAS DA MANUTENÇÃO; extraído de (Braga, 2002a)

Com o objetivo de descobrir a classe que implementa o participante *Executor da Manutenção*, o *"cookbook"* também foi consultado durante a aplicação do padrão *Skim the Documentation*. Descobriu-se que esse participante é implementado pela classe `TransactionExecutor`. Tal classe já passou por reengenharia, e também implementa o participante denominado *Executor da Transação* do padrão IDENTIFICAR O EXECUTOR DA TRANSAÇÃO, que é o décimo terceiro padrão da GRN. Como todos os outros participantes do padrão já passaram por reengenharia, na aplicação seguinte do padrão *Read All the Code in One Hour*, somente a classe `MaintenanceTask` e as anotações sobre as associações e multiplicidades foram analisadas.

Após a aplicação do padrão *Read All the Code in One Hour* todas as informações da GRN e do *"cookbook"* foram confirmadas. Como a reengenharia do padrão resumiu-se à reengenharia de uma classe simples (composta em sua maioria por métodos *getters* e *setters*), a seguinte descrição sobre a funcionalidade implementada por essa classe foi considerada satisfatória:

"⇒ A classe `MaintenanceTask` mantém informações sobre o problema que deve ser resolvido, a descrição da mão-de-obra necessária para resolução do problema, a quantidade de horas gastas, o custo, o executor e a transação (manutenção ou cotação).
⇒ Essas informações sobre a tarefa de manutenção devem ser persistidas.
⇒ Deve ser possível carregar todas as tarefas de manutenção relacionadas à mesma transação de manutenção."

Na Subseção 5.4.6 descreve-se como essa descrição foi transformada em uma lista de casos de teste e implementada utilizando TDD.

5.4.3 Abordando os Padrões mais Complexos

Devido à complexidade de alguns padrões, diagramas de classes foram produzidos por meio da aplicação do padrão *Speculate About Design*. Os diagramas de classes em nível de análise, documentados pela GRN, foram utilizados como modelo inicial e, iterativamente, refinados a cada consulta ao código fonte; como descrito na Subseção 5.3.2. Dois padrões implementados no GREN necessitaram da criação de diagramas de classes para auxiliar na compreensão da funcionalidade, a saber: LOCAR O RECURSO e MANTER O RECURSO (Braga, 2002b). A Figura 5.6 ilustra o diagrama em nível de análise do padrão LOCAR O RECURSO, já o diagrama de classes da Figura 5.7 ilustra o diagrama resultante após a aplicação do padrão *Speculate About Design*. Para obtenção do diagrama de classes da Figura 5.7 o padrão *Speculate About Design* foi aplicado em três iterações de aproximadamente três horas cada.

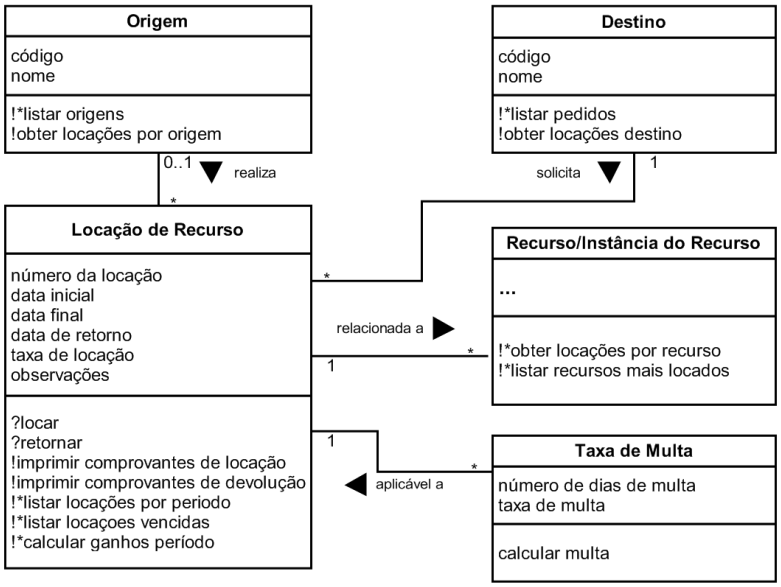


Figura 5.6: Padrão LOCAR O RECURSO; extraído de (Braga, 2002a)

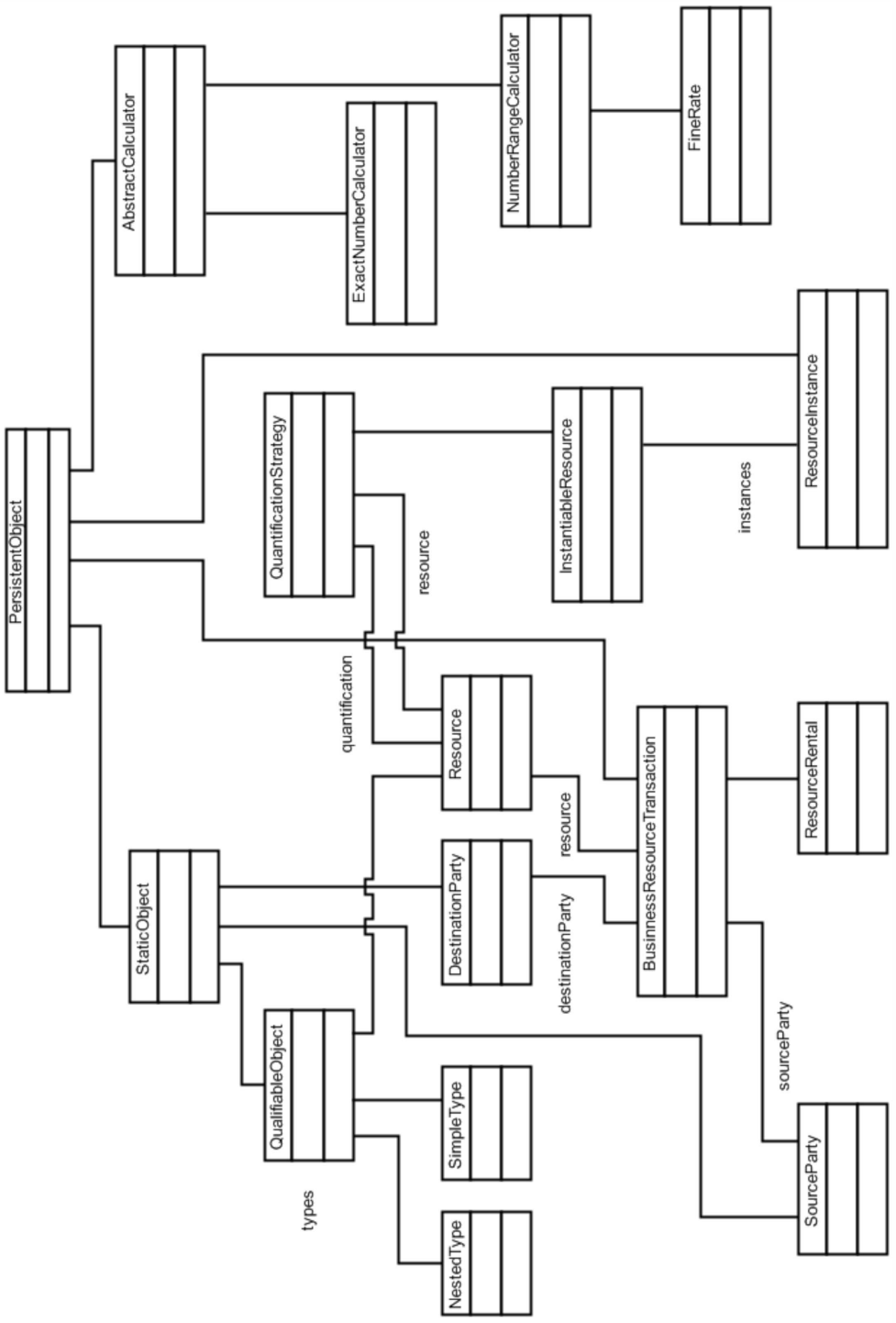


Figura 5.7: Diagrama de classes em nível de projeto do padrão LOCAR O RECURSO, obtido por meio da aplicação iterativa do padrão *Speculate About Design*

Observando os dois diagramas pode-se perceber que, devido ao nível de abstração das duas representações, há bastante diferença entre os conceitos documentados pela GRN e o que está implementado no framework GREN.

Observou-se que a aplicação do padrão *Speculate About Design* demanda muito tempo, é trabalhosa e propensa a erros. Durante sua utilização para obtenção do diagrama de classes do padrão LOCAR O RECURSO, nenhuma ferramenta foi utilizada, de forma que o diagrama hipotético e todos os outros foram feitos manualmente. Essa abordagem apresentou as seguintes desvantagens: os diagramas de classe produzidos são menos legíveis que os criados com o auxílio de ferramentas e a introdução de alterações é mais demorada. Com o intuito de evitar tais desvantagens, durante a engenharia reversa do padrão MANTER O RECURSO, a ferramenta JUDE (JUDE, 2008) foi empregada para criação dos diagramas de classes. Além disso, os artefatos da ferramenta foram produzidos sob controle de versões, conforme recomendado na seção *Trade-offs* do padrão (Demeyer et al., 2002), com o propósito de facilitar a "rastreabilidade" dos diagramas produzidos.

O padrão de projeto *Strategy* (Gamma et al., 1995) foi amplamente aplicado para implementação de alguns padrões no framework GREN, entre eles: LOCAR O RECURSO e MANTER O RECURSO. Isso facilitou a compreensão do projeto subjacente e aplicação do padrão *Speculate About Design*. De modo que, após identificada a utilização de um padrão de projeto, descobre-se o papel desempenhado por várias classes do diagrama sendo refinado.

A linguagem Smalltalk é dinamicamente tipada, o que dificultou a inferência dos tipos durante a realização das atividades de engenharia reversa que têm como principal fonte de informações o código do GREN, como, por exemplo, a aplicação dos padrões: *Read All the Code in One Hour* e *Speculate About Design*. As conseqüências dessa dificuldade, considerando as atividades de engenharia avante, são apresentadas na Subseção seguinte.

5.4.4 Inferência de Tipos

Inicialmente, o processo de reengenharia foi conduzido de maneira *test-last*. Entretanto, devido às diferenças entre as duas linguagens o código produzido necessitou ser modificado constantemente. Como mencionado, o fato da linguagem Smalltalk ser dinamicamente tipada dificulta, em alguns casos, a inferência de tipos. Essa informação sobre tipos é relevante durante a implementação da solução alvo. Dado que a linguagem Java™ é fortemente tipada, o que implica que toda variável antes de ser utilizada deve ser declarada e ter um tipo associado.

Informações sobre os tipos, que devem ser utilizados como parâmetros de um método, são facilmente obtidas consultando o código fonte do GREN, devido à convenção que foi utilizada para nomear os parâmetros. Mais especificamente, tipos de parâmetros cuja primeira letra do nome é uma vogal recebem o prefixo `an`, para os outros tipos o prefixo é `a` – esses prefixos, em inglês, significam um ou uma. Como pode ser observado no exemplo apresentado na Listagem 5.1, o método `fromItemTransaction:`

recebe como parâmetro instâncias da classe `TransactionItem` ou instâncias de subclasses dessa classe. Como o nome do tipo não começa com vogal o prefixo `a` é usado, desse modo o parâmetro é denominado: `aTransactionItem` (um item de transação, em inglês).

Listagem 5.1: Método da classe `BasicMaintenance`, protocolo `instance creation`

```
fromItemTransaction: aTransactionItem
        "creates a transaction from the corresponding transaction Item"

    | newTrans |
    newTrans := super fromItemTransaction: aTransactionItem.
    newTrans faultsPresented: aTransactionItem transaction faultsPresented.
    newTrans tasks: aTransactionItem transaction tasks.
    newTrans totalTasks: aTransactionItem transaction totalTasks.
    newTrans parts: aTransactionItem transaction parts.
    newTrans totalParts: aTransactionItem transaction totalParts.
    ^newTrans
```

Todavia, com exceção do caso anteriormente mencionado, nem todas as informações relacionadas aos tipos podiam ser extraídas precisamente. Algumas vezes, essas informações eram apenas conjecturas sobre os tipos que deveriam ser implementados. Nem sempre essas conjecturas eram confirmadas durante a implementação do GRENJ, o que acarretava em modificações, normalmente refatorações, com o propósito de alterar a hierarquia dos tipos.

A realização de "ajustes" sem a presença de testes resultava, comumente, na introdução de problemas e desperdício de tempo em sessões de depuração. A API do GRENJ também passava por várias refatorações, pois as convenções de nomenclatura e disposição dos parâmetros nas assinaturas dos métodos são bem diferentes entre as duas linguagens. Na linguagem Smalltalk os nomes dos métodos (também conhecidos como mensagens) podem ser separados pelos respectivos parâmetros. O método da Listagem 5.2 é denominado `rotateBy: around:` e seu primeiro parâmetro é um ângulo e o segundo é um vetor; `anAngle` e `aVector`, respectivamente. Adicionalmente, os métodos em Smalltalk não declaram um tipo de retorno, pois, a linguagem é dinamicamente tipada. O operador de retorno na linguagem Smalltalk é o ^ (Listagem 5.2). Métodos que possuem o nome semelhante ao do método da Listagem 5.2 são denominados *keyword messages* (Sharp, 1997). Na linguagem Java™ o nome dos métodos não pode ser dividido. Assim, durante a reengenharia do GREN essas decisões relacionadas à adaptação do nome dos métodos foram tomadas. A Listagem 5.3 exibe uma versão implementada em Java™ do método da Listagem 5.2. Na versão Java™ é necessário que o tipo de retorno e dos parâmetros sejam declarados. O nome do método também deve ser alterado, visto que somente `rotateBy`

ou `rotateByAround` não comunicam apropriadamente a finalidade do método (Listagem 5.3).

Listagem 5.2: Método implementado na linguagem Smalltalk

```
rotateBy: anAngle around: aVector
   | result |
   result := self computeAnswer.
   ^result
```

Listagem 5.3: Versão Java™ do método da Listagem 5.2

```
public Transformation rotateAround( float anAngle, Vector aVector ) {
   return new Transformation( this.computeAnswer() );
}
```

Os problemas mencionados anteriormente foram abordados aplicando TDD durante as atividades de engenharia avante. Com TDD evita-se que problemas sejam introduzidos durante as possíveis futuras alterações. Pois, os testes produzidos podem ser executados após as mudanças com o objetivo de certificar que o comportamento observável não foi modificado. O conceito de *programar por intenção* (Seção 4.2) auxiliou durante a implementação da API do GRENJ, apoiando a criação de métodos com nomes significativos.

5.4.5 Realização das Atividades de Engenharia Avante

O framework GRENJ foi implementado na linguagem Java™, versão 5.0, utilizando TDD. O IDE usado no desenvolvimento foi o Eclipse (Eclipse, 2007), versão 3.2.1. Além do Eclipse, também foram empregadas as seguintes ferramentas para apoiar a utilização do TDD: JUnit (JUnit, 2007)(Subseção 4.6.1), EasyMock (Apêndice B), EclEmma (EclEmma, 2008) (Subseção 5.5.1).

Para cada padrão que passou por reengenharia, uma lista de casos de teste foi criada com base nas informações obtidas por meio das atividades de engenharia reversa. Essa lista de casos de testes "guiou" a implementação dos padrões. Cada padrão foi considerado implementado somente após a implementação do último caso de teste e da respectiva funcionalidade que satisfaz esse teste. Ao final da implementação de cada padrão as versões Java™ dos métodos que passaram por reengenharia foram documentadas no *protocolo de reengenharia*.

5.4.6 Atividades de Engenharia Avante: Implementação do Padrão IDENTIFICAR AS TAREFAS DA MANUTENÇÃO

Com base na descrição da funcionalidade, fornecida na Subseção 5.4.2, a lista de testes da Tabela 5.3 foi produzida. É importante salientar que essa lista de testes só possui testes relacionados à classe `MaintenanceTask`. Pois, as outras classes do padrão já haviam passado por reengenharia, o que justifica a pequena quantidade de casos de teste.

Tabela 5.3: Lista de casos de teste para implementação da classe `MaintenanceTask`

Lista de Casos de Teste
A classe `MaintenanceTask` deve possuir atributos que representem o problema a ser resolvido, a descrição da mão-de-obra empregada para resolução do problema, a quantidade de horas gastas, custo, executor e a transação de manutenção ou cotação.
Deve ser possível criar instâncias de `MaintenanceTask` a partir de informações recuperadas do banco de dados.
Deve ser possível persistir as informações da classe `MaintenanceTask` no banco de dados (também deve ser possível remover essas informações persistidas).

Como pode ser observado na Figura 4.1, o primeiro passo no ciclo do TDD é a seleção do caso de teste a ser implementado. Essa escolha é subjetiva, entretanto, é recomendável que se selecione aquele cuja funcionalidade avaliada possa ser mais facilmente implementada (Beck, 2002). Assim, seleciona-se o primeiro caso de teste da Tabela 5.3. De acordo com esse caso de teste, a classe `MaintenanceTask` deve possuir atributos que armazenam as informações relevantes relacionadas a uma tarefa de manutenção. Desse modo, o teste apresentado na Listagem 5.4 foi implementado.

Listagem 5.4: Teste que avalia o valor *default* das informações mantidas por uma tarefa de manutenção

```
7  @Test
8  public void testAttributesDefaultValues() {
9      MaintenanceTask aMaintenanceTask = new MaintenanceTask();
10     Assert.assertNull( aMaintenanceTask.transaction );
11     Assert.assertEquals( "", aMaintenanceTask.problemToSolve );
12     Assert.assertEquals( "", aMaintenanceTask.laborDescription );
13     Assert.assertEquals( 0.0D, aMaintenanceTask.hoursSpent );
14     Assert.assertEquals( "R$ 0,00", aMaintenanceTask.cost.toString() );
15 }
```

Durante a implementação do caso de teste, o desenvolvedor *programa por inten-*

ção, ou seja, implementa o teste utilizando elementos (classes, métodos, etc.) que ainda não estão implementados. Assim, no teste exibido na Listagem 5.4 o desenvolvedor especifica que a classe `MaintenanceTask` possui um construtor que não recebe nenhum parâmetro e que ela tem os seguintes atributos públicos: `transaction`, `problemToSolve`, `laborDescription`, `hoursSpent` e `cost`. Nesse momento, o teste não compila, visto que a classe `MaintenanceTask` não está implementada. Assim, os seguintes passos são realizados para que o teste compile:

(i) implementação da classe `MaintenanceTask` como exibido na Listagem 5.5. Na linguagem Java™, caso nenhum construtor seja especificado, um construtor que não recebe nenhum parâmetro é fornecido. Porém, ainda é necessário implementar os atributos avaliados no teste da Listagem 5.4.

Listagem 5.5: Classe `MaintenanceTask`

```
1 package grenj.model;
2
3 public class MaintenanceTask {
4
5 }
```

(ii) implementação de um atributo que represente tanto transações de manutenção quanto cotações. Utiliza-se uma variável de referência do tipo `BasicMaintenance`, que no framework GREN e no GRENJ é a superclasse de `ResourceMaintenance` e `MaintenanceQuotation`, classes que representam transações de manutenção e cotações de manutenção, respectivamente. O atributo não precisa ser inicializado com `null`, dado que, na linguagem Java™, esse é o valor *default* para variáveis de referência não inicializadas (Listagem 5.6).

Listagem 5.6: Inclusão do atributo `transaction`

```
1 package grenj.model;
2
3 public class MaintenanceTask {
4    public BasicMaintenance transaction;
5 }
```

(iii) implementação de um atributo do tipo `String`, representando o problema a ser resolvido pela tarefa de manutenção. Como mencionado, variáveis de referência são inicializadas com `null`, assim é necessário que a variável `problemToSolve` seja inicializada com "" para que a avaliação na linha 11 do teste seja satisfeita.

A Listagem 5.7 ilustra o atributo recém-implementado; os outros trechos de código da classe são suprimidos.

Listagem 5.7: Atributo do tipo `String` implementado para representar o problema a ser resolvido pela tarefa de manutenção

```
5 public String problemToSolve = "";
```

(iv) implementação de mais um atributo do tipo `String`, denominado `laborDescription`, representando uma descrição da mão-de-obra envolvida. A Listagem 5.8 apresenta a implementação desse atributo; novamente os outros trechos de código da classe são omitidos.

Listagem 5.8: Atributo do tipo `String` implementado para representar a descrição da mão-de-obra

```
6 public String laborDescription = "";
```

(v) para representar a quantidade de horas gastas na tarefa de manutenção decidiu-se, durante a implementação do caso de teste, utilizar um tipo **double**. Dessa forma, `0.3` representa meia hora, `0.45` equivale à quarenta e cinco minutos e `1.0` uma hora. O framework GREN também implementa esse conceito utilizando números de ponto flutuante. Implementa-se, então, um variável de instância do tipo primitivo **double** denominada `hoursSpent` (Listagem 5.9). Essa variável não precisa ser inicializada, visto que tipos primitivos numéricos em Java™são inicializados com zero.

Listagem 5.9: Atributo do tipo **double** que representa o número de horas gastas para realização da tarefa de manutenção

```
7 public double hoursSpent;
```

(vi) acrescenta-se um atributo, denominado `cost`, que representa o custo da tarefa de manutenção. Como no framework GRENJ todos os valores financeiros são implementados utilizando a classe `Money` (Seção 5.7), `cost` é uma variável de referência desse tipo. A Listagem 5.10 mostra o código da classe `MaintenanceTask` após a inclusão de todos atributos requeridos.

Após os ajustes mencionados executa-se o teste apresentado na Listagem 5.4. O teste é satisfeito, no entanto, todos os atributos da classe `MaintenanceTask` são públicos, o que viola os princípios da programação orientada a objetos. Dado que o

próximo passo do ciclo do TDD é refatorar o código existente para eliminar *bad smells* a refatoração *Encapsulate Field* (Fowler et al., 1999) é aplicada. O IDE Eclipse automatiza essa refatoração, diminuindo o tempo necessário para sua realização. Essa refatoração altera os atributos públicos para privados e, com o propósito de manter o encapsulamento, introduz métodos *getters* e *setters* para consulta e alteração desses atributos, respectivamente. As alterações podem ser observadas no trecho de código da Listagem 5.11 – somente os métodos *get* e *set* do atributo `transaction` são exibidos na Listagem 5.11.

Listagem 5.10: Classe `MaintenanceTask` e os atributos adicionados

```
1  package grenj.model;
2
3  import java.util.Locale;
4  import grenj.util.Money;
5
6  public class MaintenanceTask {
7     public BasicMaintenance transaction;
8     public String problemToSolve = "";
9     public String laborDescription = "";
10    public double hoursSpent;
11    public Money cost = new Money( 0.0D, new Locale("pt", "BR") );
12 }
```

Listagem 5.11: Classe `MaintenanceTask` após a realização das refatorações

```
6  public class MaintenanceTask {
7     private BasicMaintenance transaction;
8     private String problemToSolve = "";
9     private String laborDescription = "";
10    private double hoursSpent;
11    private Money cost = new Money( 0.0D, new Locale("pt", "BR") );
12
13    public void setTransaction( BasicMaintenance transaction ) {
14       this.transaction = transaction;
15    }
16
17    public BasicMaintenance getTransaction() {
18       return transaction;
19    }
```

Como pode ser observado na Listagem 5.11, não é mais possível ter acesso aos atributos diretamente, só por meio dos respectivos métodos *get*. Dessa forma, o IDE Eclipse também modifica o código do teste da Listagem 5.4 de acordo com o exibido na Listagem 5.12. Fowler et al. (1999) ressaltam que mesmo após a realização de

refatorações automatizadas os testes devem ser executados. O teste mostrado na Listagem 5.12 é executado e continua sendo satisfeito, confirmando que as refatorações não introduziram nenhuma alteração no comportamento observável. Depois dessas mudanças, o caso de teste escolhido é considerado implementado e satisfeito.

Listagem 5.12: Código do teste após as refatorações

```
7  @Test
8  public void testAttributesDefaultValues() {
9      MaintenanceTask aMaintenanceTask = new MaintenanceTask();
10     Assert.assertNull( aMaintenanceTask.getTransaction() );
11     Assert.assertEquals( "", aMaintenanceTask.getProblemToSolve() );
12     Assert.assertEquals( "", aMaintenanceTask.getLaborDescription() );
13     Assert.assertEquals( 0.0D, aMaintenanceTask.getHoursSpent() );
14     Assert.assertEquals( "R$ 0,00",
15         aMaintenanceTask.getCost().toString() );
16 }
```

A implementação da classe `MaintenanceTask` prossegue com a escolha do segundo caso de teste (Tabela 5.3), que é implementado como na Listagem 5.13. Esse teste especifica que deve ser possível criar instâncias da classe `MaintenanceTask` com base em informações recuperadas de registros do banco de dados. Especificamente, no teste da Listagem 5.13 a variável de referência declarada na linha 21 será instanciada com as informações contidas em uma suposta tupla cujo campo `transaction` possui o valor 9.

Listagem 5.13: Teste que avalia a recuperação de tarefas de manutenção persistidas no banco de dados

```
19 @Test
20 public void testLoadFromDB() {
21     MaintenanceTask aMTask =
22         (MaintenanceTask) PersistentObject.load( "transaction = 9",
23         MaintenanceTask.class );
24     Assert.assertEquals( "load", aMTask.getProblemToSolve() );
25 }
```

Todo o "mecanismo" de persistência do GRENJ já havia sido implementado, e a classe que implementa a maior parte dessa funcionalidade é a `PersistentObject`. Tal classe provê métodos para criar instâncias com base em informações recuperadas do banco de dados e persistir as informações dessas instâncias no banco de dados. Para utilização adequada dessa funcionalidade é necessário que a classe `MaintenanceTask` seja uma subclasse de `PersistentObject`, o que implica em sobrescrever os seguintes métodos: `insertionFieldClause`, `insertionValueClause`,

`updateSetClause`, `updateWhereClause` e `deletionClause`. Alguns métodos na classe `PersistentObject` utilizam *reflection* para instanciação de classes com informações recuperadas do banco de dados. Esses métodos verificam se a classe passada como parâmetro possui um construtor que recebe como parâmetros instâncias das classes `java.sql.ResultSet` e `grenj.util.Index`. Dessa forma, todas as classes que têm informações persistidas e recuperadas do banco devem fornecer um construtor com essas características.

Alguns trechos da classe `MaintenanceTask`, após as alterações realizadas, são mostrados nas Listagem 5.14 e 5.15. Como pode ser observado na Listagem 5.15, os métodos que necessitam ser sobrescritos são, inicialmente, implementados como métodos *stub*.

Listagem 5.14: `MaintenanceTask` deve estender a classe `PersistentObject`

```
6 public class MaintenanceTask extends PersistentObject {
```

Listagem 5.15: O método `insertionFieldClause` é um dos métodos da classe `PersistentObject` que devem ser sobrescritos

```
56 @Override
57 public String insertionFieldClause() {
58    return null;
59 }
```

O construtor necessário é ilustrado na Listagem 5.16. Nesse construtor, as linhas 21, 23, 25, e 27 contêm instruções que obtêm as informações recuperadas do banco de dados e inicializam os atributos da classe com essas informações. Executam-se os testes. O teste recém-implementado não é satisfeito, pois, é necessário criar uma tabela na base de dados que possui o mesmo nome da classe, nesse caso: `MaintenanceTask`. O *script* MySQL utilizado para criação da tabela é ilustrado na Listagem 5.17. Para que o teste seja satisfeito, após a implementação do construtor da Listagem 5.16 e criação da tabela necessária utilizando o *script* da Listagem 5.17, deve-se inserir uma tupla cujo campo `transaction` tenha o valor igual a 9. Executam-se novamente os testes, depois da criação da tupla mencionada, todos os testes são satisfeitos. Como nenhuma atividade de refatoração é necessária prossegue-se, então, para implementação do caso de teste restante.

O código que implementa o último caso de teste é apresentado na Listagem 5.18. Os métodos `checkSave` (linha 36, Listagem 5.18) e `checkDelete` (linha 39, Listagem 5.18) foram definidos na classe de testes (`TestMaintenanceTask`). Eles acessam o banco de dados a fim de verificar se as informações foram devidamente persistidas

ou removidas, respectivamente. O código fonte desses métodos não é apresentado. Novamente a classe `PersistentObject` implementa grande parte da funcionalidade necessária para que o teste seja satisfeito. O desenvolvedor necessita somente especificar, na classe `MaintenanceTask`, utilizando os métodos `insertionFieldClause` e `insertionValueClause`, quais atributos são persistidos no banco de dados. O método `save` (linha 35, Listagem 5.18), definido na classe `PersistentObject`, acessa o banco de dados e persiste as informações especificadas no método `insertionValueClause`. O método `delete` (linha 38, Listagem 5.18), também definido na classe `PersistentObject`, remove a primeira tupla do banco de dados cujos campos correspondem a sentença especificada no método `deletionClause`.

Listagem 5.16: Implementação do construtor necessário para que instâncias da classe sejam inicializadas, usando *reflection*, com informações recuperadas do banco de dados

```
15 public MaintenanceTask( ResultSet result, Index anIndex ) {
16
17    this.transaction = null;
18    assert anIndex.getIndex() == 2;
19    try {
20
21       problemToSolve = result.getString( anIndex.getIndex() );
22       anIndex.incrementIndexByOne();
23       laborDescription = result.getString( anIndex.getIndex() );
24       anIndex.incrementIndexByOne();
25       hoursSpent = result.getDouble( anIndex.getIndex() );
26       anIndex.incrementIndexByOne();
27       cost = new Money( result.getDouble( anIndex.getIndex() ),
28          this.getDefaultLocale() );
29       anIndex.incrementIndexByOne();
30       executor = null;
```

Listagem 5.17: *Script* utilizado para criação da tabela

```
1 create table MaintenanceTask (
2    transaction integer not null,
3    problemToSolve char( 30 ),
4    laborDescription char( 40 ),
5    hoursSpent float,
6    cost float );
```

Aplicando TDD o desenvolvedor não fica restrito a realizar sempre pequenas alterações, como por exemplo constantes por variáveis e métodos. Caso a implementação de determinada funcionalidade avaliada pelos testes não seja complexa, ele pode forne-

cer a implementação óbvia – *Use Obvious Implementation* (Beck, 2002; Koskela, 2007) – implementando diretamente todo o código necessário para que o teste, criado anteriormente, seja satisfeito. Assim, os métodos `updateSetClause`, `updateWhereClause`, `insertionFieldClause`, `insertionValueClause`, e `deletionClause` foram implementados com a aplicação dessa técnica. As informações fornecidas por esses métodos possibilitam que os métodos `save` e `delete` exerçam suas funções apropriadamente. Somente o código correspondente à implementação do método `insertionFieldClause` é exibido na Listagem 5.19.

Listagem 5.18: Teste que avalia se os dados são corretamente persistidos e removidos do banco de dados

```
27 @Test
28 public void testSaveAndDelete() {
29     MaintenanceTask aMTask = new MaintenanceTask();
30
31     aMTask.setProblemToSolve( "testing" );
32     aMTask.setHourSpent( 1.5 );
33
34     //save
35     Assert.assertEquals( 1, aMTask.save() );
36     Assert.assertTrue( checkSave( aMTask ) );
37
38     Assert.assertEquals( 1, aMTask.delete() );
39     Assert.assertTrue( checkDelete( aMTask ) );
40 }
```

Listagem 5.19: Implementação do método `insertionFieldClause`

```
56 @Override
57 public String insertionFieldClause() {
58     StringBuilder insertionFieldClause = new StringBuilder();
59     insertionFieldClause.append( "transaction, problemToSolve,"
60         + "laborDescription, hoursSpent, cost" );
61     if ( this.hasExecutor() ) {
62         insertionFieldClause.append( ", executor" );
63     }
64
65     return insertionFieldClause.toString();
66 }
```

Depois de implementar todos os métodos necessários, executam-se os testes. Os três testes são satisfeitos. Nenhum *bad smell* foi inserido durante a implementação dos métodos, assim nenhuma refatoração é realizada. Uma decisão de projeto tomada durante a implementação do framework GRENJ é que todas as classes que represen-

tam papéis importantes nos padrões, como por exemplo, recurso, manutenção, locação e cliente são implementados como classes abstratas. Dessa forma, o usuário é obrigado a criar subclasses das classes principais do framework durante a instanciação das aplicações, o que reforça que o GRENJ é um framework caixa-branca. Desse modo, a classe `MaintenanceTask` foi refatorada para se tornar uma classe abstrata, o que implica que as instâncias de `MaintenanceTask` criadas no código dos testes tiveram que ser substituídas por objetos *stub* e o nome da tabela criada deve ser alterado de acordo com o nome do objeto *stub*. Realizadas as alterações, os testes foram novamente executados e satisfeitos. Como todos os casos de teste foram implementados pode se considerar que todas as classes do padrão IDENTIFICAR AS TAREFAS DA MANUTENÇÃO foram devidamente implementadas.

5.5 Benefícios do Uso de Testes Automatizados durante a Implementação do GRENJ

As dificuldades anteriormente mencionadas foram mitigadas por meio da aplicação do TDD (Beck, 2002), visto que durante a criação dos testes – *programando por intenção* – decisões sobre a API são consideradas antes da implementação da funcionalidade. Além disso, a criação de testes automatizados desde o início da implementação possibilitou que as alterações, como por exemplo refatorações e regras de unificação (Cortes et al., 2003), fossem realizadas de maneira mais apropriada, evitando a inserção de efeitos colaterais inesperados. O IDE Eclipse automatiza algumas refatorações como: *Extract Method*, *Extract Superclass*, *Encapsulate Field*, entre outras. No entanto, mesmo após a realização de refatorações com o apoio de ferramentas, é recomendável a presença de testes a fim de avaliar a introdução de problemas (Fowler et al., 1999). Os testes executados durante as alterações são denominados testes de regressão (Koskela, 2007). Esses testes foram implementados originalmente como testes de unidade e, posteriormente, adicionados no conjunto de testes de regressão (Myers et al., 2004). Um problema de regressão ocorre quando alguma funcionalidade do sistema deixa de realizar o comportamento esperado devido às alterações realizadas. Alguns dos benefícios provenientes da existência de testes automatizados, principalmente no contexto de reengenharia, são (Demeyer et al., 2002):

- Documentam como os artefatos de software pertencentes ao sistema devem ser usados. Em contraste com a "documentação escrita" (*written documentation*), os testes automatizados são descrições sempre atualizadas e que refletem o estado atual do sistema.
- Representam uma forma tangível de confiança na funcionalidade do sistema.

5.5.1 O *Plugin* EclEmma

Considera-se um trecho de código coberto pelos testes quando, pelo menos, um teste "exercita" esse trecho (Koskela, 2007; Meszaros, 2007). Algumas ferramentas podem ser utilizadas para facilitar a identificação dos trechos de código não cobertos pelos testes, por exemplo: Cobertura[2], NoUnit[3], Clover[4], entre outras. Durante a implementação do framework GRENJ o EclEmma foi utilizado para identificação e visualização dessas parcelas do código.

O EclEmma é um *plugin* para o IDE Eclipse que realiza análise de cobertura de código. Esse *plugin* é distribuído sob a *Eclipse Public License*[5]. O *plugin*, após a execução dos testes, classifica e destaca (*highlight*) cada linha de código da seguinte forma:

Verde: linha coberta por um ou mais testes;

Amarela: linha parcialmente coberta pelos testes;

Vermelha: linha não exercitada pelos testes.

Essas informações, destacadas pelo EclEmma, foram utilizadas para aprimorar a cobertura dos testes implementados durante o desenvolvimento do framework GRENJ e também para apoiar atividades de refatoração, de forma que os trechos de código não cobertos pelos testes só eram refatorados após a introdução dos testes necessários. A Figura 5.8 ilustra como linhas cobertas (verdes) e não cobertas (vermelhas) pelos testes são visualizas com o auxílio do EclEmma. Também é possível visualizar a porcentagem de linhas cobertas pelos testes em cada classe, pacote ou considerando todo o projeto. A Figura 5.9 apresenta como o *plugin* organiza tais informações, como pode ser observado exibe-se a porcentagem de linhas cobertas para o pacote `grenj.model` e para cada classe pertencente ao pacote.

5.5.2 Organização dos Testes

A criação dos testes durante a implementação aumentou consideravelmente o número de arquivos fonte que compõe o framework GRENJ. Para gerenciar tais arquivos a seguinte hierarquia de pacotes (*packages*) foi utilizada: cada pacote do GRENJ possui um subpacote, denominado *test*, que contém os testes das classes do pacote "principal". Há dois tipos de classes nesses subpacotes, as que armazenam os testes são denominadas utilizando o prefixo *Test* e as que são implementações de objetos stub

[2]http://cobertura.sourceforge.net/
[3]http://nounit.sourceforge.net/
[4]http://www.atlassian.com/software/clover/
[5]http://www.eclipse.org/legal/epl-v10.html

usam o prefixo *Testable*. Essa organização e convenção de nomenclatura facilita a execução dos testes tanto individualmente quanto por pacote utilizando ferramentas como o Ant (Apache Ant, 2008). Os três pacotes mais relevantes do framework GRENJ são ilustrados na Figura 5.10 e, como exibido, cada pacote possui um pacote aninhado denominado *test*.

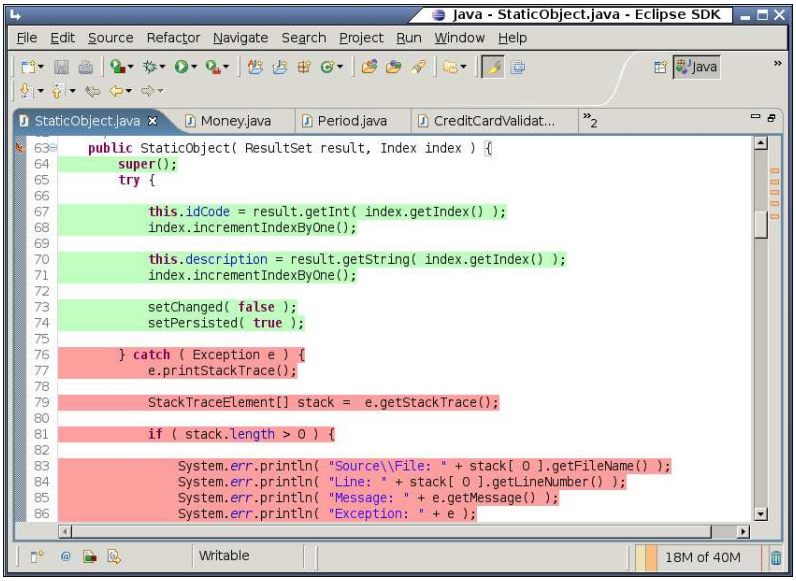
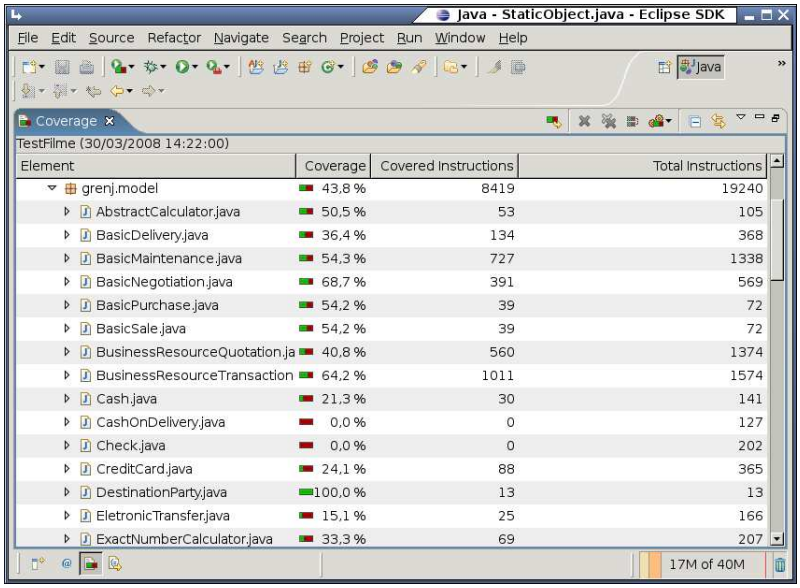


Figura 5.8: Linhas cobertas e não cobertas pelos testes destacadas pelo *plugin* EclEmma



Element	Coverage	Covered Instructions	Total Instructions
grenj.model	43,8 %	8419	19240
AbstractCalculator.java	50,5 %	53	105
BasicDelivery.java	36,4 %	134	368
BasicMaintenance.java	54,3 %	727	1338
BasicNegotiation.java	68,7 %	391	569
BasicPurchase.java	54,2 %	39	72
BasicSale.java	54,2 %	39	72
BusinessResourceQuotation.ja	40,8 %	560	1374
BusinessResourceTransaction	64,2 %	1011	1574
Cash.java	21,3 %	30	141
CashOnDelivery.java	0,0 %	0	127
Check.java	0,0 %	0	202
CreditCard.java	24,1 %	88	365
DestinationParty.java	100,0 %	13	13
EletronicTransfer.java	15,1 %	25	166
ExactNumberCalculator.java	33,3 %	69	207

Figura 5.9: Informações sobre a porcentagem de linhas de código cobertas pelos testes considerando o pacote `grenj.model`

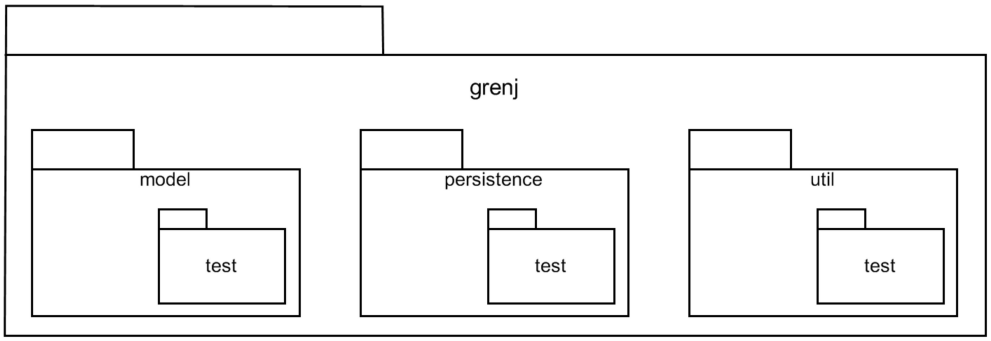


Figura 5.10: Hierarquia dos principais pacotes do framework GRENJ

5.6 Convenção Utilizada Durante a Implementação do GRENJ

A convenção utilizada para denominar métodos e variáveis na linguagem Smalltalk é diferente da empregada na linguagem Java™ (Sun Microsystems, Inc., 1997). Por exemplo, métodos *getters* e *setters* (também conhecidos como *accessor methods*) na convenção utilizada na linguagem Smalltalk, comumente, recebem o mesmo nome da variável que eles devem retornar ou ajustar, respectivamente (Sharp, 1997). Na linguagem Java™métodos desse tipo empregam o prefixo `get` ou `set`, respectivamente, seguido pelo nome da variável sendo consultada ou alterada. Manter a convenção utilizada na linguagem Smalltalk facilitaria a compreensão da solução alvo por parte dos indivíduos já familiarizados com a solução legada. Porém, poderia dificultar a compreensão, manutenção e instanciação de aplicações utilizando a solução alvo. Assim, optou-se por utilizar a convenção de codificação, definida para a linguagem Java™, proposta por Vermeulen et al. (2000). O que justifica a alteração do nome de alguns métodos gancho, entre outras.

5.7 Aperfeiçoamentos Introduzidos no Framework GRENJ

Algumas funções realizadas pelo framework GRENJ como comercialização, aluguel e manutenção de recursos envolvem abundante manipulação de valores monetários. Na linguagem Java™não há nenhuma classe que represente valores monetários e os tipos primitivos numéricos (**byte**, **short**, **int**, **long**, **float** e **double**) também não são adequados. Dessa forma, uma classe denominada `Money`, pertencente ao pacote `grenj.util`, foi implementada com esse propósito. Essa classe possibilita que valores monetários de precisão arbitrária sejam criados, pois, internamente, a classe `java.math.BigDecimal` é utilizada para armazenar tais valores.

Uma das vantagens de se utilizar a classe `BigDecimal` é que a precisão é mantida em operações aritméticas, evitando problemas de arredondamento. A classe `Money` é,

além disso, uma aplicação do padrão *Value Object* (Fowler, 2002), o que evita problemas de *aliasing*. Um problema de *aliasing* ocorre quando dois objetos compartilham uma referência para um terceiro, se o valor dessa referência é alterado por um dos dois objetos ela deixa de ser consistente para o outro. Usando a classe `Money` também é possível que os valores sejam formatados e representados em várias moedas.

O framework GRENJ também gerencia a devolução das locações, o vencimento de parcelas, entre outras funções que manipulam datas. A classe `grenj.util.Period` foi criada justamente para facilitar essa manipulação, e tanto ela quanto a classe `Money` podem ser usadas independentemente do framework GRENJ.

A classe `ConnectionManager` do framework GRENJ, que gerencia as conexões com o banco de dados e possui uma versão equivalente no framework GREN, teve sua implementação aprimorada. Ela foi implementada como uma aplicação do padrão de projeto *Singleton* (Gamma et al., 1995) utilizando *lazy instantiation* (Bloch, 2001), de forma que somente uma instância da classe é criada em tempo de execução e apenas quando tal instância é requisitada, poupando recursos computacionais.

5.8 Considerações Finais

O processo de reengenharia do framework GREN para linguagem Java foi, inicialmente, realizado de maneira iterativa, porém, *test-last*. Isso dificultava a realização de refatorações no código recém-implementado devido a ausência de testes automatizados. Essas alterações eram necessárias principalmente devido às diferenças existentes entre as duas linguagens sendo uma dinamicamente tipada e a outra fortemente tipada. Uma das vantagens do processo ser iterativo é que o aprendizado, obtido em uma iteração, pôde ser metodicamente usado para aprimorar as iterações subseqüentes, melhorando assim a efetividade do processo de reengenharia proposto. Isto resultou na aplicação do TDD, auxiliou o projeto da API da solução alvo e proporcionou a criação de um consistente conjunto de testes de regressão, facilitando atividades de refatoração. Os dois primeiros padrões da GRN implementados no GREN passaram por reengenharia aplicando os padrões de engenharia reversa mencionados e foram implementados de maneira *test-last*. Os padrões posteriores foram implementados de maneira *test-first*. O número de testes implementados utilizando a abordagem *test-first* (TDD) foi maior em relação à abordagem *test-last*.

Foi necessária a criação de diagramas de classe para auxiliar a compreensão somente dos seguintes padrões: LOCAR O RECURSO e MANTER O RECURSO. O padrão *Speculate About Design* foi aplicado para tal.

Os padrões de projeto implementados no framework GREN auxiliaram a compreensão da funcionalidade, dado que cada padrão carrega consigo sua semântica. O padrão de projeto *Strategy* (Gamma et al., 1995) é exaustivamente aplicado no fra-

mework GREN. A utilização desse padrão foi mantida na implementação do GRENJ.

Este Capítulo também apresentou vários benefícios da presença de testes automatizados no contexto de atividades de reengenharia. Justificou-se também a convenção de codificação utilizada durante a implementação do framework GRENJ.

CAPÍTULO
6

O Processo de Instanciação de Aplicações Utilizando o Framework GRENJ

6.1 Considerações Iniciais

Desenvolvedores interessados em utilizar frameworks caixa-branca devem conhecer detalhadamente a estrutura interna desses frameworks (Fayad e Schmidt, 1997). As aplicações instanciadas por meio desse tipo de framework tendem a ser fortemente acopladas com os detalhes de implementação e com a hierarquia de classes do framework. Assim, apesar de aprimorar a qualidade e reduzir o esforço necessário para produção de aplicações, a utilização de frameworks caixa-branca é complexa.

De acordo com Braga (2002b), caso uma linguagem de padrões tenha sido utilizada como base para o desenvolvimento de um framework, essa linguagem de padrões pode apoiar o processo de instanciação de aplicações. Visto que o GRENJ é equivalente ao GREN que foi criado com base na GRN, porém implementado na linguagem Java™, a GRN pode ser usada para "guiar" a instanciação de aplicações usando esses dois frameworks. Neste Capítulo descreve-se a abordagem recomendada para instanciação de uma aplicação, pertencente ao domínio de gestão de recursos de negócios, utilizando o framework GRENJ e a GRN. A abordagem é exemplificada por meio da instanciação de uma aplicação que satisfaz os requisitos propostos para uma locadora fictícia de DVDs. A Seção 6.2 descreve os passos necessários para modelagem da aplicação usando a GRN, a realização do "mapeamento" entre os elementos dessa

modelagem e as classes do GRENJ e a implementação da aplicação empregando TDD. Na Seção 6.3 são apresentadas as considerações finais.

6.2 Instanciação de Aplicações Utilizando o Framework GRENJ e a GRN

O processo para instanciação de aplicações utilizando o framework GRENJ e a linguagem de padrões GRN é composto por quatro etapas: análise da aplicação, mapeamento entre o modelo de análise e o framework, implementação das classes específicas e teste do sistema resultante. Essas etapas são descritas a seguir e exemplificadas por meio da instanciação de uma aplicação de locadora de DVDs fictícia. A Subseção 6.2.1 trata da análise da aplicação, a Subseção 6.2.2 apresenta o mapeamento entre o modelo de análise da aplicação e o framework GRENJ, a Subseção 6.2.3 exibe a implementação das classes da aplicação usando o GRENJ e a Subseção 6.2.4 é relacionada à avaliação da aplicação resultante.

6.2.1 Análise da Aplicação

Nessa primeira etapa, os requisitos da aplicação, exibidos na Tabela 6.1, são analisados com base na GRN, para verificar se tal aplicação pode ser modelada usando a GRN. Para o exemplo em questão, uma locadora fictícia de DVDs, essa condição é satisfeita e, gradualmente, a aplicação será modelada utilizando os padrões da GRN.

Como o primeiro padrão da GRN é IDENTIFICAR O RECURSO, a partir dos requisitos da Tabela 6.1, verificou-se quais são os possíveis candidatos a recurso, considerando as seções *Contexto*, *Problema* e *Influências* desse padrão. O diagrama de classes em nível de análise desse padrão, fornecido pela GRN, pode ser visualizado na Figura 3.2. Visto que um conceito só pode ser modelado como recurso se estiver envolvido em pelo menos uma transação, tais como venda, locação, manutenção e reserva, o único recurso identificado foi *Filme*. Ainda considerando o primeiro padrão da GRN, *Filme* pode ser classificado tanto por *Categoria* quanto por *Gênero*, o que resultou na introdução de mais duas classes no diagrama da aplicação sendo modelada.

O segundo padrão da GRN determina a forma de quantificação do recurso. Como para cada *Filme* podem existir várias cópias, é necessário ter controle sobre essas instâncias específicas, pois elas são negociadas individualmente. Quando deve-se distinguir entre instâncias do recurso aplica-se o subpadrão RECURSO INSTANCIÁVEL (Braga, 2002a), ilustrado na Figura 6.1. A aplicação de tal subpadrão implica na introdução de mais uma classe no diagrama da aplicação que está sendo modelado, a classe DVD, que mantém informações sobre o número da instância (DVD), localização e disponibilidade. Essa classe desempenha o papel da classe *Instância do Recurso* no

subpadrão referido. Como não há necessidade de manter informações sobre o armazenamento de DVDs (recurso), o terceiro e último padrão do primeiro grupo da GRN, que trata justamente esse ponto, não é aplicado.

Tabela 6.1: Requisitos da aplicação

#	**Descrição**
1	A locadora realiza o aluguel de DVDs de filmes que podem ter uma ou mais cópias (DVDs).
2	Cada filme possui um código, título e ano.
3	Cada DVD possui um código que identifica sua posição na prateleira, informação que indica se está disponível ou alugado e o título do filme nele contido.
4	Os filmes são classificados por categoria, que indica o valor diário do aluguel desse DVD.
5	Os filmes também são classificados por gênero (comédia, terror, ação, drama, etc.).
6	Os DVDs são alugados para os clientes cadastrados da locadora. As informações que o sistema deve manter sobre os clientes são: código, nome, telefone e cpf.
7	As informações de locação são: código, a data de locação, a data de devolução prevista, código do cliente, DVDs alugados, data de devolução efetiva e o valor. Um cliente pode alugar mais de um DVD em uma mesma locação e deve devolver todos no mesmo instante. A data de devolução prevista é de um dia para cada DVD alugado em relação à data de locação. O valor da locação varia de acordo com a soma dos valores de cada DVD alugado.
8	Se os DVDs não forem devolvidos na data de devolução prevista, o cliente deve pagar multa. O valor da multa é um valor fixo multiplicado pelos dias de atraso na devolução dos DVDs.

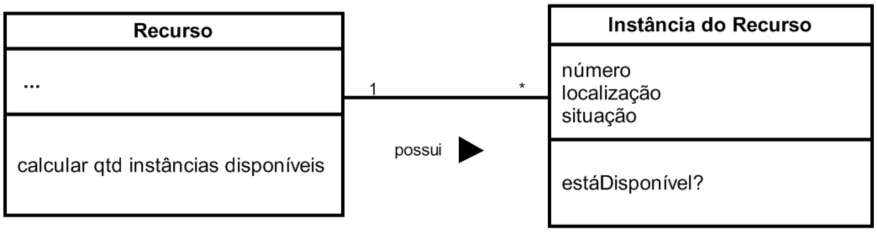


Figura 6.1: Padrão RECURSO INSTANCIÁVEL; extraído de Braga (2002a)

A Tabela 6.2 mostra os atributos que foram inseridos em cada classe. O diagrama de classes parcial da aplicação de locadora de DVDs, após a análise de todos os padrões do primeiro grupo e aplicação de alguns deles, é exibido na Figura 6.2. O formato utilizado no interior das setas é *"P#n: papel"* , onde *"n"* é o número do padrão

na GRN e *"papel"* é o papel desempenhado pela classe no padrão subjacente; esse formato é baseado no utilizado por Braga (2002b).

Tabela 6.2: Classes e atributos adicionados após a aplicação do primeiro grupo de padrões da GRN

Classe	Papel na GRN	Atributos Adicionados
Filme	*Recurso*	ano
Categoria	*Tipo de Recurso*	valor
Gênero	*Tipo de Recurso*	–
DVD	*Instância do Recurso*	–

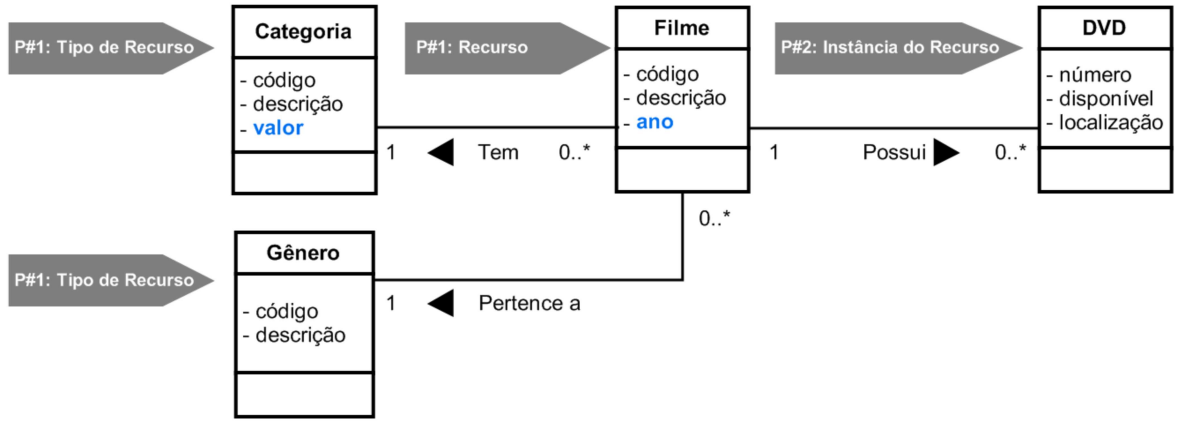


Figura 6.2: Diagrama de classes parcial da aplicação de locadora de DVDs

O segundo grupo de padrões da GRN é composto por padrões responsáveis pelo gerenciamento das transações envolvendo os recursos. Desse grupo apenas o padrão que aborda o aluguel de recursos, quarto padrão da GRN denominado LOCAR O RECURSO (Braga, 2002a), precisou ser aplicado. O diagrama de classes em nível de análise desse padrão é exibido na Figura 5.6. A aplicação desse padrão acarreta a introdução de mais três classes ao diagrama de classes da aplicação: *Locação*, *Cliente* e *Multa*. *Locação* representa a transação de mesmo nome que é realizada toda vez que os clientes desejam locar um ou mais DVDs; a classe *Cliente* representa os interessados nessas transações; *Multa* trata das tarifas cobradas quando ocorrem atrasos na devolução dos DVDs envolvidos em uma locação. A Tabela 6.3 exibe os atributos que foram inseridos em cada classe.

O terceiro grupo da GRN é formado por padrões que abordam os detalhes relacionados às transações. Esse grupo é constituído por cinco padrões. Dois deles são aplicáveis somente quando atividades de manutenção foram necessárias. Como os requisitos da locadora de DVD não abrangem nenhum tipo de manutenção, esses dois

padrões não foram considerados. Analisando os outros padrões, somente o padrão ITEMIZAR A TRANSAÇÃO DO RECURSO (Braga, 2002a) foi aplicado. O diagrama de classes desse padrão, fornecido pela GRN, é ilustrado na Figura 3.5. O padrão permite que uma ou mais instâncias sejam vinculadas a uma transação, no caso do sistema em desenvolvimento isso é representado pelo cliente que pode alugar um ou mais DVDs. Sua aplicação resulta na introdução da classe *ItemLocacao* ao diagrama de classes da aplicação. Essa classe é responsável por encapsular os DVDs relacionados a uma locação. Nenhum atributo foi adicionado à classe *ItemLocacao*, conforme pode ser visto na Tabela 6.4. O diagrama de classes da aplicação é ilustrado na Figura 6.3.

As Tabelas 6.2 e 6.3 exibem os atributos adicionados ao modelo com a aplicação da GRN para atender aos requisitos da aplicação, como sugere o processo proposto por Braga (2002b).

Tabela 6.3: Classes e atributos adicionados após a aplicação do segundo grupo de padrões da GRN

Classe	Papel na GRN	Atributos Adicionados
Locação	*Locação do Recurso*	–
Cliente	*Destino*	telefone, cpf
Multa	*Taxa de Multa*	–

Tabela 6.4: Classes e atributos adicionados após a aplicação do terceiro grupo de padrões da GRN

Classe	Papel na GRN	Atributos Adicionados
ItemLocacao	*Item da Transação*	–

6.2.2 Mapeamento entre o Modelo de Análise e o Framework

O objetivo desta etapa é produzir o "mapeamento" entre o diagrama de classes criado na etapa anterior e as classes do framework que devem ser estendidas. Essa etapa é realizada visto que não há uma relação biunívoca entre as classes do framework e as classes do padrão em nível de análise. Um dos motivos é que novas classes podem ser inseridas durante a implementação do padrão para introduzir mais flexibilidade ou para abordar outros problemas em nível de projeto. Além disso, esse "mapeamento" auxilia na identificação das classes que devem ser estendidas durante a instanciação da aplicação, já que a maioria das classes do framework é abstrata.

A Figura 6.4 ilustra um possível cenário de instanciação, no qual as classes *ClasseF1*, *ClasseF2* e *ClasseF3* são estendidas por classes concretas da aplicação: *Clas-*

seA1, *ClasseA2* e *ClasseA3*. As classes *ClasseA2* e *ClasseA3* adicionam alguns atributos, o *método3_3* é sobrescrito na classe *ClasseA3*.

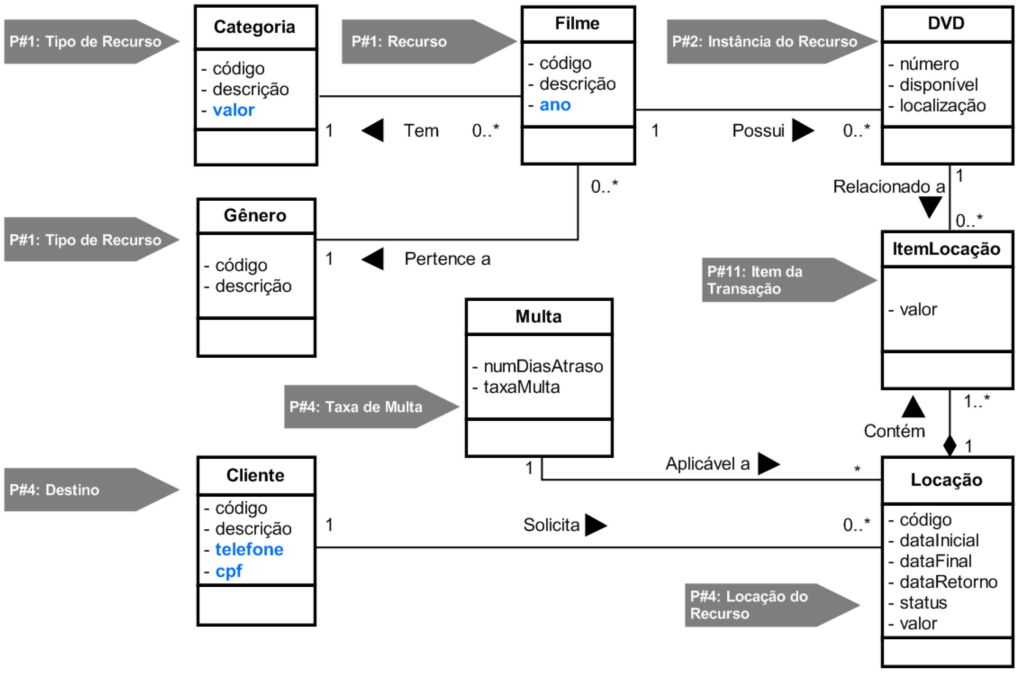


Figura 6.3: Diagrama de classes da aplicação de locadora de DVDs

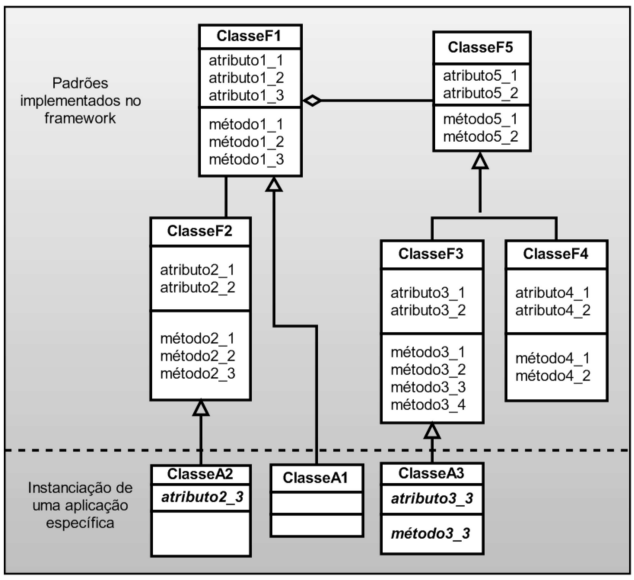


Figura 6.4: Instanciação do framework para uma aplicação específica

Para que esse "mapeamento" seja possível, o relacionamento entre os padrões da linguagem de padrões e as classes do framework deve estar adequadamente docu-

mentado. O *"cookbook"* do GREN documenta esse mapeamento entre as classes da GRN e as classes do framework que devem ser estendidas. Dado que a nomenclatura das classes utilizadas no framework GREN foi mantida no framework GRENJ, esse documento pode ser utilizado para auxiliar na identificação dessas classes durante a instanciação de aplicações usando o GRENJ. O "mapeamento" entre as classes da aplicação e as respectivas classes do GRENJ estendidas por elas, bem como os atributos adicionados são mostrados na Tabela 6.5. A próxima Subseção aborda a implementação dessas classes.

Tabela 6.5: Mapeamento entre as classes que devem ser criadas e as respectivas superclasses do GRENJ

Classe da Aplicação	**Superclasse do GRENJ**	**Atributos Adicionados**
Filme	`Resource`	ano
Categoria	`SimpleType`	valor
Gênero	`SimpleType`	–
DVD	`ResourceInstance`	–
Locação	`ResourceRental`	–
Cliente	`DestinationParty`	telefone, cpf
Multa	`FineRate`	–
ItemLocacao	`TransactionItem`	–

6.2.3 Implementação das Classes Específicas

Nesta etapa as classes da aplicação são implementadas. Essas classes estendem as classes do framework, conforme o "mapeamento" mostrado na Tabela 6.5. As customizações necessárias como adição de novos atributos e métodos são introduzidas nas classes da aplicação. Porém, durante a implementação dessas classes alguns detalhes específicos da linguagem Java™e do framework GRENJ devem ser considerados.

Classes que não adicionam nenhum atributo necessitam estender a classe abstrata do GRENJ indicada no "mapeamento" e fornecer dois construtores. Por exemplo, a classe *Multa* deve estender a classe `FineRate`, como indicado na Tabela 6.5. Um dos construtores não deve receber nenhum parâmetro; e como nenhum atributo foi adicionado, nada é inicializado nesse construtor. A única instrução possível nesse construtor é uma invocação a **super** (também sem nenhum parâmetro). No entanto, essa instrução é opcional, dado que ela é implicitamente inserida pela linguagem Java™, caso não seja provida. O outro construtor que deve ser implementado recebe dois parâmetros, um do tipo `java.sql.ResultSet` e outro do tipo `grenj.util.Index`. Esses dois parâmetros são utilizados para manter as informações recuperadas do banco de dados e auxiliar na recuperação dessas informações, respectivamente. A

única instrução no corpo desse construtor deve ser uma invocação ao construtor da superclasse. A implementação da classe *Multa* é ilustrada na Listagem 6.1; como pode ser observado, essa classe estende a classe `FineRate` e possui os dois construtores obrigatórios. A classe `grenj.persistence.PersistentObject`, que contém os métodos que inicializam instâncias com valores extraídos do banco de dados, busca por esses dois construtores utilizando *reflection*. Caso eles não sejam encontrados o framework "lança" (*throws*) uma `java.lang.NoSuchMethodException`.

Listagem 6.1: Implementação da classe *Multa*

```
1  import java.sql.ResultSet;
2
3  import grenj.model.FineRate;
4  import grenj.util.Index;
5  //estende a superclasse indicada no "mapeamento"
6  public class TaxaDeMulta extends FineRate {
7
8      //necessary
9      public TaxaDeMulta() {
10         super();
11     }
12
13     //necessary
14     public TaxaDeMulta( ResultSet result, Index anIndex ) {
15       super( result, anIndex );
16     }
17
18 }
```

Classes que introduzem atributos, além dos dois construtores obrigatórios, devem implementar métodos *getters* e *setters* para cada atributo adicionado e sobrescrever os métodos `insertionFieldClause`, `insertionValueClause` e `updateSetClause`. Os métodos gancho herdados da superclasse do framework também devem ser sobrescritos de maneira adequada, a quantidade de métodos gancho que deve ser sobrescrita depende das classes estendidas. Informações sobre como cada método gancho deve ser sobrescrito são fornecidas pela documentação do GRENJ. Os testes implementados durante o desenvolvimento do GRENJ também contêm exemplos de instanciação, assim esses testes podem ser consultados a fim de elucidar detalhes específicos sobre como determinados métodos gancho devem ser sobrescritos.

Os *scripts* utilizados para geração das tabelas do banco de dados são fornecidos pelo *"cookbook"* do GREN. Esses *scripts* são alterados dependendo de como determinados métodos gancho são implementados e dependendo de quais padrões são aplicados. No *"cookbook"* são detalhadas todas as alterações que devem ser realizadas nesses *scripts*. A seguir dois exemplos são utilizados para exemplificar os ajustes

que devem ser realizados nos *scripts*, de acordo com os padrões aplicados e atributos inseridos nas classes da aplicação. No primeiro exemplo, o *script* deve ser alterado considerando os padrões aplicados, conforme exibido na Listagem 6.2. As observações (i), (ii) e (iii) referem-se aos campos opcionais que devem ser inseridos se:

(i) o subpadrão RECURSO INSTANCIÁVEL tiver sido utilizado.

(ii) o subpadrão RECURSO MENSURÁVEL tiver sido utilizado.

(iii) o subpadrão RECURSO EM LOTES tiver sido utilizado.

Listagem 6.2: *Script* usado para criação de tabelas utilizadas na persistência de informações relacionadas às subclasses da classe `TransactionItem`

```
1 create table TransactionItem (
2    transaction integer,
3    resource integer,
4    value float,
5    instanceCode char( 10 ),   #(i)
6    quantity float,            #(ii)
7    lotNumber char( 20 ) ); ##(iii)
```

No *script* utilizado para criação de tabelas referentes a subclasses de `DestinationParty`, exibido na Listagem 6.3, não há nenhuma variabilidade decorrente da aplicação de padrões. No entanto, se atributos foram introduzidos nas subclasses o *script* deve ser alterado. Por exemplo, o *script* utilizado para criação da tabela da classe *Cliente* é o ilustrado na Listagem 6.4; no qual foram adicionados os campos *telefone* e *cpf*. É importante salientar que o nome da tabela no banco de dados deve ser igual ao nome da classe criada, pois o framework GRENJ utiliza *reflection* para o obter o nome da classe e procurar a tabela correspondente. Somente o nome da classe é considerado, dessa forma duas classes que devem ter suas informações persistidas, mesmo pertencendo a pacotes diferentes, não devem possuir o mesmo nome.

Listagem 6.3: *Script* usado para criação de tabelas utilizadas na persistência de informações relacionadas às subclasses da classe `DestinationParty`

```
1 create table DestinationParty (
2    idCode integer not null,
3    description char( 35 ) );
```

Ao se considerar todas as classes que devem ser estendidas para implementação da aplicação da locadora de DVDs, a classe `ResourceRental` é a que possui a maior

quantidade de métodos gancho que devem ser sobrescritos, exatamente dezoito métodos. Entre eles o método `getResourceClass`, que indica qual classe representa os recursos que são transacionados. Para implementação da aplicação sendo considerada, a classe *Locação* deve sobrescrever esse método como apresentado na Listagem 6.5, retornando um *metaobjeto* (*metaobject*) (Forman e Forman, 2004) usado para representar a classe *Filme*.

Listagem 6.4: *Script* utilizado para criação da tabela que armazena informações sobre a classe `Cliente` da aplicação de locadora

```
1 create table Cliente (
2    idCode integer not null,
3    description char( 35 ),
4    telefone char( 16 ),
5    cpf char(15) );
```

Listagem 6.5: Método `getResourceClass` da classe *Locação*

```
1 @Override
2 public Class< ? extends Resource> getResourceClass() {
3
4    return Filme.class;
5 }
```

Todos os métodos do framework GRENJ que devem ser sobrescritos a fim de retornar *metaobjetos*, representando as abstrações utilizadas nas aplicações que estão sendo instanciadas, empregam *generics*. Mais especificamente, *generics* são um aprimoramento introduzido na sintaxe da linguagem Java™, a partir da versão 5.0, que permite que as classes sejam "especializadas" para um ou mais tipos. Assim, como o método da Listagem 6.5 indica que somente *metaobjetos* relacionados às classes que estendem `Resource` devem ser retornados, se outro tipo de retorno for considerado, um erro de compilação é indicado. Ao usar um IDE, o desenvolvedor tem *feedback* imediato se o tipo de retorno do método que está sendo sobrescrito é correto. Essa é uma das vantagens do framework GRENJ em relação ao GREN, decorrente da linguagem utilizada.

Recomenda-se que a instanciação de aplicações seja realizada aplicando TDD. No entanto, as classes da aplicação sendo considerada não precisam ser exaustivamente testadas, visto que as classes do framework já possuem testes associados. Somente pontos em que erros podem ser introduzidos durante a instanciação devem ser testados, como por exemplo, a configuração do comportamento de persistência de cada classe. A seguir a implementação da classe *Cliente*, aplicando TDD, é descrita.

Implementação da Classe *Cliente* Utilizando TDD

O primeiro passo para instanciação da classe *Cliente* usando TDD é a criação da lista de casos de teste. Para qualquer classe instanciada por meio do framework GRENJ os casos de testes da Tabela 6.6 sempre devem ser considerados. Esses casos de teste "forçam" o desenvolvedor a implementar os dois construtores e a maioria dos vários métodos necessários para que as classes da aplicação estendam as classes do framework corretamente.

Tabela 6.6: Lista de casos de teste para implementação da classe *Cliente*

Lista de Casos de Teste
Avaliar a "inicialização *default*" de objetos dessa classe (nesse caso *Cliente*).
Avaliar se todos os campos estão sendo apropriadamente recuperados do banco de dados.
Testar se objetos dessa classe são adequadamente persistidos e removidos do banco de dados.

Conforme pode ser observado na Tabela 6.5, a classe *Cliente* deve adicionar dois atributos (*telefone* e *cpf*) não fornecidos pela sua superclasse no framework. A inclusão desses dois atributos implica que os métodos relacionados à persistência devem ser sobrescritos a fim de adicionar as informações sobre *telefone* e *cpf*, para que essas informações possam ser armazenadas no banco de dados. Além disso, esses dois atributos devem ter valores *default*.

Inicia-se a implementação da classe *Cliente* por meio da implementação do primeiro caso de teste da Tabela 6.6. A escolha dos casos de teste é subjetiva, qualquer caso de teste poderia ter sido escolhido. Durante a implementação desse caso de teste decide-se que o valor *default* para o atributo *telefone* é a `String` `"(XX)XXXX-XXXX"` e, para o atributo *cpf*, esse valor é `"000.000.000-00"`. Desse modo, o teste da Listagem 6.6 verifica se tanto os atributos da superclasse de *Cliente* quanto os atributos dessa classe foram inicializados com os valores *default* especificados.

A linha 10 da Listagem 6.6 verifica se o atributo `idCode`, definido na classe `StaticObject` (superclasse de `DestinationParty`) foi devidamente inicializado e a linha 11 examina se o atributo `description`, também definido na classe `StaticObject`, foi inicializado com a `String` `""`. Dado que nenhum atributo foi alterado após a criação da instância de *Cliente* na linha 9, a instrução da linha 14, que verifica se o estado interno do objeto foi alterado, deve ser avaliada como **false**.

Esse teste não compila, pois a classe *Cliente* ainda não foi implementada. Essa classe, conforme especificado no mapeamento da Tabela 6.5, deve estender a classe abstrata do framework GRENJ denominada `DestinationParty`. Além disso, um

construtor, que não recebe nenhum parâmetro, deve ser implementado; as variáveis de instância *cpf* e *telefone* devem ser adicionadas à classe *Cliente* e inicializadas com os valores *default* especificados anteriormente. A implementação parcial dessa classe é ilustrada na Listagem 6.7. Como pode ser observado na Listagem 6.7, os atributos "adicionais", *cpf* e *telefone*, são declarados nas linhas 3 e 4 e o construtor *default*, linhas 6 – 11, inicializa as variáveis incluídas com os valores *default*. No entanto, para que o teste da Listagem 6.6 compile é necessário que os métodos `getCpf` e `getTelefone` sejam implementados, devido a simplicidade desses métodos, a implementação dos mesmos é omitida na Listagem 6.7.

Após a implementação da classe *Cliente*, como exibido na Listagem 6.7, executa-

Listagem 6.6: Primeiro caso de teste implementado que avalia a inicialização de instâncias da classe *Cliente* com valores *default*

```
1  import org.junit.Assert;
2  import org.junit.Test;
3
4  public class TestCliente {
5
6      @Test
7      public void testDefaultInitialization() {
8
9          Cliente cliente = new Cliente();
10         Assert.assertEquals( 0, cliente.getIdCode() );
11         Assert.assertEquals( "", cliente.getDescription() );
12         Assert.assertEquals( "000.000.000-00", cliente.getCpf() );
13         Assert.assertEquals( "(XX)XXXX-XXXX", cliente.getTelefone() );
14         Assert.assertFalse( cliente.isChanged() );
15     }
16 }
```

Listagem 6.7: Implementação parcial da classe *Cliente*

```
1  public class Cliente extends DestinationParty {
2
3      private String cpf;
4      private String telefone;
5
6      public Cliente() {
7          super();//DestinationParty
8          cpf = "000.000.000-00"; //default
9          telefone = "(XX)XXXX-XXXX";//default
10         super.setChanged( false );
11     }
12 }
```

se o teste da Listagem 6.6. O teste é satisfeito e, analisando o código do teste e da classe *Cliente*, percebe-se que nenhuma refatoração é necessária. Prossegue-se com a implementação do segundo caso de teste da Tabela 6.6, que avalia a recuperação de informações armazenadas no banco de dados. A implementação desse caso de teste é exibida na Listagem 6.8.

Listagem 6.8: Implementação do segundo caso de teste que verifica a recuperação das informações armazenadas no banco de dados

```
18 @Test
19 public void testInitializationWithValuesFromDB() throws Exception {
20
21    Cliente cliente = (Cliente) PersistentObject.load( 1,
22       Cliente.class );
23
24    Assert.assertEquals( 1, cliente.getIdCode() );
25    Assert.assertEquals( "Durelli", cliente.getDescription() );
26    Assert.assertEquals( "320.728.748-64", cliente.getCpf() );
27    Assert.assertEquals( "(16)3372-3864", cliente.getTelefone() );
28    Assert.assertTrue( cliente.isPersisted() );
29    Assert.assertFalse( cliente.isChanged() );
30 }
```

Depois da implementação do segundo caso de teste, executam-se todos os testes e o recém-implementado não é satisfeito. É necessário que uma tabela denominada *Cliente* seja criada e, pelo menos uma tupla seja inserida. O *script* da Listagem 6.4 foi utilizado para criação da tabela necessária. O primeiro passo para que esse teste seja satisfeito é a inserção de uma tupla na tabela *Cliente*. A instrução SQL para inserção da tupla avaliada nos testes é apresentada na Listagem 6.9.

Listagem 6.9: Instrução usada para inserção da tupla avaliada no teste da Listagem 6.8

```
1 INSERT INTO Cliente (idCode, description, telefone, cpf )
2 VALUES ( 1, "Durelli", "(16)3372-3864", "320.728.748-64" );
```

Na linha 21 o método `load`, da classe `PersistentObject`, é utilizado para carregar as informações de determinada tupla da tabela *Cliente* em uma instância da classe com o mesmo nome, especificamente a tupla cujo `idCode` seja igual a 1 – detalhes relacionados a implementação desse método serão omitidos. As linhas seguintes comparam as informações extraídas do banco de dados com as constantes: `1`, `“Durelli”`, `“320.728.748-64”` e `“(16)3372-3864”`. Para que esse teste seja satisfeito, é necessário implementar um construtor que recebe uma referência para uma instância de `java.sql.ResultSet` e uma de `grenj.util.Index`. Esse construtor deve inicializar

as variáveis de referência, definidas na classe, utilizando os valores recuperados do banco de dados e armazenados na instância de `ResultSet`. A instância de `Index` é usada para "controlar" a ordem que esses dados, contidos na instância de `ResultSet`, devem ser obtidos. A implementação desse construtor é exibida na Listagem 6.10.

Listagem 6.10: Construtor necessário para que o teste da Listagem 6.8 seja satisfeito

```
15 //necessary
16 public Cliente( ResultSet result, Index anIndex ) {
17    super( result, anIndex );
18    try {
19       this.telefone =
20           result.getString(anIndex.getIndex() );
21       anIndex.incrementIndexByOne();
22
23       this.cpf =
24          result.getString( anIndex.getIndex() );
25       anIndex.incrementIndexByOne();
26    } catch (SQLException e) {
27       e.printStackTrace();
28    }
29    super.setChanged( false );
30 }
```

Depois da implementação do construtor exibido na Listagem 6.10, executam-se todos os testes. O segundo teste implementado é satisfeito. Visto que a implementação desse construtor e do segundo caso de teste não introduziram nenhum *bad smell* (Fowler et al., 1999), nenhuma refatoração é realizada. O próximo passo, então, é a implementação do último caso de teste, que tem o propósito de verificar a persistência e remoção de informações. A implementação desse caso de teste é exibida na Listagem 6.11.

Como mencionado na Subseção 5.4.2, classes que adicionam atributos e necessitam ter suas informações persistidas no banco de dados devem sobrescrever os métodos: `insertionFieldClause`, `insertionValueClause`, `updateSetClause` e `updateWhereClause`. Esses métodos especificam como os atributos inseridos na classe que está sendo implementada devem ser persistidos:

- o método `insertionFieldClause` fornece o nome dos novos campos no banco de dados;
- o método `insertionValueClause` formata e retorna o valor de cada atributo que será persistido;
- `updateSetClause` retorna tanto o nome dos campos no banco de dados quanto os valores de cada atributo que será persistido;

- o método `updateWhereClause` retorna um objeto `String` que informa a tupla que deve ser atualizada.

Listagem 6.11: Implementação do terceiro caso de teste que verifica a persistência e remoção de informações do banco de dados

```
33 @Test
34 public void testSaveAndDelete() {
35     Cliente cliente = new Cliente();
36     cliente.setIdCode( 99 );
37     cliente.setDescription( "Ronald Bilius Weasley" );
38     cliente.setTelefone( "(16)3325-6785" );
39     cliente.setCpf( "330.889.907-99" );
40     //return 1 if the insertion was successful, otherwise -1
41     Assert.assertEquals( 1, cliente.save() );
42     //delete
43     Assert.assertEquals( 1, cliente.delete() );
44 }
```

Observa-se na Listagem 6.11 que os métodos anteriormente mencionados não são invocados diretamente. O desenvolvedor usando o GRENJ implementa os métodos gancho que são implicitamente invocados pelo framework. Esse conceito é denominado "Princípio de Hollywood" (Larman, 2004). Para que as informações sobre a instância da classe *Cliente*, criada na linha 35, sejam persistidas esse métodos foram implementados como exibido na Listagem 6.12.

Depois de implementar os quatro métodos da Listagem 6.12, executam-se os testes e todos são satisfeitos. Assim, a instanciação da classe *Cliente*, usando o GRENJ, pode ser considerada completa.

Durante a implementação do código necessário para que os testes fossem satisfeitos a técnica *Use Obvious Implementation* (Beck, 2002; Koskela, 2007) foi aplicada. Essa técnica pode ser utilizada por um desenvolvedor com conhecimentos avançados sobre a arquitetura interna do GRENJ e seus métodos gancho. Já um desenvolvedor sem tais conhecimentos pode utilizar a técnica *Fake It* (Beck, 2002; Koskela, 2007) e, gradualmente, substituir constantes por variáveis conforme obtiver mais informações sobre o GRENJ. Dessa forma, tanto desenvolvedores experientes quanto inexperientes, em relação ao GRENJ, podem aplicar TDD durante a instanciação de aplicações por meio do framework. No entanto é recomendável conhecimentos intermediários nos seguintes pontos da linguagem Java™: *generics* e *reflection*.

6.2.4 Teste do Sistema Resultante

De acordo com Braga (2002b), nessa etapa, a aplicação instanciada deve ser testada tanto para verificar se ela atende aos requisitos estabelecidos, quanto para ava-

liar se ela funciona no ambiente do usuário final. Nenhuma abordagem específica para realização dessa etapa é proposta por ela e isso também está fora do escopo deste trabalho. Dessa forma, no caso da aplicação da locadora de DVDs, somente os testes de unidade realizados durante a instanciação da aplicação foram realizados. É importante ressaltar que, como a camada de interface gráfica do GREN não passou por reengenharia, foi necessário criar uma interface gráfica com o usuário para que essa aplicação pudesse ser utilizada. Essa interface gráfica foi desenvolvida usando o *GUI builder Matisse*, do IDE Netbeans[1]. O padrão arquitetural MVC (*Model-View-Controller*) (Gamma et al., 1995) conforme proposto por Durelli et al. (2008), foi aplicado a fim de, coerentemente, separar a camada com as classes reutilizadas do framework, nesse caso *Model*, da camada de interface gráfica com o usuário, *View*. Detalhes adicionais relacionados ao desenvolvimento dessa interface gráfica com o usuário são omitidos.

Listagem 6.12: Implementação dos métodos requeridos para persistência

```
1  @Override
2  public String insertionFieldClause() {
3     return super.insertionFieldClause() + ", telefone, cpf";
4  }
5
6  @Override
7  public String insertionValueClause() {
8     StringBuilder insertionValueClause =
9        new StringBuilder( super.insertionValueClause() );
10    insertionValueClause.append( ", \'" + this.getTelefone() + "\'" );
11    insertionValueClause.append( ", \'" + this.getCpf() + "\'" );
12    return insertionValueClause.toString();
13 }
14
15 @Override
16 public String updateSetClause() {
17    StringBuilder updateSetClause =
18       new StringBuilder( super.updateSetClause() );
19    updateSetClause.append( ", telefone = \'"  +
20    this.getTelefone() + "\'" );
21    updateSetClause.append( ", cpf = \'"  +
22    this.getCpf() + "\'" );
23    return updateSetClause.toString();
24 }
25
26 @Override
27 public String updateWhereClause() {
28    return "idCode = " + getIdCode() + ", cpf = \'" + getCpf() + "\'";
29 }
```

[1] *http://www.netbeans.org/*

6.3 Considerações Finais

Apesar de frameworks proporcionarem reúso de projeto e código, alcançar esse reúso de maneira apropriada não é trivial. Frameworks são softwares mais abstratos, pois, devem implementar a flexibilidade necessária para instanciar várias aplicações pertencentes ao domínio por ele abrangido. Dessa forma, na maioria das vezes, eles são difíceis de serem compreendidos. Assim, demora para que um desenvolvedor ou usuário obtenha o conhecimento necessário para sua utilização. Porém, frameworks desenvolvidos com base em linguagens de padrões podem utilizar essa linguagem de padrões para auxiliar a instanciação de aplicações. Este Capítulo apresentou em detalhes o processo que pode ser empregado para instanciação de aplicações por meio do framework GRENJ e utilizando a GRN como "guia", durante o processo de instanciação.

O processo de instanciação, definido por Braga (2002b), foi exemplificado com uma aplicação de locadora de DVDs. Os requisitos dessa aplicação fictícia foram satisfeitos com a aplicação de alguns padrões da GRN, que também foram usados para modelá-la. Mais especificamente, os padrões aplicados foram: IDENTIFICAR O RECURSO, RECURSO INSTANCIÁVEL, LOCAR O RECURSO e ITEMIZAR A TRANSAÇÃO DO RECURSO. Além da introdução de quatro atributos, nenhuma alteração foi necessária. Aplicar esses padrões para modelar a aplicação resulta em reúso de experiência, pois, o desenvolvedor não precisa conhecer o domínio sendo abordado, basta reutilizar a "estrutura" e solução proposta pelos padrões.

O "mapeamento" entre os padrões da GRN e as classes que os implementam esses padrões no framework GRENJ, realizado na segunda etapa do processo (Subseção 6.2.2), facilita a identificação das classes que devem ser estendidas. O *"cookbook"* do GREN pode ser usado durante esse "mapeamento", visto que nas classes do GRENJ foram mantidos os mesmos nomes das classes do GREN.

Recomenda-se que, durante a instanciação das aplicações usando o GRENJ, o desenvolvedor utilize TDD a fim de evitar a introdução de problemas e para ajudá-lo a manter o enfoque nas atividades que devem ser realizadas. Por exemplo, ao invés de implementar todos os métodos gancho de uma vez, o desenvolvedor implementa somente aqueles necessários para que o teste anteriormente criado seja satisfeito. Porém, a implementação dos casos de teste aumenta consideravelmente o número de linhas de código que devem ser implementadas. Durante a instanciação da aplicação mencionada neste Capítulo, implementou-se aproximadamente duas mil e trezentas linhas de código, das quais pouco mais de novecentas estão relacionadas aos testes criados. O diagrama de classes da aplicação, exibido na Figura 6.3, demorou aproximadamente uma hora para ser obtido. O "mapeamento" entre as classes desse diagrama e as classes do GRENJ, realizado com o apoio do *"cookbook"* do GREN, de-

morou em torno de quarenta minutos e a implementação das classes necessárias, aplicando TDD, foi realizada em pouco menos de quatro horas; sem considerar o tempo necessário para implementação da interface gráfica com o usuário.

Tanto os desenvolvedores experientes quanto os inexperientes, em relação ao framework GRENJ, podem utilizar TDD durante a instanciação de aplicações. Como descrito neste Capítulo, as técnicas *Use Obvious Implementation* (Beck, 2002; Koskela, 2007) e *Fake It* (Beck, 2002; Koskela, 2007) fornecem apoio necessário para cada tipo de desenvolvedor, respectivamente. Entretanto, é recomendável conhecimentos intermediários nos seguintes pontos da linguagem Java™: *generics* e *reflection*.

O próximo Capítulo apresenta as principais contribuições deste trabalho, suas limitações e possíveis trabalhos futuros.

CAPÍTULO
7

Conclusões

7.1 Considerações Finais

Este trabalho apresentou um processo iterativo para reengenharia de software utilizando padrões de engenharia reversa, aplicando a técnica de TDD e refatoração. A reengenharia do framework GREN foi realizada, gerando o framework GRENJ implementado na linguagem Java™e AspectJ. Com este trabalho o grupo de Engenharia de Software do Departamento de Computação da Universidade Federal de São Carlos, bem como colaboradores desse grupo, pertencentes a outras instituições de ensino superior, têm disponível um framework que possibilita a instanciação de sistemas na linguagem de programação Java™. Esses sistemas instanciados pelo framework abrangem o domínio de gestão de recursos de negócio. Porém, há possibilidade de se estender o framework GRENJ a fim de abranger outros domínios, relacionados ou não a gestão de recursos de negócio. O GRENJ também torna viável a realização de experimentos com outros produtos de pesquisa, também implementados na linguagem Java™, que anteriormente não poderiam ser realizados, devido às diferenças entre a linguagem Java™e Smalltalk.

As contribuições e limitações deste trabalho são descritas nas Seções 7.2 e 7.3, respectivamente. Na Seção 7.4 são apresentadas sugestões para trabalhos futuros que podem ser realizados para complementar o aqui apresentado.

7.2 Contribuições

Este trabalho propõe um processo iterativo que pode ser empregado para reengenharia de sistemas de software envolvendo a conversão de linguagens, principalmente quando as duas linguagens envolvidas têm características diferentes. Tal processo foi definido com o propósito de auxiliar os engenheiros de software que confrontam-se com problema semelhante, de forma que eles possam reutilizar as experiências e técnicas já avaliadas e que deram certo.

Duas práticas fundamentais para eficácia do processo são: TDD e refatoração. O TDD é empregado, no contexto do processo, para facilitar e "guiar" a implementação. A funcionalidade de cada "parcela" do sistema que está passando por reengenharia é dividida em casos de testes, que são implementados conforme se avança. Após todos os casos de teste terem sido satisfeitos, a funcionalidade é considerada implementada. Um dos benefícios de se decompor a implementação da funcionalidade em casos de teste é transformar o problema em algo mais concreto e com menor escopo, tornando-o mais fácil de ser abordado.

Conforme salientado na Seção 4.2, TDD é uma técnica de análise e projeto e não de teste, apesar da sua denominação. Entretanto, além de serem aplicados enquanto o desenvolvedor *programa por intenção* (Koskela, 2007), para definição das classes e métodos que deverão ser implementados, os testes produzidos durante o desenvolvimento podem ser utilizados em futuras atividades de manutenção, atuando como testes de regressão. Além disso, esses testes podem ser considerados como documentação sempre atualizada do sistema, pois as alterações realizadas no código fonte devem ser "refletidas" nos testes (Demeyer et al., 2002).

O TDD engloba um passo para realização de atividades de refatoração (Seções 2.6 e 4.3). Essas atividades auxiliam a manter a legibilidade do código e possibilitam a introdução de funcionalidade. Isso é fundamental para que a funcionalidade implementada pelas várias "parcelas" do sistema que está passando por reengenharia possam ser gradualmente adicionadas conforme se avança.

Outra contribuição deste trabalho é o framework GRENJ, "versão" Java™do framework GREN. A documentação do framework GRENJ foi produzida utilizando *tags javadoc* (Sun Microsystems, Inc., 2007) inseridas no código fonte do framework. Essas informações, fornecidas por meio dessas *tags*, documentam a API pública do GRENJ, ou seja, somente classes e métodos públicos tiveram seu propósito documentado. As seguintes informações são fornecidas:

- hierarquia em que a classe se encontra;
- autor, ou seja, programador responsável pela implementação da classe ou método (atualmente, todas as classes e métodos do framework GRENJ foram im-

plementadas pelo autor deste trabalho);

- a partir de qual versão do framework a classe ou método foi introduzido (como o GRENJ ainda não passou por nenhuma atividade de manutenção todas as classes pertencem a versão 1.0);
- tipo de retorno e significado dos possíveis valores retornados pelos métodos, bem como as exceções que podem ser lançadas.

Para o GRENJ foram implementados duzentos e quarenta e nove testes de unidade totalizando aproximadamente onze mil linhas de código. De acordo com a análise de cobertura realizada empregando a ferramenta EclEmma (EclEmma, 2008), esses testes "exercitam" pouco mais de cinqüenta por cento do código do framework. No entanto, é importante ressaltar que devido às várias "combinações" possíveis de padrões e subpadrões que podem ser aplicados, bem como o número elevado de métodos gancho que devem ser sobrescritos durante a instanciação de aplicações, seria inviável, no tempo dedicado para realização deste trabalho, obter uma maior cobertura por parte dos testes implementados durante o desenvolvimento do GRENJ.

Durante o processo de reengenharia apresentado neste trabalho, vários padrões de projeto foram seletivamente aplicados durante o desenvolvimento do GRENJ, entre eles: *Value Object* (Fowler, 2002), *Persistent Layer* (Yoder et al., 1998), *Factory Method*, *Singleton* (utilizando *lazy instantiation*) e *Template Method* (Gamma et al., 1995). A aplicação desses padrões tem três motivos principais: (i) reusar soluções comprovadas de sucesso para introduzir a flexibilidade requerida em determinados pontos do framework; (ii) obter um produto, nesse caso o framework, de qualidade e (iii) facilitar a compreensão do código fonte por parte dos desenvolvedores e futuros mantenedores desse framework. Adicionalmente, a flexibilidade introduzida por meio da aplicação dos padrões de projeto proporciona que o framework seja facilmente estendido.

A principal contribuição deste trabalho é o framework GRENJ que instancia sistemas de software, pertencentes ao domínio de gestão de recursos de negócios, na linguagem Java™.

7.3 Limitações do Trabalho Efetuado

Para instanciação de aplicações por meio de frameworks caixa-branca é necessário que os desenvolvedores possuam conhecimentos avançados sobre a arquitetura do framework que está sendo utilizado. Os relacionamentos existentes entre as várias classes do framework devem ser compreendidos e, principalmente, a forma como os métodos gancho devem ser sobrescritos para que se obtenha a customização almejada para cada aplicação. O GREN-Wizard (Braga, 2002b) auxilia na instanciação de

aplicações quando o framework GREN é utilizado. Por meio dele, o usuário ou desenvolvedor necessita somente conhecer a GRN e o funcionamento do GREN-Wizard, propriamente dito, para instanciar aplicações que reutilizam e estendem o código do GREN. Dado que o GREN-Wizard não passou por reengenharia, uma das limitações de se instanciar aplicações por meio do GRENJ é que o desenvolvedor deve ter amplo conhecimento:

- da GRN, a fim de identificar o papel de cada classe do framework GRENJ e aplicá-la para modelagem da aplicação a ser instanciada, quando necessário;
- da arquitetura do GRENJ;
- dos métodos gancho e o propósito de cada um deles;
- da linguagem Java™, visto que durante o processo de instanciação, o desenvolvedor codifica métodos que utilizam *features* nela existentes, como por exemplo: (i) *reflection*, para carregamento dinâmico das classes customizadas e (ii) *generics*, a fim de evitar que classes de tipos inapropriados sejam codificadas no método.

A camada de interface gráfica com o usuário do framework GREN também não passou por reengenharia, neste projeto, o que limita a reutilização alcançada por meio do framework GRENJ, considerando que, para utilização das aplicações instanciadas, a camada de interface gráfica deve ser completamente projetada a partir do zero (*from scratch*).

Os testes criados durante o desenvolvimento do framework exercitam pouco mais de cinqüenta por cento do código fonte do framework, dado que nem todas as possíveis combinações de instanciações foram testadas. Testar todas as possíveis combinações de aplicação dos padrões tornaria o desenvolvimento inviável no tempo estipulado.

As outras limitações do framework GRENJ são devidas ao fato da GRN pertencer a um domínio restrito, gerenciamento de recursos de negócios. Isso implica que o GRENJ deve passar por manutenção caso precise ser estendido a fim de viabilizar sua aplicação em outros domínios.

7.4 Sugestões de Trabalhos Futuros

Considerando que o framework GRENJ pode ser amplamente utilizado em novas pesquisas ou para a instanciação de sistemas de software, a seguir são listados alguns trabalhos que podem ser realizados para mitigar as limitações comentadas anteriormente ou adicionar mais funcionalidade e qualidade ao framework existente.

Implementação da camada de interface gráfica com o usuário: conforme mencionado, a camada de interface gráfica com o usuário do framework GREN não passou por reengenharia. O desenvolvimento dessa camada, utilizando as tecnologias *JavaServer Pages*[1] e *Servlet*[2], possibilitaria a instanciação de aplicações voltadas para Web usando o framework GRENJ.

GRENJ-Wizard: A criação de um *wizard*, semelhante ao GREN-Wizard, poderia ser realizada; possibilitando a instanciação automatizada de aplicações sem que o desenvolvedor tenha conhecimento de detalhes relacionados à implementação do framework GRENJ.

Introdução de Frameworks Transversais: verificar a possibilidade de utilizar frameworks ou famílias de frameworks existentes para o tratamento de interesses transversais, tais como, persistência e segurança de dados. Um estudo pode ser realizado substituindo o mecanismo de persistência atualmente existente no GRENJ por framework ou família de frameworks, tal como a proposta por Camargo (2006).

Geração de relatórios: os métodos do GRENJ relacionados à produção de relatórios, como por exemplo, lista de clientes inadimplentes, transações pendentes, entre outros, retornam, na maioria das vezes, uma lista de objetos que contêm as informações desejadas – uma implementação de `java.util.List`. No entanto, a exibição dessas informações fica a cargo do desenvolvedor responsável pela implementação da interface gráfica. É possível, por exemplo, exibir tais informações em uma tabela (`javax.swing.JTable`). A introdução de ferramentas como a biblioteca de código livre JasperReports[3] para produção de relatórios em vários formatos, como por exemplo *Portable Document Format* (PDF), diminuiria o esforço por parte do desenvolvedor e, além disso, aprimoraria a qualidade dos relatórios gerados.

Eliminar as dependências dos testes com a base de dados: os testes criados durante o desenvolvimento do GRENJ comunicam-se com o banco de dados a fim de avaliar a inserção, remoção e atualização de tuplas. Essa dependência evita que os testes sejam executados caso o banco de dados ainda não tenha sido configurado com as tabelas necessárias. Esse problema poderia ser eliminado por meio da utilização de objetos *mock* para a simulação da interação com o banco de dados. Dessa forma, a "instalação" dos testes do framework GRENJ seria facilitada, pois eliminaria a necessidade de se configurar o banco de dados com as tabelas avaliadas pelos testes.

[1] *http://java.sun.com/products/jsp/*
[2] *http://java.sun.com/products/servlet/*
[3] *http://www.jasperforge.org/jaspersoft/opensource/business_intelligence/jasperreports/*

Avaliação da extensibilidade do processo iterativo de reengenharia: aplicar o processo iterativo de reengenharia em sistemas de pequeno porte (menos de vinte classes) a fim de aprimorar esse processo. Durante a realização da reengenharia desses sistemas outros padrões de engenharia reversa podem ser introduzidos ao processo. É recomendável que os sistemas submetidos à reengenharia sejam implementados em alguma linguagem dinamicamente tipada como, por exemplo, Ruby[4] e Python[5], e que a linguagem selecionada para implementação da solução alvo seja estaticamente tipada, como Java™e C++.

Avaliar a substituição do GREN pelo GRENJ no ARA: o Arcabouço de Reengenharia Ágil (ARA) (Cagnin, 2005), desenvolvido com o intuito de apoiar a migração de sistemas procedimentais para o paradigma orientado a objetos, utiliza o GREN para alcançar reúso de projeto e código. Pode-se avaliar a substituição do GREN pelo GRENJ nesse contexto.

Produção do "cookbook" do GRENJ: um dos fatores que pode contribuir para facilitar a instanciação de aplicações usando o GRENJ é a disponibilidade de uma documentação abrangente e adequada. Atualmente, o framework é documentado unicamente por meio de *tags javadoc* (Sun Microsystems, Inc., 2007). A produção de um documento semelhante ao *"cookbook"* do framework GREN poderia auxiliar na instanciação de aplicações. Tal *"cookbook"* documentaria os métodos que precisam ser sobrescritos, mapeamento entre a GRN e o GRENJ, os *scripts* usados para criação das tabelas necessárias e exemplos detalhados de instanciação.

[4] *http://www.ruby-lang.org/en/*
[5] *http://www.python.org/*

Apêndices

APÊNDICE

A

Breve Introdução ao Framework JUnit Versão 4.1

A.1 Introdução

Este Apêndice apresenta uma concisa introdução ao framework JUnit versão 4.1, amplamente utilizado pela comunidade de desenvolvedores Java™ (Stuckert, 2006). O JUnit foi usado para facilitar e automatizar a criação de testes unitários durante o desenvolvimento do framework GRENJ. Assim, esta breve introdução tem a finalidade de facilitar a compreensão, por parte do leitor, dos trechos de código relacionados aos testes apresentados no decorrer deste trabalho. O framework JUnit é um projeto de código aberto (*open source*) hospedado em *http://sourceforge.net/* e mantido sob uma licença de distribuição denominada *Common Public License*, que pode ser encontrada em *http://junit.sourceforge.net/cpl-v10.html*.

O JUnit possui várias funcionalidades que facilitam a criação de testes de unidade. As fundamentais como, por exemplo, a criação dos testes e das *fixtures* terão sua sintaxe exibida e brevemente explicada a seguir. Informações adicionais podem ser encontradas no *"cookbook"* do JUnit, criado por Kent Beck e Erich Gamma. O *"cookbook"* está disponível em: *http://junit.sourceforge.net/doc/cookbook/cookbook.htm*. Detalhes sobre a instalação do framework são omitidos neste Apêndice, no entanto, tal assunto é abordado em: *http://junit.sourceforge.net/doc/faq/faq.htm#started_2*.

A.2 Criando Testes Utilizando a Anotação `@org.junit.Test`

A criação de testes utilizando o framework JUnit é bem simples. Qualquer método público assinalado com a anotação (*annotation*) `@org.junit.Test` é considerado um caso de teste pelo framework. Dado que os casos de teste são métodos da linguagem Java™ anotados com `@Test`, eles devem pertencer a uma classe de teste. É recomendável que o nome da classe de teste sinalize seu propósito (Koskela, 2007), uma das convenções é utilizar o prefixo `Test`. Os métodos anotados com `@Test` devem possuir o modificador de acesso **public**, caso contrário o framework JUnit lança uma exceção. A Listagem A.1 ilustra a criação de dois casos de teste, linhas 7–11 e 13–17. Casos de teste também podem declarar que lançam exceções, conforme pode ser visualizado na Listagem A.1 na linha 14.

Listagem A.1: Utilizando a anotação `@org.junit.Test` para criação de testes

```
1  import junit.framework.Assert;
2
3  import org.junit.Test;
4
5  public class TestClass {
6
7      @Test
8      public void testMathAbs() {
9          long result = 99;
10         Assert.assertEquals( result, Math.abs( -99 ) );
11     }
12
13     @Test
14     public void testLoad() throws ClassNotFoundException {
15         Class classString = Class.forName( "java.lang.String" );
16         Assert.assertEquals( "String", classString.getSimpleName() );
17     }
18
19 }
```

A.3 Assertivas JUnit

Nos testes de unidade parcelas do código do sistema são "exercitadas", entradas são fornecidas quando necessário, e, posteriormente, o comportamento do código é avaliado (Louridas, 2005). Em outras palavras, testes avaliam o comportamento esperado do sistema após determinado processamento realizado. O JUnit permite que o resultado seja averiguado por meio da utilização de assertivas. Existem assertivas para comparar todos os tipos primitivos da linguagem Java™. Além disso há assertivas

para verificar se duas variáveis mantém referência para o mesmo objeto, para avaliar se uma referência não é nula e para avaliar quando dois objetos são iguais. Cada uma dessas assertivas possui uma versão que permite que uma mensagem, criada pelo desenvolvedor, seja exibida em casos onde a assertiva não é satisfeita. A Listagem A.2 ilustra a avaliação da igualdade de dois inteiros, a primeira versão da assertiva não exibe nenhuma mensagem em caso de desigualdade (linha 11), a segunda exibe a mensagem: "98 != 99" (linhas 12–13).

Listagem A.2: Utilizando as assertivas do JUnit

```
1  import junit.framework.Assert;
2
3  import org.junit.Test;
4
5  public class TestIntegers {
6
7      @Test
8      public void testIntegers() {
9          int ninetyNine = 99;
10         int ninetyEight = 98;
11         Assert.assertEquals( 49 + 49, ninetyEight );
12         Assert.assertEquals( String.format( "%d != %d", ninetyEight,
13             ninetyNine ), ninetyEight, ninetyNine );
14     }
15
16 }
```

Listagem A.3: Utilizando assertivas sem instrução de **import**

```
junit.framework.Assert.assertEquals( 49 + 49, ninetyEight );
```

Para que as assertivas possam ser utilizadas pode se utilizar uma instrução de **import** normal e utilizar o nome da classe juntamente com as assertivas, conforme ilustrado no trecho de código da Listagem A.2. Utilizar o nome da classe completamente especificado (sem necessidade de instrução de **import**), conforme pode ser visualizado no trecho de código da Listagem A.3. Por fim, é possível utilizar uma facilidade introduzida a partir da versão 5.0 da linguagem Java™, denominada *static import*. Essa variação da instrução de **import** facilita a importação de membros definidos como **static** para o *namespace* da classe em questão, de forma que não é necessário "qualificá-los" durante a utilização. A Listagem A.4 fornece um exemplo da utilização de tal instrução.

A.4 Configurando o Contexto Necessário Para Realização dos Testes

O contexto necessário para execução dos testes pode ser inicializado (ou configurado) na *fixture* do teste. Para criação do código, que é sempre executado antes de cada caso de teste, utiliza-se a anotação `@org.junit.Before`. Um exemplo pode ser visualizado nas linhas 4–7 da Listagem A.5. De maneira semelhante, métodos com a anotação `@org.junit.After` representam o que deve ser executado após cada teste, conforme exposto nos linhas 9–12. Métodos com tal anotação, geralmente, são utilizados para liberar os recursos alocados na *fixture* do teste. Essa fase do teste é denominada *fixture teardown* (Meszaros, 2007). Caso vários métodos estejam marcados com `@Before` ou `@After` a ordem de execução é indeterminada (Stuckert, 2006).

Listagem A.4: Utilizando *static import*

```
1  //static import
2  import static junit.framework.Assert.*;
3
4  import org.junit.Test;
5
6  public class TestString {
7
8      @Test
9      public void testSubstring() {
10         //usage
11         assertEquals( 'e', "test".charAt( 1 ) );
12     }
13 }
```

Quando a locação e liberação de recursos é custosa, como por exemplo a criação e liberação de uma conexão com o banco de dados, é desejável que o código relacionado à locação e liberação de recursos seja executado somente uma vez. As anotações `@org.junit.BeforeClass` e `@org.junit.AfterClass` do framework JUnit proporcionam o comportamento desejado. Com a anotação `@org.junit.BeforeClass` define-se o código que irá inicializar o contexto para a execução de todos os testes, sendo executado somente um vez para todo o conjunto de casos de teste, ao contrário de `@org.junit.Before` que sinaliza o código executado no início de cada teste. Equivalentemente, a anotação `@org.junit.AfterClass` sinaliza o código que é executado somente ao término da execução de todos os casos de teste, com o propósito de liberar os recursos alocados. A Listagem A.6 apresenta um exemplo de utilização dessas duas anotações e, como pode ser observado, os métodos assinalados com `@BeforeClass` ou `@AfterClass` devem ser métodos **static** (Stuckert, 2006). É possível definir vários

métodos assinalados com essas duas anotações.

Listagem A.5: Ajustando o contexto para execução de cada teste

```
1  public class TestFibonacci {
2     private int[][] fibonacciValues;
3
4     @org.junit.Before
5     public void prepareTestData() {
6        fibonacciValues = new int[][] { { 0, 0 }, { 1, 1 }, { 2, 1 } };
7     }
8
9     @org.junit.After
10    public void cleanupTestData() {
11       fibonacciValues = null;
12    }
13
14 }
```

Listagem A.6: Ajustando o contexto para execução de todos os testes

```
1  import java.io.IOException;
2  import java.util.Scanner;
3
4  public class TestSetUpAndTeardown {
5
6     private static Scanner scanner;
7
8     @org.junit.BeforeClass
9     public static void setUp() {
10       scanner = new Scanner( "Test-Driven Development" );
11    }
12
13    @org.junit.AfterClass
14    public static void teardown() throws IOException {
15       scanner.close();
16    }
17 }
```

A.5 Coibindo a Execução de Casos de Teste

Em alguns casos pode ser necessário impedir que um caso de teste, do conjunto de casos de teste, não seja executado. Nas versões anteriores do framework JUnit não havia uma forma eficiente de coibir a execução de determinados casos de teste. A partir da versão 4 do framework, para que um caso de teste não seja executado ele

deve ser precedido pela anotação `@org.junit.Ignore`. Pode-se fornecer um mensagem que justifique a exclusão temporária do caso de teste. A Listagem A.7 exibe exemplos da utilização dessa anotação.

Listagem A.7: Coibindo a execução de alguns casos de teste

```
1   //with message
2   @org.junit.Ignore("<fill in a good reason here>")
3   @Test
4   public void testArrayCopy() {
5      int[] array = new int[ 3 ];
6      System.arraycopy( new int[] { 1, 2, 3 }, 0, array, 0, 3 );
7      for ( int i = 0; i < array.length; i++ ) {
8      Assert.assertEquals( i + 1, array[ i ] );
9      }
10  }
11
12  //without message
13  @org.junit.Ignore
14  @Test
15  public void testAnotherSpecialFuncionality() {
16     int[] sortedIntArray = new int[]{ 30, 44, 90, 90, 100 };
17     int index = Arrays.binarySearch( sortedIntArray, 90 );
18     Assert.assertEquals( 2, index );
19  }
```

APÊNDICE

B

Breve Introdução à Biblioteca EasyMock Versão 2.3

B.1 Introdução

A seguir é apresentada uma introdução à biblioteca EasyMock, mais especificamente na sua versão 2.3. Essa introdução tem o propósito de facilitar a compreensão dos trechos de código apresentados no decorrer deste trabalho.

O EasyMock foi a primeira biblioteca, implementada na linguagem Java™, voltada para geração dinâmica de objetos *mock*. Disponível sob os termos da licença do MIT a biblioteca pode ser encontrada em: *http://www.easymock.org/Downloads.html*.

Utilizando o EasyMock é possível criar objetos *mock* (abordados na Seção 4.5) a partir de interfaces Java™. Há também uma extensão da biblioteca voltada para criação de objetos *mock* a partir de classes.

B.2 Motivação

Geralmente, as classes de um sistema orientado a objetos colaboram entre si para a realização da funcionalidade atribuída. Todavia, durante os testes é recomendável que cada classe seja testada separadamente. Objetos *mock*, denominados genericamente de *test doubles*, servem como "substitutos" para as instâncias das classes que colaboram com a classe sendo testada.

Por exemplo, considere a criação dos testes relacionados à classe `SubjectImplementation`, ilustrada no diagrama de classes apresentado na Figura B.1. Para testar

alguns métodos dessa classe, que implementa a interface `Subject`, é necessário criar objetos *mock*. Tais objetos *mock* têm o propósito de substituir possíveis instâncias que irão implementar a interface `Observer`. Assim, empregando alguma biblioteca ou framework para geração dinâmica de objetos *mock*, o desenvolvedor não necessita criar nenhuma classe que implemente a interface `Observer`, durante a criação dos testes da classe `SubjectImplementation`. A biblioteca ou framework se encarrega de gerar dinamicamente substitutos para esse relacionamento, como ilustrado na diagrama de objetos – também conhecido como diagrama de instâncias[1] – da Figura B.2.

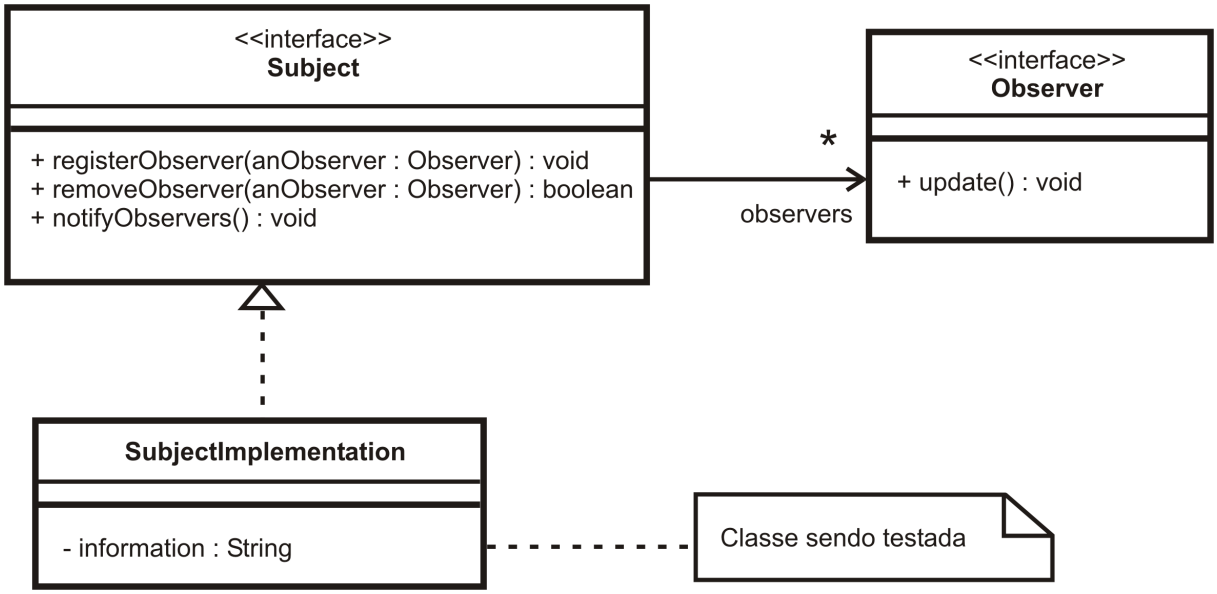


Figura B.1: Diagrama de classes do sistema hipotético sendo testado; aplicação do padrão de projeto *Observer* (Gamma et al., 1995)

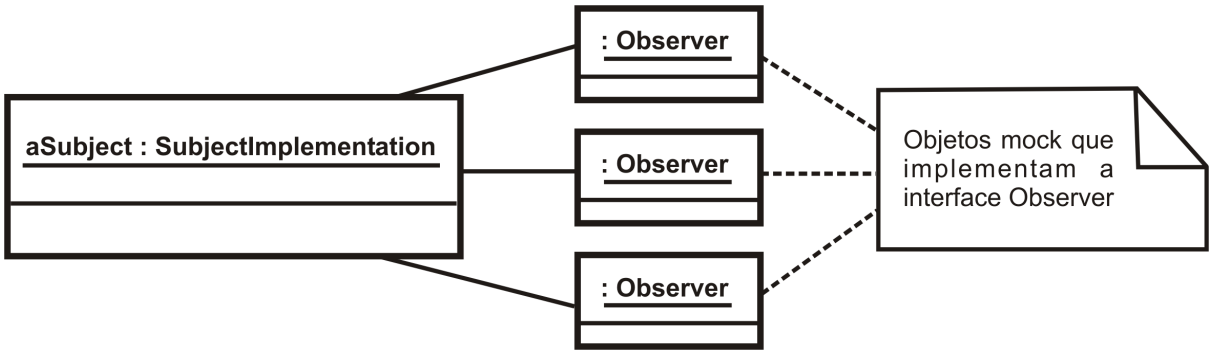


Figura B.2: Diagrama de objetos mostrando a relação entre instâncias de `SubjectImplementation` e objetos *mock*

[1]Mais precisamente, os elementos do diagrama são especificações de instâncias (OMG, 2008). Pois, de acordo com Fowler (2005), é válido omitir atributos ou utilizar uma classe abstrata como qualificador.

B.3 Criação e Utilização de Objetos *Mock*

Como o enfoque do Apêndice é elucidar a biblioteca EasyMock, nos exemplos a seguir, somente as partes do código relacionadas à utilização da biblioteca serão apresentadas, qualquer outro código será suprimido na maioria das Listagens. Os trechos de código apresentados são fundamentados no diagrama de classes da Figura B.1.

Para utilizar a biblioteca EasyMock o primeiro passo é *importar* a classe EasyMock. Conforme exibido na Listagem B.1.

Listagem B.1: Importando a classe `org.easymock.EasyMock`

```
import org.easymock.EasyMock;
```

Após importar a classe EasyMock, a criação dos objetos *mock* é realizada como a seguir:

Listagem B.2: Criando objeto *mock* utilizando a biblioteca EasyMock

```
Observer mockObserver = EasyMock.createMock( Observer.class );
```

O método `createMock` é utilizado para criação dos objetos *mock*. É necessário indicar a interface que será implementada pelo objeto *mock* a ser gerado. No caso da Listagem B.2 a interface passada como parâmetro para o método é a interface `Observer`. Há uma versão do método `createMock` que permite que o objeto *mock* seja nomeado: `EasyMock.createMock( String nome, Class mock )`. Facilitando assim a depuração em casos onde vários objetos *mock* são criados.

Após criados, os objetos *mock* possuem a mesma API da interface utilizada na sua criação. O que possibilita que eles sejam utilizados como substitutos para instâncias que implementam a interface, conforme exemplificado na Listagem B.3. Onde utiliza-se o objeto *mock*, criado na linha 2, como substituto para um objeto de alguma classe que implementa a interface `Observer`. Na Listagem B.3 o objeto *mock* é utilizado como argumento para o método `registerObserver` (linha 3).

Listagem B.3: Utilizando objetos *mock*

```
1 subjectImplementation = new SubjectImplementation();
2 Observer mockObserver = EasyMock.createMock( Observer.class );
3 subjectImplementation.registerObserver( mockObserver );
```

B.4 O Ciclo dos Testes Usando Objetos *Mock*

A principal característica que diferencia objetos *mock* de objetos *stub* (ambos definidos na Seção 4.5) é que objetos *mock*, além da funcionalidade de objetos *stub*, possuem a capacidade de verificar a ocorrência do comportamento. Nesse caso, comportamento são os métodos que devem ser invocados, assim como a ordem e quantidade de vezes que essas invocações ocorrerão.

Um objeto *mock* criado utilizando a biblioteca EasyMock permanece no estado recém-criado até o início da especificação do comportamento. Quando inicia-se a especificação do comportamento o objeto *mock* passa para um estado denominado "gravação" (*record state*). O último estado de um objeto *mock* é o estado de verificação, que investiga se o comportamento, especificado no estado anterior, aconteceu. Assim, após a criação dos objetos *mock*, a realização de testes utilizando a biblioteca EasyMock é dividida em duas etapas: gravação e verificação do comportamento (etapa 1 e 2, respectivamente). O fim da etapa de gravação e início do etapa de verificação é sinalizado pelo método `replay` e o encerramento da etapa de verificação ocorre com a execução do método `verify` – ambos pertencentes à biblioteca EasyMock. As duas etapas e os três estados que envolvem o ciclo de um objeto *mock* são ilustrados na Figura B.3.

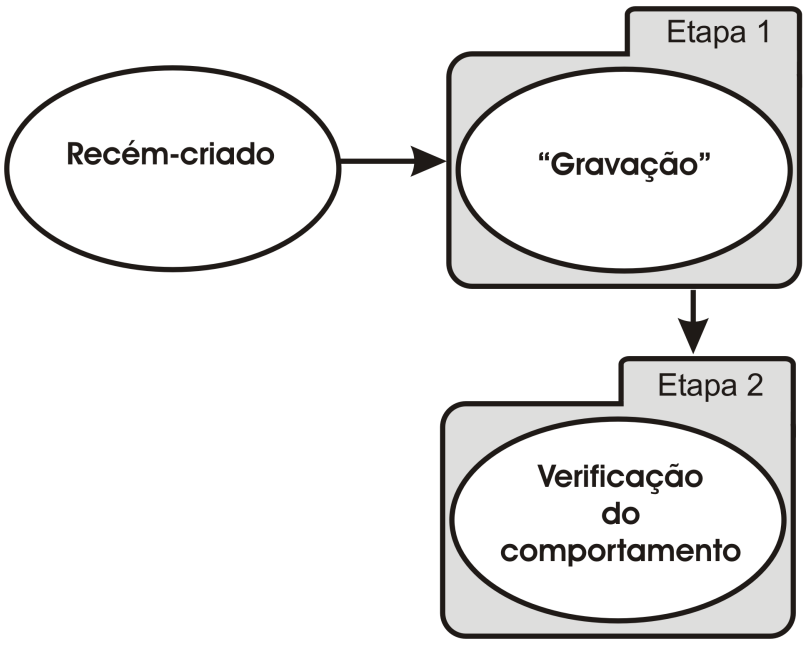


Figura B.3: Estados de um objeto *mock*

A Listagem B.4 fornece um exemplo que abrange as duas etapas. Conforme pode ser observado na primeira etapa – gravação – especifica-se que o método `update` será invocado, linha 6. A segunda etapa inicia-se após a invocação do método `replay`

na linha 9. Em seguida, na linha 10, o método `registerObserver` acrescenta o objeto *mock* à lista de objetos interessados nas alterações do objeto referenciado por `subjectImplementation`. Como trata-se de uma aplicação do padrão de projeto *Observer* (Gamma et al., 1995), logo após a alteração do atributo mantido pela instância de `SubjectImplementation` (linha 11), todas instâncias que manifestaram interesse nas alterações (*observers*) são notificadas, ou seja, é invocado o método `update` de cada instância. A ocorrência desse comportamento é verificada no final da segunda etapa por meio do método `verify`, linha 17. Caso o método `update` não seja invocado ocorre uma exceção: `java.lang.AssertionError`.

Listagem B.4: Etapas 1 e 2: gravação e verificação do comportamento

```
1  @Test
2  public void testNotify() {
3     //cria-se o objeto mock
4     Observer mockObserver = EasyMock.createMock( Observer.class );
5     //1 etapa -- especifica-se o comportamento esperado
6     mockObserver.update(); //"grava" uma chamada ao metodo update
7
8     //inicio da 2 etapa
9     EasyMock.replay( mockObserver );
10    subjectImplementation.registerObserver( mockObserver );
11    subjectImplementation.setInformation( "test-driven development" );
12
13    Assert.assertEquals( "test-driven development",
14    subjectImplementation.getInformation() );
15
16    //verifica a ocorrencia do comportamento especificado
17    EasyMock.verify( mockObserver ); //fim da 2 etapa
18 }
```

B.4.1 Especificação do Comportamento

O exemplo anterior demonstra somente um método que não retorna nenhum valor – tipo **void**. Para tais métodos, basta invocá-los antes do método `replay` para que o comportamento seja especificado (gravado). Métodos que possuem outro tipo de retorno precisam, na primeira etapa, especificar o valor que deve ser retornado. Por exemplo, considerando o método `readByte` da interface `java.io.DataInput`, uma das formas de se especificar que o método deve retornar (**byte**) `99` quando for invocado na segunda etapa é como demonstrado na Listagem B.5, linha 7.

O método `expect` juntamente com métodos definidos na interface `IExpectationSetters` definem os valores que devem ser retornados quando os métodos forem invocados na segunda etapa. Também é possível, utilizando o método `andThrow`, especificar que uma exceção será lançada quando o método for invocado. O método

`andThrow` é utilizado de maneira semelhante ao método `andReturn`, como pode ser visualizado na Listagem B.6. Detalhes adicionais sobre a especificação do comportamento utilizando o EasyMock podem ser obtidos na documentação distribuída em conjunto com a biblioteca.

Listagem B.5: Refinando a especificação do comportamento

```
1  @Test
2  public void testDataInput() throws IOException {
3
4     DataInput dataInputMock = EasyMock.createMock( DataInput.class );
5     //especifica que quando o metodo readByte for invocado o
6     //valor 99 (byte) sera retornado -- etapa de "gravacao"
7     EasyMock.expect( dataInputMock.readByte() ).andReturn( (byte) 99 );
8
9     //inicio da segunda etapa
10    EasyMock.replay( dataInputMock );
11    //invocacao do metodo e teste que avalia
12    //o valor retornado
13    Assert.assertEquals( (byte) 99, dataInputMock.readByte() );
14    EasyMock.verify( dataInputMock ); //verifica
15 }
```

Listagem B.6: Indicando o lançamento de exceções

```
EasyMock.expect( dataInputMock.readByte() )
.andThrow( new IllegalArgumentException() );
```

B.5 Reutilizando Objetos *Mock*

Algumas vezes é coerente reutilizar os objetos *mock* existentes a fim de poupar recursos computacionais. Usando o método `reset`, da biblioteca EasyMock, os objetos *mock*, criados anteriormente, podem ser novamente configurados com um outro comportamento. Isto é, pode-se alterar os métodos que deverão ser chamados na segunda etapa, o valor de retorno de tais métodos e exceções que serão lançadas. A Listagem B.7 fornece um exemplo da utilização desse método.

Listagem B.7: Utilizando o método `reset`

```
//reutilizando dois objetos mock
//ambos voltam para o estado de "gravacao"
EasyMock.reset( mockObserver, anotherMockObserver );
```

Depois que o método `reset` é invocado os objetos *mock* voltam para o estado de gravação – segunda etapa do teste utilizando objetos *mock*. O efeito do método `reset` é ilustrado na Figura B.4.

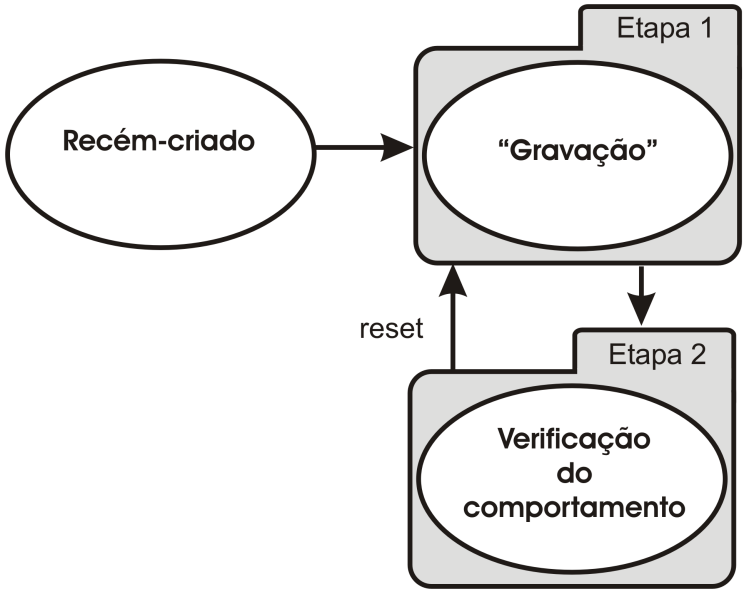


Figura B.4: Conseqüência da invocação do método `reset` em relação aos objetos *mock*

APÊNDICE

C

Protocolo de Reengenharia

C.1 O *Protocolo de Reengenharia*

Um documento denominado *protocolo de reengenharia* foi preenchido para cada classe durante a reengenharia do framework GREN. Esses documentos contêm tanto a "versão" em Smalltalk de todos os métodos de uma classe, quanto a "versão" em Java™obtida ao final da reengenharia da classe, bem como informações sobre os problemas encontrados durante a leitura do código Smalltalk, observações, atributos e *protocolos*[1] da classe. A seguir apresenta-se a organização dessas informações nesse documento.

C.1.1 Informações Gerais

NOME DA CLASSE:	**NOME DA CLASSE PAI:**
PROTOCOLOS:	**ATRIBUTOS:**
	CATEGORIA:

Nome da classe: indica o nome da classe que está passando (ou passou) por reengenharia.

Nome da classe pai: indica o nome da classe pai (superclasse) da classe que está passando (ou passou) por reengenharia.

[1] Em Smalltalk os métodos de uma classe são organizados em grupos lógicos chamados *protocolos*

Protocolos: lista com o nome de todos os *protocolos* da classe.

Atributos: lista com todos os atributos da classe.

Categoria: categoria (*category*) em que a classe encontra-se.

Problemas identificados: descreve os problemas encontrados durante a leitura do código.

Observações: documenta informações que podem auxiliar na reengenharia da classe.

C.1.2 Informações Relacionadas a cada Método

PROTOCOLO:	**PARTE:**
NOME DO MÉTODO:	**NOTA:**
DEFINIÇÃO NO GREN	
POSSÍVEL IMPLEMENTAÇÃO NO GRENJ	

PROTOCOLO: o nome do *protocolo* em que o método encontra-se.

PARTE: *instance* ou *class*; informa se o método é de instância ou de classe.

Nome do método: nome do método na linguagem Smalltalk.

Nota: informações relevantes relacionadas ao método.

Definição no GREN: como o método encontra-se implementado no framework GREN; o código do método.

Possível implementação no GRENJ: "versão" do método acima mencionado implementada na linguagem Java™.

Referências Bibliográficas

Agerbo, E. e Cornils, A. (1998). How to Preserve the Benefits of Design Patterns. *ACM SIGPLAN Notices*, 33.

Apache Ant (2008). The Apache Ant Project. Disponível em: *http://ant.apache.org/*. (Acessado 24 de Abril de 2008).

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley Professional.

Beck, K. (2007). *Implementation Patterns*. Addison-Wesley.

Beck, k., Coplien, J., Crocker, R., Dominick, L., Meszalos, G., e Paulish, F. (1996). Industrial Experience with Design Patterns. *Proceedings of the 18th International Conference on Software Engineering*, pages 103–114.

Bianchi, A., Caivano, D., Marengo, V., e Visaggio, G. (2003). Iterative Reengineering of Legacy Systems. *IEEE Transactions on Software Engineering*, 29:225–241.

Bloch, J. (2001). *Effective Java^{TM}Programming Language Guide*. Prentice Hall PTR.

Braga, R. T. V. (2002a). GRN: Uma linguagem de padrões para gerenciamento de recursos de negócio. Disponível em: *http://www.icmc.usp.br/∼rtvb/*. (Acessado 25 de Novembro de 2006).

Braga, R. T. V. (2002b). *Um processo para construção e instanciação de frameworks baseados em uma linguagem de padrões para um domínio específico*. Tese de doutorado, ICMC/USP, São Carlos - SP.

Braga, R. T. V., Germa no, F. S. R., e Masiero, P. C. (1999). A Pattern Language for Business Resource Management. *Proceedings of the Annual Conference on Pattern Languages of Programs*, 6:1–33.

Braga, R. T. V. e Masiero, P. (2002a). A Process for Framework Construction Based on a Pattern Language. *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*.

Braga, R. T. V. e Masiero, P. (2002b). Manual de Instanciação do Framework GREN usando a GRN. ICMC/USP, São Carlos - SP. documento de trabalho, 94p.

Braga, R. T. V. e Masiero, P. (2002c). The Role of Pattern Languages in the Instantiation of White-Box Object-oriented Frameworks. *Cadernos de Computação do ICMC*, 3:119–145.

Brugali, D. e Sycara, K. P. (2000). Frameworks and Pattern Languages: an Intriguing Relationship. *ACM Computing Surveys*, 32.

Cagnin, M. I. (2005). *PARFAIT: uma contribuição para a reengenharia de software baseada em linguagens de padrões e frameworks*. Tese de doutorado, ICMC/USP, São Carlos - SP.

Camargo, V. V. (2006). *Frameworks transversais: definições, classificações, arquitetura e utilização em um processo de desenvolvimento software*. Tese de doutorado, ICMC/USP, São Carlos - SP.

Chikofsky, E. J. e Cross II, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17.

Coady, Y., Kiczales, G., Feeley, M., e Smolyn, G. (2001). Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*.

Cockburn, A. (2006). Dos equis driven design. Disponível em: *http://alistair.cockburn.us/index.php/Dos_equis_driven_design*. (Acessado 19 de Fevereiro de 2008).

Coplien, J. (1996). *Software Patterns*. Sigs Books.

Cortes, M., Fontoura, M., e Lucena, C. (2003). Using Refactoring and Unification Rules to Assist Framework Evolution. *ACM SIGSOFT Software Engineering Notes*, 28.

Cunningham, H., Liu, Y., e Zhang, C. (2006). Using Classic Problems to Teach Java Framework Design. *Science of Computer Programming*, 59.

Cunningham, H. C., Liu, Y., e Zhang, C. (2004). Using The Divide and Conquer Strategy to Teach Java Framework Design. *Principles and practice of programming in Java*.

Cunningham, H. C., Tadepalli, P., e Liu, Y. (2005). Secrets, Hot Spots, and Generalization: Preparing Students to Design Software Families. *Journal of Computing Sciences in Colleges*, 20.

Demeyer, S. (2005). Refactor Conditionals into Polymorphism: What's the Performance Cost of Introducing Virtual Calls? *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*.

Demeyer, S., Du Bois, B., e Verelst, J. (2004). Refactoring - Improving Coupling and Cohesion of Existing Code. *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*.

Demeyer, S., Ducasse, S., Hirschfeld, R., Jazayeri, M., Mens, T., e Wermelinger, M. (2005a). Challenges in Software Evolution. *Principles of Software Evolution, Eighth International Workshop on*, pages 13–22.

Demeyer, S., Ducasse, S., e Nierstrasz, O. (2002). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann.

Demeyer, S., Ducasse, S., e Nierstrasz, O. (2005b). Object-Oriented Reengineering: Patterns and Techniques. *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 723–724.

Deursen, A., Moonen, L., Bergh, A., e Kok, G. (2001). Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*.

Ducasse, S., Demeyer, S., Ciupke, O., e Tichelaar, S. (1999a). The FAMOOS Object-Oriented Reengineering Handbook (as released to the general public). Disponível em: *http://www.iam.unibe.ch/∼famoos/handbook/4handbook.pdf*. (Acessado 23 de Fevereiro de 2007).

Ducasse, S., Richner, T., e Nebbe, R. (1999b). Type-Check Elimination: Two Object-Oriented Reengineering Patterns. *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, pages 157–166.

Durelli, V. H. S., Viana, M. C., e Penteado, R. A. D. (2008). Uma Proposta de Reúso de Interface Gráfica com o Usuário Baseada no Padrão Arquitetural MVC. *IV Simpósio Brasileiro de Sistemas de Informação*, pages 48–59.

EasyMock (2008). EasyMock. Disponível em: *http://www.easymock.org/*. (Acessado 15 de Janeiro de 2008).

EclEmma (2008). EclEmma 1.3.1 – Java Code Coverage for Eclipse. Disponível em: *http://www.eclemma.org/*. (Acessado 1 de Março de 2008).

Eclipse (2007). The Eclipse Foundation. Disponível em: *http://www.eclipse.org/*. (Acessado 20 de Fevereiro de 2007).

Elrad, T., Aksits, M., Kiczales, G., Lieberherr, K., e Ossher, H. (2001). Discussing Aspects of AOP. *Communications of the ACM*, 44.

Engelbrecht, R. L. e Kourie, D. G. (2003). Translating Smalltalk Blocks to Java. *IEE Proceedings - Software*, 150:203–211.

Erdogmus, H., Morisio, M., e Torchiano, M. (2005). On the Effectiveness of the Test-First Approach to Programming. *Software Engineering, IEEE Transactions on*, 31:226–237.

Fach, P. (2001). Design Reuse Through Frameworks and Patterns. *Software, IEEE*, 18.

Fayad, M. e Schmidt, D. C. (1997). Object-Oriented Application Frameworks. *Communications of the ACM*, 40.

Filho, F., Cacho, N., Ferreira, R., Figueiredo, E., Garcia, A., e Rubira, C. (2006). Exceptions and Aspects: The Devil is in the Details. *nternational Conference on Foundations on Software Engineering*.

Forman, I. R. e Forman, N. (2004). *Java Reflection in Action*. Manning Publications.

Fowler, M. (1996). *Analysis Patterns: Reusable Object Models*. Addison-Wesley.

Fowler, M. (2001). Avoiding Repetition. *Software, IEEE*, 18:97–99.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Fowler, M. (2003). Patterns. *Software, IEEE*, 20:2–3.

Fowler, M. (2005). UML E*ssencial:* U*m breve guia para a linguagem-padrão de modelagem de objetos*. Bookman. 3° Edição.

Fowler, M. (2007). Mocks Aren't Stubs. Disponível em: *http://martinfowler.com/articles/mocksArentStubs.html*. (Acessado 23 de Janeiro de 2008).

Fowler, M., Beck, K., Brant, J., Opdyke, W., e Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Freeman, S., Mackinnon, T., Pryce, N., e Walnes, J. (2004). Mock roles, not objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236–246.

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

Grand, M. (1998). *Patterns in Java*, volume 1. John Wiley & Sons.

Guerra, E. M. e Fernandes, C. T. (2007). Refactoring Test Code Safely. In *ICSEA '07: Proceedings of the International Conference on Software Engineering Advances (ICSEA 2007)*, pages 44–50.

Hanenberg, S. (2003). Idioms for Building Software Frameworks in AspectJ. *AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*.

Hannemann, J. e Kiczales, G. (2002). Design Pattern Implementation in Java and AspectJ. *ACM SIGPLAN Notices*, 37.

Harrison, N. B., Avgeriou, P., e Zdun, U. (2007). Using Patterns to Capture Architectural Decisions. *IEEE Software*, 24(4):38–45.

Hirschfeld, R. (2001). AspectS: AOP with squeak. *OOPSLA 2001 Workshop on Advanced Separation of Concerns*.

Huang, L. e Holcombe, M. (2006). Empirical Assessment of Test-First Approach. In *TAIC-PART '06: Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, pages 197–202. IEEE Computer Society.

Janzen, D. e Saiedian, H. (2005). Test-Driven Development: Concepts, Taxonomy, and Future Direction. *Computer*, 38(9):43–50.

Janzen, D. e Saiedian, H. (2006). On the Influence of Test-Driven Development on Software Design. In *CSEET '06: Proceedings of the 19th Conference on Software Engineering Education & Training (CSEET'06)*, pages 141–148. IEEE Computer Society.

Jeffries, R. e Melnik, G. (2007). Guest Editors' Introduction: TDD–The Art of Fearless Programming. *IEEE Software*, 24(3):24–30.

Johnson, R. E. (1992). Documenting Frameworks Using Patterns. *ACM SIGPLAN Notices*, 27.

Johnson, R. E. (1997a). Components, Frameworks, Patterns. *ACM SIGSOFT Software Engineering Notes*, 22.

Johnson, R. E. (1997b). Frameworks = (components + patterns). *Communications of the ACM*, 40.

JUDE (2008). JUDE UML Modeling Tool. Disponível em: *http://jude.change-vision.com/jude-web/product/community.html*. (Acessado 24 de Abril de 2008).

JUnit (2007). JUnit.org Resources for Test Driven Development. Disponível em: *http://www.junit.org/*. (Acessado 11 de Dezembro de 2007).

Kerievsky, J. (2004). *Refactoring to Patterns*. Addison-Wesley Professional.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., e Griswold, W. (2001). Getting started with AspectJ. *Communications of the ACM*, 44:59–65.

Koskela, L. (2007). *Test Driven: TDD and Acceptance TDD for Java Developers*. Manning Publications.

Krueger (1992). Software Reuse. *ACM Computing Surveys*, 24.

Kulesza, U., Sant´ Anna, C., e Lucena, C. (2005). Refactoring the JUnit Framework Using Aspect-Oriented Programming. *ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*.

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning.

Larman, C. (2004). *Utilizando UML e Padrões: Uma introdução à análise e ao projeto orientados a objetos e ao* P*rocesso* U*nificado*. Bookman. 2° Edição.

Lehman, M. (1996). Laws of Software Evolution Revisited. In *European Workshop on Software Process Technology*, pages 108–124. Springer.

Lippert, M. e Lopes, C. (2000). A Study on Exception Detection and Handling Using Aspect-Oriented Programming. *International Conference on Software Engineering*, 00.

Lohmann, D., Blaschke, G., e Spinczyk, O. (2004). Generic advice: On the combination of AOP with generative programming in AspectC++. *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE '04)*, 3286:55–74.

Louridas, P. (2005). JUnit: Unit Testing and Coding in Tandem. *IEEE Software*, 22(4):12–15.

Mahrenholz, D., Spinczyk, O., e Oder-preikschat, W. S. (2002). Program Instrumentation for Debugging and Monitoring with AspectC++. *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*.

Mäntylä, M. V., Vanhanen, J., e Lassenius, C. (2004). Bad Smells – Humans as Code Critics. *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, 00:399–408.

Martin, R. C. (2007). Professionalism and Test-Driven Development. *IEEE Software*, 24(3):32–36.

Melis, M., Turnu, I., Cau, A., e Concas, G. (2006). Evaluating the impact of test-first programming and pair programming through software process simulation. *Software Process: Improvement and Practice*, 11(4):345–360.

Mens, T. e Tourwe, T. (2004). A Survey of Software Refactoring. *Transactions on Software Engineering, IEEE*, 30.

Meszaros, G. (2007). *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.

Miller, S. (2001). Aspect-Oriented Programming Takes Aim at Software Complexity. *Computer, IEEE*, 34.

Monteiro, M. e Fernandes, J. (2005a). Refactoring a Java Code Base to AspectJ - An Illustrative Example. *Proceedings of the 21st IEEE International Conference on Software Maintenance*.

Monteiro, M. e Fernandes, J. (2005b). Towards a Catalog of Aspect-Oriented Refactorings. *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 111–122.

Müller, H. A., Janhke, J. H., Smith, D. B., Storey, M.-A., Tilley, S. R., e Wong, K. (2000). Reverse Engineering: A Roadmap. In Finkelstein, A., editor, *The Future of Software Engineering 2000*. ACM Press.

Müller, M. M. e Hagner, O. (2002). Experiment about test-first programming. *IEE Proceedings – Software*, 149(5):131–136.

Müller, M. M. e Höfer, A. (2007). The effect of experience on the test-driven development. *Empirical Software Engineering*, 12(6):593–615.

Murphy, G. e Schwanninger, C. (2006). Guest Editors Introduction: Aspect-Oriented Programming. *Software, IEEE*, 23.

Myers, G. J., Sandler, C., Badgett, T., e Thomas, T. M. (2004). *The Art of Software Testing*. Wiley.

MySQL (2007). MySQL. Disponível em: *http://www.mysql.com/*. (Acessado 1 de Março de 2007).

Niemeyer, P. e Knudsen, J. (2005). *Learning Java™*. O' Reilly. 3° Edição.

OMG (2008). Object Management Group (OMG) Unified Modeling Language (UML), Infrastructure, V2.1.2 — OMG Available Specification without Change Bars. Disponível em: *http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/*. (Acessado 2 de Fevereiro de 2008).

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. Ph.D. Thesis, University of Illinois.

Parsons, D., Rashid, A., Speck, A., e Telea, A. (1999). A "framework" for Object Oriented Frameworks Design. *Technology of Object-Oriented Languages and Systems, 1999. Proceedings of*.

Rashid, A. e Chitchyan, R. (2003). Persistence as an Aspect. *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 120–129.

Shalloway, A. e Trott, J. (2001). *Design Patterns Explained: A New Perspective On Object-Oriented Design*. Addison-Wesley.

Sharp, A. (1997). *Smalltalk by Example*. McGraw-Hill.

Shiva, S. G. e Shala, L. A. (2007). Software Reuse: Research and Practice. *International Conference on Information Technology*, pages 603–609.

Silva, M. T. F., Braga, R. T. V., e Masiero, P. C. (2004). Evolução Orientada a Aspectos de um Framework OO. *Workshop de Manutenção de Software Moderna*, 1.

Soares, S., Laureano, E., e Borba, P. (2002). Implementing Distribution and Persistence Aspects with AspectJ. *ACM SIGPLAN Notices*, 37.

Srinivasan, S. (1999). Design Patterns in Object-Oriented Frameworks. *Computer, IEEE*, 32:24–32.

Stevens, P. e Pooley, R. (1998). Systems Reengineering Patterns. *SIGSOFT Softw. Eng. Notes*, 23(6):17–23.

Stuckert, R. (2006). JUnit Reloaded. Disponível em: *http://today.java.net/pub/a/today/2006/12/07/junit-reloaded.html#omitting-tests*. (Acessado 13 de Julho de 2007).

Sun Microsystems, Inc. (1997). Code Conventions for the Java Programming Language. Disponível em: *http://java.sun.com/docs/codeconv/*. (Acessado 25 de Junho de 2007).

Sun Microsystems, Inc. (2007). Sun Developer Network (SDN) - Javadoc Tool Home Page. Disponível em: *http://java.sun.com/j2se/javadoc/*. (Acessado 26 de Fevereiro de 2007).

Terekhov, A. e Verhoef, C. (2000). The Realities of Language Conversions. *Software, IEEE*, 17.

Thomas, D. e Hunt, A. (2002). Mock Objects. *IEEE Software*, 19(3):22–24.

Vermeulen, A., Ambler, S. W., Bumgardner, G., Metz, E., Misfeldt, T., Shur, J., Vermeulen, A., e Thompson, P. (2000). *The Elements of Java Style*. Cambridge University Press.

Viega, J. e Voas, J. (2000). Can Aspect-Oriented Programming Lead to More Reliable Software? *Software, IEEE*, 17.

Williams, L., Maximilien, E. M., e Vouk, M. (2003). Test-Driven Development as a Defect-Reduction Practice. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 34–48. IEEE Computer Society.

Wirfs-Brock, R. (2006). Refreshing Patterns. *Software, IEEE*, 23:45–47.

Wirfs-Brock, R. (2007). Driven to . . . Discovering Your Design Values. *IEEE Software*, 24(1):9–11.

Woolf, B. (1996). The Null Object Pattern. In *Pattern Language of Programming (PloP)*.

Yoder, J. W., Johnson, R. E., Wilson, Q. D., e Douglas, M. (1998). Connecting Business Objects to Relational Databases. *Fifth Conference on Patterns Languages of Programs (PLoP '98)*.