UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

“Uso de um Framework Transversal na camada de persistência do GRENJ”

Ivan Botacini Zanon

São Carlos/SP

Maio/2009

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

Z33uf

Zanon, Ivan Botacini.
Uso de um framework transversal na camada de persistência do GRENJ / Ivan Botacini Zanon. -- São Carlos : UFSCar, 2009.
93 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2009.

1. Análise e projeto de sistemas. 2. Framework (Programa de computador). 3. Orientação a aspectos. 4. Software - manutenção. I. Título.

CDD: 004.21 (20ª)

Agradecimentos

Em primeiro lugar, agradeço a Deus, pela oportunidade de estar vivo e aprendendo cada dia mais nesse grande ginásio psicológico que é a vida.

Agradeço à professora Rosangela Ap. D. Penteado, em primeiro pela oportunidade de ingressar nesta que foi uma das mais engrandecedoras experiências de minha vida, e em segundo por toda confiança e apoio dados a mim desde o início desta caminhada.

Agradeço à minha família, pelo apoio fundamental e incondicional durante todo o tempo dedicado à atividade que resultou neste projeto.

Agradeço aos meus amigos de longa data, em especial a Leonardo e Dayane, pela paciência e por todo apoio dado a esse amigo que ainda não sabe se merece.

Agradeço aos colegas e amigos do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, em especial aos meus contemporâneos, por toda amizade, companheirismo e auxílio.

Agradeço aos integrantes do GDMS, com quem tive a oportunidade, honra e prazer de dividir mais de perto toda essa experiência, com todas as suas alegrias e dificuldades.

Agradeço aos integrantes do Grupo de Estudos das Obras Básicas do Centro Espírita Obreiros do Bem, por me possibilitarem crescimento, conhecimento e auxílio espiritual, todos imprescindíveis para o meu bom aproveitamento.

Agradeço aos trabalhadores do Posto de Rua Eurípedes Barsanulfo, pela oportunidade constante de crescimento e de aprendizagem.

Agradeço aos integrantes da República do Limão, em especial a Will, Helder e Klebson, por todo o companheirismo, amizade e refúgio em momentos de crises existenciais, recorrente em estudantes de mestrado.

Agradeço, em especial, à amiga Juliana Martinelli, por sua companhia, preocupação e dedicação. Isso nunca será esquecido.

Agradeço à CAPES pelo apoio financeiro durante a execução do projeto.

Enfim. Agradeço a todos que participaram de alguma maneira nesta grande empreitada. Do fundo do meu coração, muito obrigado a todos vocês.

Resumo

A reutilização de artefatos de software, código, conceitos e modelos existentes em sistemas já construídos torna o desenvolvimento de software mais ágil, bem como propicia processos e produtos com mais qualidade. Frameworks de aplicação representam uma técnica de reúso que se caracteriza pelo modelo de uma aplicação semi-completa que atende às especificações de um determinado domínio e que deve ser customizado para atender aos requisitos específicos do sistema de software que se pretende desenvolver. Frameworks podem ser construídos sob o paradigma de programação orientada a objetos ou da programação orientada a aspectos. Frameworks transversais, um tipo especial de framework orientado a aspectos, são estruturas semi-completas que implementam o tratamento de um interesse transversal específico e podem ser acoplados a qualquer tipo de código. Porém, a aplicação de frameworks transversais em uma estrutura que ainda será especializada, como um framework de aplicação, merece atenção e há pouca informação na literatura especializada sobre o assunto. Esta dissertação de mestrado apresenta o framework de aplicação GRENJ-FT, oriundo da inserção de um framework transversal, responsável pelo tratamento do interesse de persistência, na estrutura do framework de aplicação GRENJ. GRENJ-FT possui o interesse de persistência isolado de maneira adequada, o que facilita sua compreensão e a instanciação de novas aplicações com o seu uso. Também são apresentadas as manutenções realizadas nas estruturas dos frameworks GRENJ e do transversal utilizado, para possibilitar tal acoplamento. Estudos de caso foram realizados, para verificar que tanto o processo de instanciação de aplicações quanto a funcionalidade oferecida pelo GRENJ se mantiveram inalteradas para o GRENJ-FT.

Abstract

Reuse of software artifacts, code, concepts and models of existing systems makes software development more agile, as well as provides higher quality process and products. Application Frameworks represents a reuse technique characterized by a model of an uncompleted application that satisfies the specifications of a certain domain and has to be customized to satisfy the requirements of a specific software system. Frameworks can be built under the object or aspect-oriented programming paradigms. Crosscutting Frameworks, a special kind of aspect-oriented framework are uncompleted structures that implements specifics crosscutting concerns that can be coupled to any base code. Therefore, the use of crosscutting frameworks in a generic structure that will be specialized, as an application framework, deserves attention and there is a lack of information about this topic in the literature. This master's thesis presents the GRENJ-FT application framework which was built from the insertion of a crosscutting framework, responsible for the persistence concern, to the GRENJ application framework structure. GRENJ-FT has the persistence concern isolated in a suitable way, what eases its comprehension and new application instantiations by users. Also the maintenances made in the GRENJ and crosscutting framework's structures, to enable this coupling are presented. Case Studies were conducted to verify that application's instantiations and functionality provided by the GRENJ remained the same in the GRENJ-FT.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

CRUD – *Create, Read, Update and Delete*

FAOA – Framework de Aplicação Orientada a Aspectos

FT – Framework Transversal

LPS – Linha de Produtos de Software

MVC – *Model, View, Controller*

OA – Orientação a Aspectos

OO – Orientação a Objetos

SOA – *Service Oriented Architecture*

TDD – *Test Driven Development*

CAPÍTULO 1
Introdução

1.1 Considerações Iniciais

O aumento constante do mercado de desenvolvimento de software faz com que a complexidade dos sistemas computacionais cresça juntamente com a exigência de qualidade e de menores prazos de desenvolvimento. Os desenvolvedores de software têm a responsabilidade de construir um número cada vez maior de produtos, com custos e prazos reduzidos. Além disso, a dinâmica do atual mercado de negócios exige que softwares sofram constantes modificações, o que demanda aumento de trabalho aos desenvolvedores. Nesse contexto, a engenharia de software estuda e estabelece práticas que permitam a criação de software de maneira mais rápida e organizada, diminuindo o custo de sua construção e de sua manutenção. Para os engenheiros que não utilizam essas práticas, o desenvolvimento pode tornar-se cada vez mais difícil, havendo demora no desenvolvimento, custos exageradamente altos, e dificuldades na compreensão do software desenvolvido, que muitas vezes inviabiliza a finalização de um produto ou dificulta sua manutenção.

Uma das técnicas existentes para o desenvolvimento de software com qualidade é a de modularização. Essa técnica define o desenvolvimento de sistemas agrupando os seus requisitos em unidades lógicas que interagem entre si para a realização de suas

tarefas, de tal modo que facilite a legibilidade, a organização e a manutenibilidade de sistemas. Um paradigma de desenvolvimento que permite a modularização de sistemas computacionais é o de orientação a objetos (OO), que separa os requisitos em classes, agrupando suas informações e operações [Klump 2001]. Porém, a OO pode apresentar problemas em relação à modularização de requisitos que influenciam vários módulos de um mesmo sistema. A implementação desses requisitos segundo os conceitos definidos para esse paradigma gera espalhamento e entrelaçamento de código. Para o isolamento adequado desses requisitos, conhecidos como interesses transversais, pode ser utilizado o paradigma de desenvolvimento orientado a aspectos (OA) [Kiczales *et al.* 1997, Laddad 2003]. Esse paradigma utiliza todos os conceitos presentes na OO com muitas de suas vantagens, mas conta com o uso de unidades lógicas chamadas de aspectos, que possuem o comportamento para o tratamento dos interesses transversais e a definição dos pontos onde esse comportamento deve ser realizado.

A modularização adequada de um sistema possibilita o uso de outra técnica que também auxilia o desenvolvimento de software, a de reúso. O reúso de software se caracteriza pelo uso de artefatos de software existentes na construção de novos sistemas computacionais [Krueger 1992, Frakes e Kang 2005]. Dentre as diversas técnicas de reúso existentes encontram-se os frameworks. Um framework é o esqueleto de um sistema de software semi-completo que atende às especificações de um determinado domínio e que deve ser customizado para atender aos requisitos específicos do sistema a ser desenvolvido [Johnson 1997]. Frameworks apóiam a instanciação de sistemas promovendo o reúso de projeto e de análise, diminuindo o esforço por parte do desenvolvedor, tanto na compreensão de parte do domínio do sistema quanto no desenvolvimento de detalhes comuns dos sistemas desse domínio. Dessa forma há ganho em qualidade dos sistemas instanciados e em tempo de desenvolvimento, pois boa parte do software já está testada e pronta para ser reusada.

Um dos tipos de frameworks mais usados no desenvolvimento de softwares são os frameworks de aplicação. Esse tipo de estrutura tem o enfoque na instanciação de aplicações e se destacam entre outras técnicas por reusarem software em nível de projeto, auxiliando os desenvolvedores na construção de sistemas pertencentes a um domínio específico. Porém, para que um framework de aplicação seja utilizado é necessário que o desenvolvedor tenha conhecimento da sua estrutura. Dessa maneira, é

importante que um framework seja construído de acordo com as práticas preconizadas pela engenharia de software, principalmente quanto à modularização de código, para garantir a sua utilização com menor grau de dificuldade. Isso é possível pois a modularização adequada do código do framework auxilia o entendimento e, por conseqüência, o reúso de sua estrutura por parte do desenvolvedor, além de melhorar a sua manutenibilidade e a dos sistemas por ele instanciados.

Outro tipo de framework é o transversal, uma estrutura orientada a aspectos que possui mecanismos de acoplamento abstratos, podendo assim entrecortar pontos de qualquer estrutura de software [Camargo e Masiero 2005]. Essas estruturas não geram aplicações novas, mas têm o objetivo de entrecortar um código para realizar o isolamento de algum interesse transversal. Esse entrecorte pode, em teoria, ser realizado em qualquer tipo de software, como componentes, classes, ou mesmo frameworks de aplicação.

1.2 Motivação e Objetivos

Frameworks são estruturas de reúso cada vez mais utilizadas para o desenvolvimento de sistemas. Existem diversos exemplos de frameworks desenvolvidos recentemente com o objetivo de atender aos mais variados domínios, como rede de sensores [Chou *et al*. 2008, Chung *et al*. 2008, Yue *et al*. 2008], sistemas web [Cao *et al*. 2008, Liao e Koonse 2007, Sacowicz 2007] e aplicações comerciais [Li e Qiang, 2008]. Uma das preocupações durante o desenvolvimento de frameworks é com relação à facilidade de utilização, legibilidade e manutenibilidade [Cortes *et al*. 2003, Dagenais e Robillard 2008].

GREN [Braga 2004] é um exemplo de framework de aplicação que pode ser usado para a instanciação de sistemas inseridos no domínio de gestão de recursos de negócio. Ele foi desenvolvido com base em uma linguagem de padrões de análise e foi implementado em linguagem *Smalltalk*[1]. Apesar de possibilitar a instanciação de aplicações de forma adequada, o fato da linguagem de programação *Smalltalk* ser pouco utilizada gerava dificuldades tanto na utilização do framework quanto na sua manutenção. Assim, o framework GREN passou por um processo de reengenharia

[1] http://www.smalltalk.org

iterativa em que foi implementado com a linguagem Java, resultando em um novo framework denominado GRENJ [Durelli 2008]. O mecanismo para tratamento da persistência implementado na estrutura do GRENJ apresenta problemas de entrelaçamento de código com a camada de negócios do framework. Esse fato, além de dificultar o processo de instanciação de novas aplicações, contribui para aumentar o custo de futuras evoluções dessa estrutura.

Uma das técnicas que pode ser utilizada para auxiliar a compreensão e evolução de frameworks de aplicação é a de orientação a aspectos. Lobato (2008) observa que frameworks que têm os seus interesses devidamente isolados com o auxílio da programação OA possuem maior flexibilidade ao passar por manutenções evolutivas do que frameworks puramente orientados a objetos. O isolamento de interesses em um código OO com o apoio da OA pode ser feito de diversas maneiras, como por exemplo, utilizando frameworks transversais (FT). Esse tipo especial de framework é implementado segundo a orientação a aspectos de modo a se acoplar a outro código existente para isolar de forma adequada um único interesse transversal específico [Camargo e Masiero, 2005].

Camargo e Masiero (2008) desenvolveram uma família de frameworks transversais que se acopla a um código-base com o objetivo de realizar o tratamento de persistência de forma modularizada. O acoplamento do FT de persistência foi testado somente em sistemas de software finalizados, e sua aplicação na estrutura de um artefato reutilizável, que será posteriormente instanciado para a geração de uma aplicação, ainda não foi testada.

Este trabalho propõe a utilização de um produto da família de frameworks transversais de persistência para isolar de maneira adequada o interesse de persistência na estrutura do GRENJ, motivada principalmente por dois motivos:

1. É implementado na linguagem AspectJ, o que o torna compatível com a linguagem Java utilizada na construção do GRENJ;

2. Seu desenvolvimento foi baseado no padrão *Persistence Layer* [Yoder *et al.* 1998], também utilizado na estrutura do GRENJ, o que facilita a integração desses frameworks;

Com a realização deste trabalho tem-se a melhoria do isolamento do interesse de persistência presente no GRENJ, a melhoria da instanciação de aplicações com o uso desse framework de aplicação, a melhoria da manutenibilidade de sistemas instanciados com o framework GRENJ e a utilização de um framework transversal acoplado à estrutura de um framework de aplicação, verificando as adaptações necessárias para que isso ocorra. Após a conclusão do acoplamento do FT de Persistência na estrutura do GRENJ, estudos de caso devem ser conduzidos para verificar se a adaptação da camada de persistência não resultou em alterações no escopo ou no processo de instanciação de novas aplicações com o apoio desse framework.

1.3 Organização do Trabalho

Esta dissertação está organizada em seis capítulos com este introdutório. No Capítulo 2 são apresentados os conceitos de reúso de software bem como são comentadas algumas das principais técnicas de reúso utilizadas neste trabalho: linhas de produtos de software, padrões de software e frameworks.

No Capítulo 3 são apresentados com detalhes os frameworks utilizados neste trabalho: o framework de aplicação GRENJ, e a família de frameworks transversais de persistência, da qual um produto será acoplado à camada de persistência do framework de aplicação resultando no GRENJ-FT.

No Capítulo 4 a manutenção realizada para a obtenção do GRENJ-FT é apresentada e tanto o processo de manutenção utilizado quanto as alterações realizadas para que os dois frameworks trabalhem juntos são detalhados.

No Capítulo 5 são mostrados alguns estudos de caso realizados com a instanciação do GRENJ-FT, para verificar que o processo de instanciação desse framework é o mesmo que o realizado com o framework GRENJ. Há interesse em avaliar o custo de manutenção quando sistemas já instanciados com o GRENJ original passam a utilizar o GRENJ-FT como base para o seu funcionamento.

No Capítulo 6 estão listadas as principais contribuições e limitações deste trabalho, bem como são sugeridos alguns trabalhos de pesquisa que podem dar continuidade a este.

CAPÍTULO 2
Técnicas de Reúso

2.1 Considerações Iniciais

A engenharia de software é a ciência que se dedica, entre outras coisas, ao desenvolvimento de técnicas que permitam melhorar a produção de software, tanto em relação ao tempo de produção quanto à qualidade dos sistemas computacionais desenvolvidos. O desenvolvimento de software baseado em reúso é uma dessas técnicas e tem o objetivo de apoiar o desenvolvimento de sistemas de software de qualidade de maneira produtiva, utilizando conhecimentos passados na confecção de novos produtos. Essa reutilização de artefatos de software pode ocorrer em qualquer nível do desenvolvimento de um software, seja na análise, projeto ou implementação.

Dentre as técnicas de desenvolvimento de software baseados em reúso destacam-se os padrões de software e os frameworks. Padrões de software são soluções de sucesso para problemas recorrentes de um determinado domínio no desenvolvimento de software que podem ser utilizadas por desenvolvedores menos experientes. Frameworks são utilizados no desenvolvimento de sistemas pertencentes a um mesmo domínio em que ocorre o reúso de partes nele definidas que são estendidas para a criação de sistemas específicos. Dada a complexidade da estrutura de um framework, a separação de interesses existentes na sua estrutura é cada vez mais comum, garantindo

assim a modularização de certos interesses do domínio em que o framework está inserido e possibilitando que a sua manutenção seja facilitada.

Este capítulo tem o objetivo de apresentar os conceitos de linhas de produto de software, padrões de software, linguagem de padrões e de dois tipos de frameworks: orientados a objetos e orientados a aspectos. Na Seção 2.2 são apresentados conceitos gerais de reúso de software. Na Seção 2.3 são apresentados conceitos de linhas de produto de software. Na Seção 2.4 o conceito de padrões e de linguagem de padrões de software é apresentado, bem como exemplos dessas formas de reúso. Na Seção 2.5 é apresentada a teoria de frameworks orientados a objetos e de frameworks orientados a aspectos, e são citados alguns exemplos de frameworks desenvolvidos atualmente. Por fim, na Seção 2.6 são apresentados alguns conceitos de manutenção de software relacionados a frameworks. Na Seção 2.7 são apresentadas as considerações finais deste capítulo.

2.2 Reúso de Software

O conceito de reúso de software é definido pela prática de construção de sistemas computacionais fazendo-se uso de artefatos de software previamente existentes [Krueger 1992, Frakes e Kang 2005]. Essa prática de desenvolvimento não se limita somente ao aproveitamento de código, mas também faz uso de documentos de análise, de projeto ou mesmo da experiência de desenvolvedores em dados problemas de implementação. Dessa maneira, reúso mostra-se como uma solução que evita a repetição de trabalho no desenvolvimento de software fazendo uso de conhecimento conseguido com esforços passados [Liu e Yang 2007]. Na prática, o reúso de software ocorre desde o início da programação, mas a sua formalização ocorreu em 1968 em um estudo realizado por Doug Mcilroy, no qual foi proposto que a indústria de software se baseasse em componentes reusáveis [Frakes e Kang 2005, Shiva e Shala 2007, Liu e Yang 2007]. O conceito ganhou popularidade recentemente com a difusão global da prática de desenvolvimento de software [Shiva e Shala 2007].

Johnson (1997) afirma que a tecnologia ideal para ser reutilizada deve prover componentes que podem ser facilmente conectados. A sua utilização deve ser fácil, garantindo que o desenvolvedor não seja obrigado a conhecer minúcias do funcionamento interno do que se está reusando. Logo, a busca por essas características

torna o desenvolvimento de artefatos de software para reúso mais custoso, pois esses artefatos devem ser projetados da maneira mais genérica possível para que, além de serem de fácil utilização, possam também ser empregados em diferentes contextos [Rothenberger *et al.* 2003].

Artefatos de reúso podem ser classificados em duas categorias principais [Liu e Yang 2007]:

- Artefatos de reúso caixa-preta: artefatos reusáveis sem a necessidade de nenhuma modificação. São mais fáceis de serem reusados, mas atendem a um número mais restrito de sistemas.
- Artefatos de reúso caixa-branca: artefatos que necessitam de alguma adaptação para serem empregados em um domínio ou sistema. Podem ser aplicados a um número maior de ambientes devido à possibilidade de serem especializados.

Nos últimos vinte anos, diversos meios de reúso de software foram desenvolvidos [Shiva e Shala 2007]. Pesquisas nessa área resultaram no desenvolvimento de diversas técnicas como bibliotecas de funções, métodos e ferramentas de engenharia de domínio, reúso de projeto, padrões de software, arquiteturas de software voltadas para domínios específicos, componentização (*componentry*), geradores, etc.

Dentre as formas de reúso existentes, destacam-se as técnicas de padrões de software, linguagens de padrões, linha de produtos de software e frameworks [Lopes 2007, Shiva e Shala 2007] por possuírem maior nível de reusabilidade, praticidade na sua utilização e aplicabilidade em grande número de softwares. Esse capítulo detalhará essas técnicas de acordo com sua relevância neste trabalho.

2.3 Linhas de Produto de Software

Uma Linha de Produtos de Software (LPS) é caracterizada por um conjunto de sistemas de software que atendem a problemas diferentes de um mesmo domínio, possuindo algumas similaridades e variabilidades específicas entre si [Kuloor e Eberlein 2003, Frakes e Kang 2005]. Logo, os módulos que representam essas similaridades e variabilidades podem ser desenvolvidos previamente, sendo depois agrupados para a construção de um produto [Pacios 2007]. Todos os módulos que os produtos

compartilham podem ser reusados em todos os produtos. Isso reduz o custo e também aumenta a produtividade e qualidade do desenvolvimento de software [Kuloor e Eberlein 2003].

O desenvolvimento de uma Linha de Produtos de Software é dividido em dois passos principais: engenharia de domínio e engenharia de aplicação [Kuloor e Eberlein 2003, Ziadi *et al.* 2003, Pohl *et al.* 2005]. A Figura 2.1 mostra o processo geral de desenvolvimento de uma LPS.

Na engenharia de domínio é realizada a coleta, organização e armazenamento das informações e requisitos referentes ao domínio que a LPS pretende atender. Então, são definidos meios para que esses requisitos se relacionem. A engenharia de domínio é dividida em três atividades básicas [Ziadi *et al.*, 2003]:

- Análise de domínio: levantamento de informações sobre o domínio e definição dos pontos comuns e das variabilidades dos sistemas de software a serem desenvolvidos. Os requisitos levantados nessa análise são armazenados em um repositório de requisitos.
- Projeto de domínio: definição da arquitetura da linha de produtos. Nesse passo especifica-se a maneira como essas partes comuns e variáveis serão implementadas.
- Implementação do domínio: implementar os módulos assim como foram definidos na arquitetura, montando assim uma coleção de artefatos reusáveis que atendem a todos os requisitos da linha de produtos.

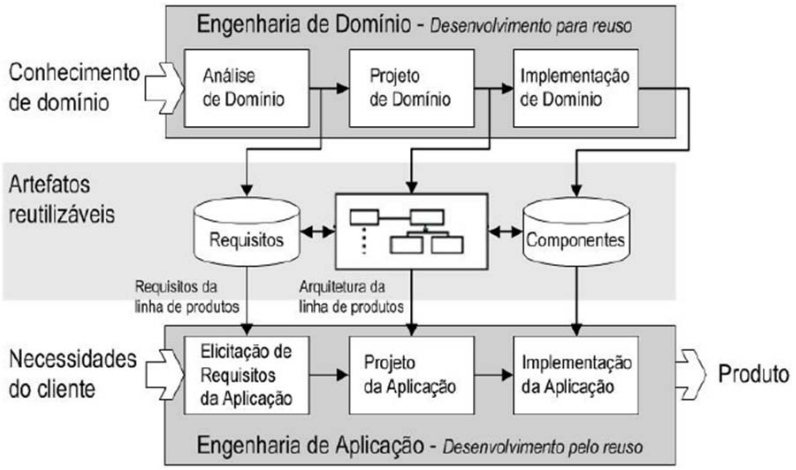


Figura 2.1: Processo geral de desenvolvimento de uma Linha de Produtos de Software.

O segundo passo caracteriza-se pela engenharia de aplicação, que consiste no agrupamento dos artefatos desenvolvidos na engenharia de domínio visando à construção de produtos de acordo com as necessidades de cada cliente.

Existem diversas abordagens para a realização da engenharia de domínio visando à definição dos artefatos que compõem uma linha de produtos de software. Alguns exemplos são as abordagens [Frakes e Kang 2005] FAST, DARE, KobrA, Koala e FODA/FORM.

A abordagem FODA/FORM é uma das abordagens que utiliza a abstração de características (*features*) de sistemas de um domínio para diferenciar o que é comum a todos os sistemas e o que é variável. Depois de identificadas, essas características são classificadas como:

- Obrigatórias (*common features*): devem estar presentes em qualquer produto derivado da LPS;
- Opcionais (*optional features*): requerido ao produto caso haja necessidade por parte do cliente;
- Alternativa (*alternative features*): a característica é selecionada entre algumas opções, onde uma delas pode ser a alternativa padrão;

Gomaa (2004) apresenta a abordagem PLUS, que oferece apoio de diagramas UML para representar as características de uma LPS. Nesse diagrama, as características são representadas por retângulos que trazem o nome da característica e estereótipos que definem a sua classificação. A Tabela 2.1 traz exemplos de estereótipos e seus significados. Essas características então são relacionadas quanto a sua dependência, sendo que uma característica pode ser composta por uma ou mais características.

Tabela 2.1. Estereótipos de classificação do modelo de características proposto por Gomaa (2005).

Estereótipo	Representação
<<common feature>>	Característica obrigatória
<<optional feature>>	Característica opcional
<<alternative feature>>	Característica alternativa
<<default feature>>	Característica padrão dentre as alternativas
<<parameterized feature>>	Característica parametrizada

<<at-least-one-of feature group>>	A característica deve possuir pelo menos uma das características opcionais que a compõe.
<<zero-or-one-of feature group>>	A característica deve possuir uma ou nenhuma característica opcional que a compõe.
<<exactly-one-of feature group>>	A característica deve possuir exatamente uma característica opcional que a compõe.

A Figura 2.2 apresenta três exemplos de diagrama de características. O diagrama **a** apresenta a descrição de uma característica de "reservas de hotéis". Essa característica pode ser formada por qualquer combinação das características "Reserva Simples", "Bloquear Reserva de Turistas" e "Bloquear Reserva de Conferencias", desde que pelo menos uma delas seja usada. A característica "Reserva Simples" é definida como padrão.

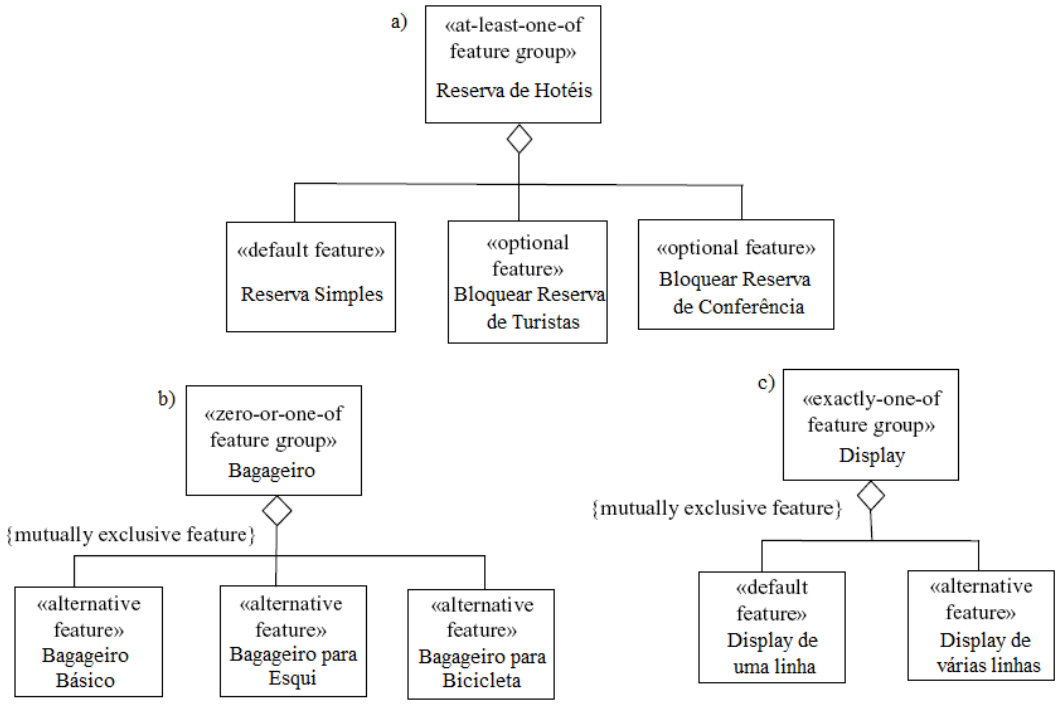


Figura 2.2: Exemplo de diagrama de features apresentado por Gomaa (2004).

Ainda na Figura 2.2, os demais diagramas mostram relações mutuamente exclusivas entre as características. O diagrama **b** apresenta a característica "Bagageiro", que pode ser composta por somente uma das características entre "Bagageiro Simples", "Bagageiro para Esqui" ou "Bagageiro para Bicicleta", ou mesmo por nenhuma delas.

Já no diagrama **c**, tem-se a característica “Display”, que obrigatoriamente deve ser composta por exatamente uma das opções apresentadas, ou seja, “Display de uma linha” ou “Display de várias linhas”.

2.4 Padrões e Linguagens de Padrões

Um padrão de software (*software pattern*) é uma solução para um problema recorrente em um contexto particular no processo de desenvolvimento de software [Gamma *et al.* 1995, Coplien 1996, Schimidt *et al.* 1996, Braga 2002]. Ele tem o objetivo de auxiliar desenvolvedores menos experientes através da comunicação de conhecimento adquirido ao longo do tempo, divulgando soluções de sucesso para certas situações e indicando as conseqüências decorrentes da sua utilização em determinados contextos [Schimidt *et al.* 1996].

Por se tratarem de reúso de experiência, padrões de software não são artefatos “inventados” por desenvolvedores [Kischer and Volter 2007]. A solução apresentada por um padrão é identificada a partir da observação de sistemas já desenvolvidos e que apresentem sucesso em seus objetivos. Logo, a identificação de um padrão depende da experiência dos seus autores com diversos sistemas e contextos diferentes [Schmidt *et al.* 1996, Kischer e Volter 2007]. Além disso, padrões de software podem ser identificados em qualquer ponto do processo de desenvolvimento de sistemas, como análise, documentação, projeto, teste e gerenciamento de configuração [Braga 2002, Kischer e Volter 2007]. Aplicações de padrões podem ser observadas também em áreas como arquiteturas de aplicações empresariais, *messaging* e arquitetura orientada a serviços (SOA) [Zimmermann *et al.* 2008].

Para que um padrão seja utilizado com sucesso, ele deve ser descrito de maneira clara e completa. Autores de padrões não procuram definir métodos formais para apresentá-los, nem ferramentas que apóiem a sua utilização. O foco da criação de um padrão está na descrição dos pontos chave para a apresentação da solução proposta [Schimidt *et al.* 1996], como o ambiente em que o padrão deve ser empregado e as conseqüências de sua utilização. Logo, os principais elementos da descrição de um padrão são: Nome, Problema, Solução e Conseqüências de uso. Ainda assim, vários autores sugerem modelos para se descrever um padrão. Entre os modelos mais comuns de representação estão os de Alexanderian, GoF [Gamma *et al.* 1995], Portland,

Appleton e Coplien [Coplien 1996]. O modelo de GoF para padrões de projeto em sistemas orientados a objeto, por exemplo, é composto por:

- **Nome e classificação**: descreve sucintamente (em duas palavras no máximo) a essência do padrão;
- **Intenção**: uma frase curta que apresenta o que o padrão faz e qual problema particular ele atende;
- **Também conhecido como**: outros nomes conhecidos para o padrão, caso haja;
- **Motivação**: um cenário que ilustra o problema e a maneira como o padrão resolve o problema;
- **Aplicabilidade**: quais as situações em que os padrões podem ser utilizados e como é possível reconhecer essas situações.
- **Estrutura**: representação gráfica das classes que compõem o padrão, ilustrando as relações dos objetos.
- **Participantes**: as classes e objetos existentes no padrão e suas respectivas responsabilidades.
- **Colaborações**: a maneira como os participantes se relacionam para realizarem suas responsabilidades.
- **Conseqüências**: quais mudanças a aplicação do padrão acarreta ao ambiente em que ele vai ser empregado e quais pontos desse ambiente não sofrem influência do padrão.
- **Implementação**: quais técnicas são mais viáveis para a implementação do padrão.
- **Amostra de código**: exemplo de código de como o padrão pode ser implementado em uma linguagem orientada a objetos.
- **Usos conhecidos**: exemplos do uso da solução em sistemas reais.
- **Padrões relacionados**: quais outros padrões de projeto estão relacionados com esse padrão, e quais as diferenças entre eles.

Como sugerido no modelo de GoF, padrões usam notação gráfica, geralmente na representação de diagramas de classes ou de seqüência, para ilustrar a sua aplicação, assim como podem apresentar trechos de código para exemplificação em uma linguagem qualquer [Schimidt *et al.* 1996]. Porém, é importante atentar para o fato de

que a representação gráfica por si só não apresenta a solução descrita no padrão. A descrição do ambiente a ser aplicado e das conseqüências advindas do uso do padrão são os pontos mais importantes a serem observados no seu uso [Kischer e Volter 2007].

Em sua maioria, os padrões são desenvolvidos para serem utilizados individualmente dentro de seu contexto. Mas, existe também a possibilidade de que haja relacionamento entre diferentes soluções buscando a melhoria dos resultados de sua utilização. Existem três formas muito comuns de relacionamentos entre padrões [Buschmann *et al.* 2007]: complementos, padrões compostos e seqüências de padrões.

Complemento entre padrões (*Pattern complements*) ocorre quando um padrão provê a outro alguma nova característica ou mesmo uma solução alternativa para um problema. Em geral complementos cooperativos resultam em padrões mais completos e balanceados. Por exemplo, o padrão *Disposal Method*, que endereça a destruição de objetos, complementa o *Factory Method*, que define a criação desses mesmos objetos, complementando a funcionalidade do padrão anterior [Gamma *et al.* 1995].

Padrões Compostos (*Pattern Compounds*) ocorre quando um grupo de padrões é aplicado da mesma maneira em um mesmo problema recorrente, podendo ser enxergado como um único padrão. Por exemplo, o padrão de projeto MVC que divide a arquitetura de uma aplicação em três camadas (*Model*, *View* e *Control*) é a composição dos padrões de projeto *Observer*, *Strategy* e *Composite* [Gamma *et al.* 1995].

Seqüência de Padrões (*Pattern Sequences*) compreende a utilização progressiva de padrões seguindo uma seqüência lógica. Uma seqüência de padrões descreve um meio de aplicar um padrão em um contexto de modo que ele gere condições para a aplicação de outro padrão pré-definido em seguida.

Existem muitos exemplos de padrões relacionados desenvolvidos pela comunidade de engenharia de software. Um deles, o padrão composto Persistence Layer, será comentado por ter grande relevância nesse trabalho.

2.4.1 Padrão composto *Persistence Layer*

O padrão composto *Persistence Layer* [Yoder *et al.* 1998] é um padrão de projeto que apresenta uma solução para o isolamento do tratamento de persistência em uma camada

a parte do sistema. Dessa forma, mudanças no gerenciamento da persistência de um sistema não geram alterações na estrutura do software.

Esse padrão é formado pela colaboração de outros nove padrões que tratam cada um de uma parte específica da persistência de dados. São eles [Yoder *et al.* 1998]:

- *CRUD* - oferece as operações de criação (*create*), leitura (*read*), atualização (*update*) e deleção (*delete*) de registros em uma tabela. Essas operações implementam o comportamento mínimo para o tratamento de um objeto da aplicação que deva ser persistido.
- *SQL Code Description* – encapsula e armazena a sintaxe de código para a execução das operações do *CRUD*.
- *Attribute Mapping Methods* – oferece métodos que atribuem as informações do banco de dados aos atributos do objeto persistente, e também que recuperam essas informações para o preenchimento de registros da tabela correspondente ao objeto.
- *Type Conversion* – faz o mapeamento de tipos de informação que são diferentes no banco de dados em relação às classes, como, por exemplo, valores nulos e campos de data.
- *Change Manager* – faz o controle de gravação dos valores de um objeto somente se houve modificação após a sua leitura, diminuindo o número de acessos ao banco de dados.
- *OID Manager* – garante a geração de chaves primárias
- *Transaction Manager* – gerenciador de transações com o banco de dados. Garante que seqüências de operações que são dependentes entre si só sejam gravadas quando toda a seqüência tiver apresentado sucesso.
- *Connection Manager* – Encapsula o gerenciamento de abertura e fechamento de conexões com o banco de dados.
- *Table Manager* – Descreve o mapeamento entre os nomes de objetos e atributos com os nomes das respectivas tabelas e colunas, mantendo essas informações isoladas do desenvolvedor.

Na Figura 2.3 pode-se observar como esses padrões se relacionam entre si, apresentando em conjunto as soluções para a realização da persistência de maneira

isolada. Pode-se observar que *Persistence Layer* provê a interface para as operações de *CRUD* e também constrói as chamadas ao banco de dados usando *SQL Code Description*. Durante a geração do código SQL, *Persistence Layer* interage com *Table Manager* para obter os nomes corretos das tabelas e das colunas presentes no banco de dados. A relação entre os campos da tabela e os atributos do objeto persistente do sistema é feito por *Attribute Mapping Methods*, que usa *Type Convertion* para fazer as correspondências entre os tipos de dados diferentes. *Persistence Layer* realiza a gravação dos dados usando conexões abertas por *Connection Manager*, desde que haja mudanças no objeto. Essas mudanças são gerenciadas por *Change Manager*. *Connection Manager* também pode interagir com *Table Manager* para decidir qual banco de dados usar. E quando o processo de gerenciamento de transações é necessário, *Persistence Layer* provê acesso a *Transaction Manager* para que este gerencie esse requisito.

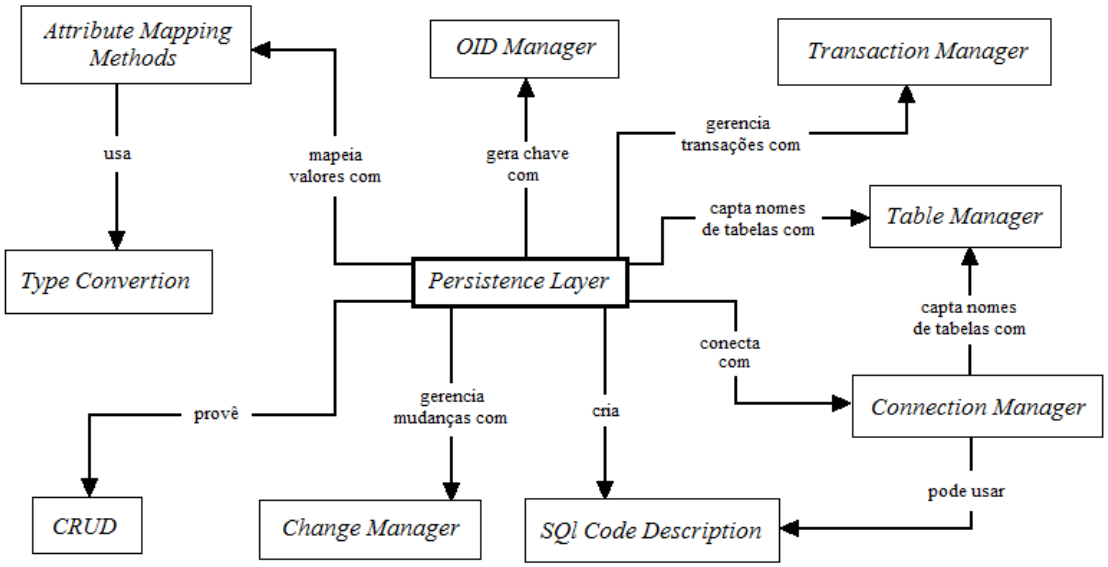


Figura 2.3: Relação dos padrões de Persistence Layer [adaptado de Yoder *et al*., 1998].

2.4.2 Linguagens de padrões

Além de colaborarem entre si na forma de complementos, composições ou seqüências, padrões podem também ser estruturados em coleções que se apóiam para abranger todos os requisitos importantes de um domínio específico, formando uma estrutura chamada de Linguagem de Padrões [Coplien 1996]

Uma estrutura montada como uma linguagem de padrões deve possuir pelo menos um padrão atendendo aos requisitos de cada uma das partes de um sistema,

cobrindo todas as partes importantes do domínio que ela se propõe a auxiliar [Braga 2002, Bushmann 2007]. Assim, uma Linguagem de Padrões age como um guia no processo de desenvolvimento, representando a seqüência de decisões que conduzem ao projeto completo de uma aplicação [Brugali *et al.* 1997].

A representação de uma linguagem de padrões deve conter, além dos padrões em si, uma visão geral sobre o domínio e uma visão geral sobre as interações entre os padrões. O formato de Appleton é um dos mais usados para a representação de linguagens de padrões [Braga 2002].

2.4.2.1 Exemplo de linguagem de padrões – GRN

Um exemplo de uma linguagem de padrões é a GRN (Gestão de Recursos de Negócios) [Braga 2002]. A criação dessa linguagem de padrões foi motivada pela observação de diversas similaridades entre sistemas diferentes desenvolvidos dentro do domínio de Gestão de Recursos. É formada por quinze padrões de análise interdependentes, que podem ser utilizados por analistas inexperientes para desenvolverem sistemas específicos dentro desse domínio [Braga 2002]. Mais especificamente, a GRN auxilia na análise de sistemas que necessitem do gerenciamento de transações de comercialização, aluguel ou manutenção de recursos.

A Figura 2.4 ilustra os padrões que compõem a linguagem, bem como a seqüência que deve ser empregada na sua instanciação. Neste diagrama, os padrões são divididos em três grupos de acordo com seu propósito [Braga 2002].

O primeiro grupo apresenta os padrões responsáveis pela identificação e qualificação dos Recursos, quando houver necessidade. O segundo grupo, e também o mais importante, trata das transações efetuadas pelo sistema. Por fim, o terceiro grupo apresenta os padrões ligados ao detalhamento das transações. Os principais padrões da linguagem, que na Figura 2.4 aparecem reforçados com uma linha mais escura, são Locar o Recurso, Comercializar o Recurso e Manter o Recurso. A utilização de pelo menos um desses padrões é obrigatória na aplicação da linguagem, mas estes não são mutuamente exclusivos.

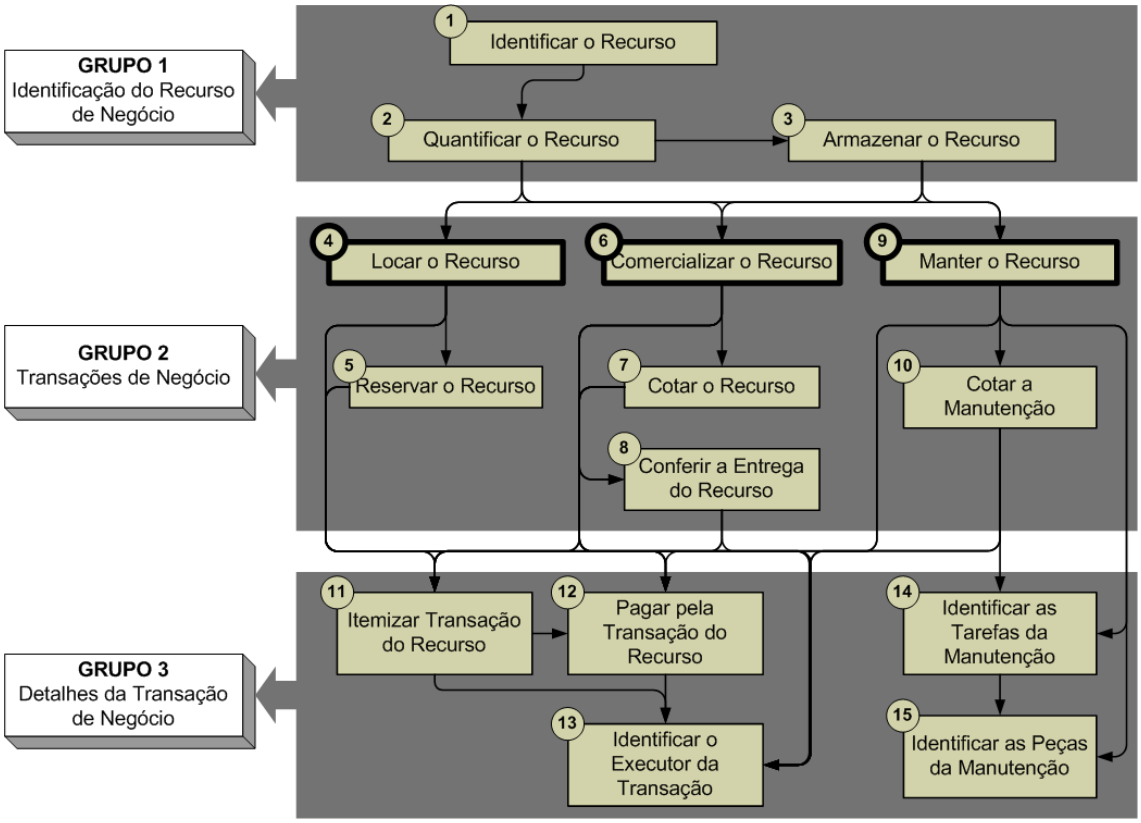


Figura 2.4: Estrutura da Linguagem de Padrões GRN (Braga, 2002).

2.5 Frameworks Orientados a Objetos

Frameworks apresentam-se como uma das tecnologias mais bem aceitas e utilizadas no desenvolvimento de software apoiado por reúso, sendo usado intensivamente no desenvolvimento de sistemas de software complexos e adaptativos [Itakosum 2008]. A estrutura de um framework se baseia em conceitos da orientação a objetos para proporcionar o reúso de software em grande escala, atendendo a domínios de aplicações específicos [Fayad e Schimidt 1997], promovendo assim reúso da análise e projeto de um software na construção de um novo sistema, ao invés de simplesmente o reúso de código [Johnson 1997].

Frameworks podem ser definidos de duas maneiras [Johnson 1997]:

- quanto ao seu propósito, um framework é um esqueleto de uma aplicação semi-completa que atende às especificações de um determinado domínio. Esse esqueleto deve ser customizado para atender aos requisitos específicos do sistema que se pretende desenvolver.

- quanto à sua estrutura, são projetos reutilizáveis de todo ou de parte de um sistema que é representado por um conjunto de classes abstratas e de meios em que suas instâncias interagem. Essas classes representam a arquitetura de um sistema orientado a objetos.

No conjunto de classes presentes na estrutura de um framework, cada classe pode possuir um papel específico, podendo ser classificada como pontos fixos (*frozen spots*) e pontos variáveis (*hot spots*). Os pontos fixos são formados por classes concretas e representam a funcionalidade do software já desenvolvida, testada e pronta para ser reusada. Esses pontos definem o comportamento comum do domínio representado pelo framework [Braga 2002]. Os pontos variáveis (*hot spots*) representam os elementos de software flexíveis que permitem ajustes para que a instanciação do framework atenda devidamente às especificações de um sistema específico [Parsons *et al.* 1999]. Esses pontos são formados por classes abstratas e por métodos abstratos, conhecidos como métodos "gancho" (*hook methods*), que devem ser concretizados a partir dos requisitos do software que se está desenvolvendo.

A organização das classes de um framework pode ser realizada de diversas maneiras com o objetivo de atender a diversos domínios diferentes. Assim, frameworks podem ser classificados quanto ao seu escopo de atuação e quanto à forma de reúso empregada na sua utilização.

Quanto ao escopo, um framework pode ser classificado como [Fayad e Schimidt 1997]:

- Frameworks de infra-estrutura de sistemas (*system infrastructure*): simplificam o desenvolvimento de infra-estruturas de sistemas, como frameworks de comunicação, ferramentas para interfaces de usuário e ferramentas para processamento de linguagem. Geralmente são construídos e usados dentro de organizações, e dificilmente são comercializados.
- Frameworks de middleware de integração (*middleware integration*): Usado para integrar aplicações e componentes distribuídos. Visa melhorar a modularização e reúso da infra-estrutura de um software em ambiente distribuído.
- Frameworks de aplicações empresariais (*enterprise application*): tratam de amplos domínios e geralmente são o alicerce das empresas. O desenvolvimento

ou aquisição de frameworks desse tipo traz altos custos, mas o retorno do investimento é rápido, dado o fato de que ele auxilia o desenvolvimento de software diretamente.

Quanto ao processo de reúso, frameworks podem ser diferenciados quanto à maneira como as classes da sua estrutura são utilizadas na construção de sistemas específicos. Nesse sentido frameworks podem ser classificados como frameworks caixa-branca e frameworks caixa-preta [Fayad e Schimidt 1997, Parsons *et al.* 1999].

Frameworks caixa-branca são compostos por classes abstratas e a instanciação de sistemas é feita por meio de herança e da implementação ou sobrescrição (*override*) de métodos "gancho" pré-definidos nesse framework [Fayad e Schimidt 1997]. Este tipo de estruturação exige que o desenvolvedor tenha acesso ao código-fonte do framework e conheça profundamente a sua estrutura interna para realizar o processo de instanciação.

Já os frameworks caixa-preta funcionam de maneira similar a componentes, tendo interfaces bem definidas para a composição de suas classes. Dessa maneira, a funcionalidade existente no framework é instanciada na construção de componentes que se acoplam a ele. Esses frameworks demandam muito menos esforço para sua instanciação, pois não exigem do desenvolvedor nenhum conhecimento de sua estrutura interna. Porém, são utilizados em um número menor de aplicações por possuírem menor estensibilidade. Além disso, frameworks caixa-preta são mais difíceis de serem desenvolvidos, uma vez que devem ser exaustivamente testados e exigem a existência de métodos e comportamentos que antecipem todos os casos de uso dos sistemas em que serão empregados [Fayad e Schimidt 1997, Parsons *et al.* 1999].

Embora haja essa diferença de relação entre as classes de frameworks, nem sempre um framework é puramente caixa-branca ou caixa-preta. Alguns pontos variáveis de um framework podem ser planejados como caixa-branca, mas podem ser implementados como caixa-preta. A evolução de um framework caixa-branca também pode alterar a maneira como ele é implementado, tornando-o um framework caixa-preta [Fayad e Schimidt 1997]. Existem também frameworks que possuem tanto mecanismos caixa-branca quanto mecanismos caixa-preta em sua estrutura, o que alguns autores chamam de frameworks caixa-cinza [Parsons *et al.* 1999].

A tecnologia de frameworks é recorrente no desenvolvimento de softwares complexos, pois possui vantagens sobre outras tecnologias comuns de reúso, além de permitir a adaptação e implementação de parte de seu código, o que amplia a sua aplicabilidade [Johnson 1997, Vanhaute *et al.* 2001]. Basicamente pode-se apresentar três vantagens principais na utilização de um framework como apoio ao desenvolvimento de software [Chou *et al.* 2007]:

- Modularidade: Frameworks de aplicação permitem uma divisão modular adequada de sua estrutura, encapsulando detalhes da sua implementação em interfaces. A modularidade pode ajudar a melhorar a qualidade de software isolando mudanças de projeto e implementação. Esse isolamento reduz o esforço no entendimento de partes irrelevantes no desenvolvimento de novas aplicações.
- Reusabilidade: As classes abstratas providas por frameworks melhoram a sua reusabilidade definindo vários mecanismos comuns que podem ser usados para construir novas aplicações. A reusabilidade dos frameworks utiliza o conhecimento do domínio e esforço prévio de desenvolvedores experientes para resolver dificuldades de projeto. Pode melhorar a produtividade de código, bem como melhorar a qualidade, a performance e a confiabilidade do software desenvolvido.
- Escalabilidade: O framework melhora a escalabilidade provendo métodos de "*callback*" explícitos para desacoplar completamente as interfaces e o comportamento do domínio da aplicação das variações requeridas para a instanciação de aplicações em um contexto particular. Isso permite o desenvolvimento de softwares mais extensíveis.

Fazendo uso dessas vantagens, o desenvolvimento de frameworks para o apoio à construção de softwares pertencentes a um mesmo domínio é muito comum. Atualmente existem frameworks desenvolvidos para os mais diversos domínios, como redes de sensores [Chou *et al*. 2008], softwares para ambientes RFID [Chung *et al*. 2008, Yui *et al*. 2008] ou aplicações web [Li e Qiang 2008]. Outros exemplos de frameworks de aplicação bastante recorrentes na literatura são: Negócio (IBM San Francisco), frameworks de manufatura (CIm, OSEFA, PRM) e de sistemas distribuídos (COM/OLE, DSOM, OpenDoc, ORB, etc.) [Mat Jani e Lee 2008].

Um característica importante com relação aos Frameworks é que suas estruturas possuem grande relação com o conceito de padrões, sendo que estas duas técnicas de reúso não são mutuamente exclusivas em um projeto. Ao contrário, elas podem muitas vezes se apoiar para promover o reúso de maneira mais adequada em certos contextos. Os padrões apresentados por Gamma *et al.* (1995), por exemplo, foram descobertos a partir do exame de diversos frameworks e foram escolhidos pela sua representatividade como sistemas de software orientados a objetos reutilizáveis [Johnson 1997]. Assim, um único framework pode conter diversos padrões de projeto implementados em sua estrutura [Johnson 1997, Demeyer *et al.* 2000]. Além disso existem frameworks amplamente utilizados no mercado criados como uma implementação do padrão de projeto MVC (*Model/View/Controller*)[Glenn e Pope 1998, Sacowicz 2007, Cao *et al*. 2008, Liao e Koonse 2007].

Ainda que frameworks promovam o reúso de projeto de software com sucesso, alguns autores apresentam problemas e desvantagens na sua utilização. Um dos problemas na utilização de um framework é a obrigatoriedade do conhecimento da sua estrutura interna, por se tratar de uma estrutura complexa e de difícil aprendizado. Sua utilização exige a presença de desenvolvedores experientes no processo de instanciação do framework em questão [Lopes *et al.* 2007]. Outra dificuldade refere-se ao fato de sua aplicação ser dependente da linguagem na qual foi desenvolvido, o que restringe o reúso dessa estrutura [Johnson 1997], sendo possível construir apenas softwares específicos para essa linguagem. A integração entre frameworks também apresenta problemas, uma vez que os frameworks assumem o controle principal da aplicação que estão instanciando. Dessa maneira, é difícil combinar dois ou mais frameworks que possam assumir esses papéis e a manter a integridade de suas estruturas. Outra desvantagem em relação a essas estruturas está na construção de frameworks que, mesmo com as várias técnicas de desenvolvimento de software atuais, ainda é um processo custoso, dada a complexidade de sua estrutura [Intakosum 2008].

2.6 Frameworks Orientados a Aspectos

A instanciação de sistemas de software a partir de frameworks favorece o reúso e possibilita assim maior qualidade e rapidez no desenvolvimento. Porém a sua complexidade pode gerar problemas quanto a sua compreensão e manutenibilidade, pois muitas vezes essas estruturas apresentam interesses transversais espalhados pelo seu

código. Alguns estudos mostram que a combinação dessa técnica com a orientação a aspectos pode possibilitar o isolamento da implementação de interesses transversais, aumentando ainda mais o reúso de aplicações proporcionados por ela [Vanhaute *et al.* 2001] além de trazer melhorias na manutenção de suas estruturas. A utilização de aspectos na estrutura de frameworks pode ser realizada tanto para o tratamento de alguns interesses da arquitetura de frameworks Orientados a Objetos quanto para a construção de frameworks para o tratamento de somente um interesse transversal [Camargo e Masiero 2005].

Para generalizar aspectos na estrutura de um framework, tornando-os utilizáveis na construção de diversas aplicações específicas, é importante identificar os pontos variáveis de seu domínio. Assim, devem ser definidos aspectos abstratos que serão implementados posteriormente considerando as especificações do sistema que se quer desenvolver, ou mesmo do interesse transversal que se pretende tratar [Vanhaute *et al.* 2001]. Muitos autores têm conseguido essa generalização com sucesso. O primeiro framework orientado a aspectos foi proposto por Constantinides (2000), para o desenvolvimento de um sistema para o controle de concorrência em sistemas distribuídos. Embora não seja implementado em uma linguagem orientada a aspectos, utiliza em sua estrutura conceitos de POA para o tratamento de objetos mutuamente concorrentes. Vanhaute *et al.* (2001) desenvolveram, na linguagem AspectJ, aspectos para o tratamento de controle de acesso e de confidencialidade e propõe uma maneira de implementá-los de forma genérica nos moldes de um framework. Para os dois casos, aspectos abstratos são definidos, em que os métodos que envolvem os elementos dependentes da segurança devem ser informados pelo desenvolvedor que instanciar o framework. Díaz *et. al.* (2005) também desenvolveu um framework de aplicação orientado a aspectos, para controlar a construção de objetos científicos relacionados à Computação de alto desempenho. Dessa maneira, o autor pretende atingir dois objetivos: melhoria quanto à reusabilidade e à simplicidade do código e melhoria do tempo de execução da aplicação. Um framework orientado a aspectos, assim como frameworks OO, pode ser definido de duas maneiras [Camargo e Masiero 2005]:

- quanto a sua estrutura, um framework orientado a aspectos é um conjunto formado por aspectos abstratos e também, mas não obrigatoriamente, por classes implementadas segundo a orientação a objetos, formando projetos

reutilizáveis de todo um sistema ou do tratamento de um interesse transversal específico.

- Já quanto ao seu propósito, um framework OA é uma aplicação semi-completa que pode ser customizada para o desenvolvimento de aplicações de um domínio e garante também que os interesses transversais fiquem isolados dos interesses base do sistema.

Muitos frameworks orientados a objetos com aspectos em sua composição tem sido desenvolvidos utilizando o conceito de frameworks Orientados a Aspectos para denominá-los, mas sem determinar uma definição clara para suas estruturas. Baseado nesse fato, Camargo e Masiero (2005) propõe a classificação de frameworks Orientados a Aspectos em dois grupos: Frameworks de Aplicação Orientados a Aspectos (FAOA) e Frameworks Transversais (FT).

Frameworks de Aplicação Orientados a Aspectos (FAOA) representam o conjunto de sistemas que tem como finalidade a instanciação de aplicações garantindo o reúso das informações contidas no framework e que são constituídos tanto de classes quanto de aspectos. Dessa forma, a sua estrutura se assemelha muito à de um framework orientado a objetos, tendo como diferencial a presença de aspectos no tratamento de alguns de seus interesses.

Já Frameworks Transversais (FT) são as estruturas que se responsabilizam exclusivamente pelos interesses transversais de um sistema, garantindo o seu isolamento e a fácil reutilização dessa estrutura em qualquer sistema que necessite desse tratamento. Essa classe de frameworks não gera um produto por si só. Ele deve sempre ser acoplado a outro sistema que contenha os interesses que o framework se propõe a tratar. Frameworks transversais também devem possuir obrigatoriamente maneiras de realizar a composição que possam ser adaptadas pelo desenvolvedor, garantindo assim o acoplamento com qualquer código-base. A composição de FTs a um sistema se caracteriza por duas atividades: a primeira, referente à instanciação, é identificar nesse sistema os pontos de junção adequados para o acoplamento do framework e a segunda, de composição, refere-se a determinar as regras para unir todas as variabilidades escolhidas na realização desse acoplamento.

2.7 Manutenção de software e Frameworks

Manutenção de Software corresponde a qualquer modificação realizada na estrutura de um sistema, seja ela durante ou após o desenvolvimento do software, e está intrinsecamente ligada ao desenvolvimento de qualquer sistema informatizado [Pressmann 2006]. O processo de manutenção é comumente caracterizada por alto custo e velocidade lenta de implementação. Porém é um processo inevitável do ciclo de vida de qualquer software visto que a boa qualidade de um software é alcançada através de mudanças e melhorias aplicadas à sua concepção inicial [Bennett e Rajlich 2000]. A manutenção de um software pode ser realizada em todas as fases do seu ciclo de vida, incluindo a fase de projeto, e de documentação [Park 2007].

As modificações realizadas em softwares podem ter diversas razões. Assim, elas são divididas em quatro categorias principais [Bennett e Rajlich 2000, Pressmann 2006, Park 2007]:

- Manutenção Perfectiva: inserção de novas funções, atendimento de novos requisitos, ou melhorias diversas no sistema.
- Manutenção Adaptativa: modificação realizada para refletir uma mudança no ambiente do qual o sistema faz parte.
- Manutenção Corretiva: resolução de problemas na execução de uma função qualquer.
- Manutenção Preventiva: detectar e corrigir problemas em potencial.

Com relação à manutenção de frameworks, diversas dificuldades podem ser encontradas na adaptação ou evolução de suas estruturas em conseqüência de sua complexidade e à dificuldade em se capturar toda a teoria do domínio ao qual ele atende [Cortes *et al.* 2003]. Logo, estudos têm sido feitos para se desenvolver e aprimorar técnicas que facilitem a manutenção de frameworks de aplicação.

Cortes *et al*. (2003), por exemplo, utilizam regras de refatoração e unificação para auxiliar o processo de manutenção e evolução de frameworks de aplicação. Os seus resultados apresentam melhorias na realização de manutenções corretivas nos frameworks testados. Dagenais e Robillard (2008) apresentam uma ferramenta que auxilia a evolução de aplicações geradas a partir de frameworks. A ferramenta

desenvolvida sugere adaptações nesses programas segundo a análise de um histórico de alterações realizadas no próprio framework.

Uma outra maneira de se melhorar a manutenibilidade de frameworks é com o uso da orientação a aspectos. Lobato *et al*. (2008) apresentam um estudo de caso sistemático onde foi comparada a evolução de versões Orientada a Objetos (OO) e Orientada a Aspectos (OA) de um mesmo framework de aplicação. Neste trabalho, foram realizadas manutenções evolucionárias como inclusão de novas funções, e composição com outros frameworks. Como resultado desse trabalho, observou-se que a modularização da orientação a aspectos auxiliou a evolução desse framework, sendo que a versão que possui interesses isolados pelo uso desse paradigma exigiu menos esforço durante as manutenções do que a versão orientada a objetos.

2.8 Considerações Finais

O reúso de software é uma prática de desenvolvimento de software que garante qualidade de desenvolvimento. Porém esta prática depende de artefatos construídos com o uso de boas práticas do desenvolvimento de software para que o reúso ocorra de maneira satisfatória.

A técnica de framework, assim como outras formas de reúso de software, dependem da construção de um código flexível e modularizado para facilitarem o desenvolvimento de software. Caso contrário, a dificuldade de compreensão e utilização do framework pode elevar o custo da instanciação de uma aplicação em relação ao seu desenvolvimento sem o auxílio do framework.

A orientação a aspectos, em especial, garante essa modularização de maneira efetiva, podendo ser aplicada com sucesso no desenvolvimento de outros artefatos de reúso. Padrões de projeto de software implementados com a orientação a aspectos podem ser utilizados no desenvolvimento de software sem a ocorrência de código invasivo na estrutura do sistema. Dessa maneira, um padrão pode ser substituído ou sofrer manutenção, sem que isso cause modificações no código do sistema ao qual ele foi aplicado. Frameworks também podem se valer da orientação a aspectos dentro de suas estruturas com o objetivo de modularizar seus interesses, facilitando a sua compreensão.

A flexibilização de modularização de um software pode também auxiliar a sua manutenção, possibilitando a aplicação de práticas mais recentes de reúso e de modularização de software a sistemas que não as utilizam em sua concepção inicial. Dessa forma pode-se, por exemplo, aplicar o conceito de orientação a aspectos a sistemas totalmente desenvolvidos com a orientação a objetos, visando resolver problemas de modularização.

O capítulo a seguir apresenta o GRENJ, framework orientado a objetos gerado a partir de um processo de reengenharia e que pode ser aperfeiçoado quanto ao tratamento de persistência de dados. Uma família de frameworks transversais de persistência que pode ser usada juntamente com o GRENJ é também apresentada.

CAPÍTULO 3
GRENJ e FT de Persistência

3.1 Considerações Iniciais

Frameworks têm sido desenvolvidos dentro dos mais diversos domínios e, muitas vezes, são construídos com base em padrões de software, aproveitando-se da experiência contida nesses padrões. Uma das principais dificuldades no desenvolvimento e também na utilização de um framework é quanto à sua complexidade, o que pode gerar, entre outros problemas, entrecorte ou espalhamento dos interesses presentes na sua estrutura. Técnicas de desenvolvimento Orientado a Aspectos, como frameworks transversais, se apresentam como uma técnica de apoio ao desenvolvimento de frameworks realizando a separação desses interesses, para torná-los mais legíveis e manuteníveis.

Este capítulo apresenta, em detalhes, os dois frameworks que são aqui utilizados: O framework de aplicação GRENJ que instancia sistemas na linguagem Java, resultante de um processo de reengenharia, que possui interesses de persistência entrecortando sua estrutura; E um framework transversal de persistência que auxilia na retirada do entrelaçamento de código de persistência em aplicações diversas. Este capítulo está organizado da seguinte maneira: Na Seção 3.2 é apresentado o framework de aplicação GRENJ, bem como detalhes da sua instanciação e dos interesses de persistência que entrecortam sua camada de modelo. Na Seção 3.3, o framework transversal de

persistência utilizado é tratado. Na Seção 3.4 são apresentadas as considerações finais do capítulo.

3.2 Framework de Aplicação GRENJ

Como visto no capítulo anterior, frameworks são conjuntos de classes, concretas e abstratas, que representam características de um determinado domínio, e que devem ser estendidas para a instanciação de uma aplicação específica pertencente a esse domínio [Johnson, 1997]. Esse tipo de estrutura possui relação com padrões de projeto, sendo que muitos frameworks são desenvolvidos com base em padrões existentes, sejam de projeto, de análise, ou mesmo de implementação.

Um exemplo de framework de aplicação caixa-branca construído a partir de uma linguagem de padrões é o framework para Gestão de Recursos de Negócio (GREN) [Braga 2002]. O GREN foi desenvolvido com o objetivo de auxiliar a instanciação de sistemas para o domínio de gestão de recursos de negócio, ou seja, sistemas que tratam de aluguel, de venda e manutenção de bens. Sua construção foi feita com base na linguagem de padrões de análise GRN (Gestão de Recursos de Negócios) [Braga 2002], apresentada na Seção 2.4.2.1. Com o objetivo de promover o encapsulamento dos interesses presentes no framework, sua arquitetura foi projetada seguindo o modelo de três camadas, como mostrado na Figura 3.1. Esse modelo define:

1. A camada de persistência, que contém as classes responsáveis pela conexão com o banco de dados, bem como o gerenciamento da persistência dos objetos do sistema. Essa camada se relaciona diretamente com os recursos básicos de banco de dados;
2. A camada de regras de negócio, que implementa os padrões da GRN e os seus respectivos relacionamentos, contendo várias classes derivadas diretamente dessa estrutura. Essa camada possui relacionamento com as classes de persistência para realizar a gravação ou recuperação de informações referentes aos padrões utilizados.
3. A camada de interface gráfica, que possui classes responsáveis pelo controle da comunicação do usuário final do sistema com as funções oferecidas pelo mesmo.

A utilização do Wizard ou da camada de interface gráfica são opcionais, havendo a possibilidade do engenheiro de aplicação estender diretamente a camada de negócios do framework na criação de uma nova aplicação e criar sua própria interface gráfica a partir disso.

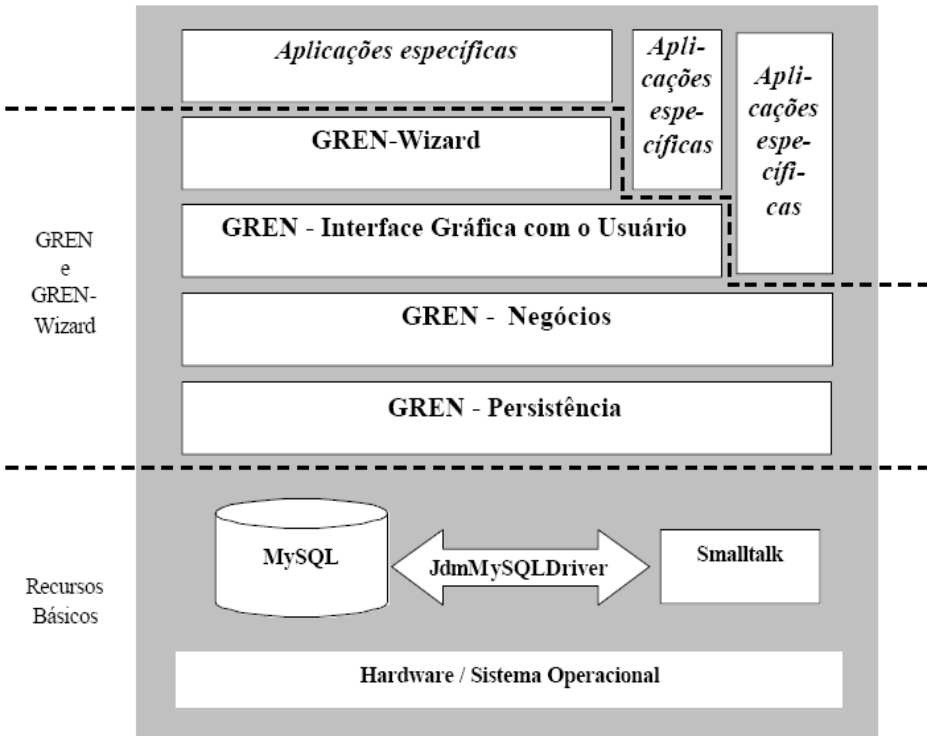


Figura 3.1: Arquitetura do framework GREN.

A implementação do framework GREN foi realizada com o uso da linguagem *Smalltalk* e a base de dados utilizada na persistência foi MySQL[2]. Padrões de projeto foram utilizados na modelagem das classes do framework, como o padrão *Strategy* [Gamma *et al.* 1995], para garantir flexibilidade na definição dos pontos variáveis do framework, e o padrão *Persistence Layer* [Yoder *et al.* 1998] para o isolamento da persistência.

3.2.1 GRENJ

Como já afirmado neste trabalho, a instanciação de sistemas a partir de frameworks é intrinsicamente dependente da linguagem na qual os frameworks foram implementados. Logo, para que um engenheiro de software o utilize, é necessário que conheça a linguagem de programação na qual o framework foi desenvolvido. Pelo fato da

[2] http://www.mysql.com/

linguagem *Smalltalk* não ser amplamente utilizada nas comunidades acadêmica e profissional de engenharia de software havia dificuldade em utilizar e evoluir o framework GREN.

Assim, foi realizada a reengenharia iterativa do framework GREN para a linguagem Java, denominando-o de GRENJ [Durelli, 2008]. Embora existam similaridades entre as linguagens *Smalltalk* e Java, como por exemplo serem interpretadas e seguirem o paradigma de orientação a objetos, a escolha pela linguagem Java ocorreu pelo fato de ser mais utilizada e possuir uma comunidade de desenvolvedores ativa e constantemente preocupada com a sua melhoria. Outra vantagem dessa linguagem sobre *Smalltalk* é que sua sintaxe foi baseada na sintaxe de C e C++, fator o que contribui para o seu aprendizado.

O processo de reengenharia que gerou o GRENJ seguiu uma abordagem iterativa em que cada ciclo da iteração tratou especificamente de um padrão da GRN presente na estrutura do GREN [Durelli 2008]. A reengenharia envolveu as duas primeiras camadas do framework GREN: a camada de persistência e a de regras de negócio. A camada de interface e o wizard que compõem o framework GREN, foram desenvolvidas por Viana (2009).

É importante frizar que o processo de reengenharia foi apoiado pela estratégia de *Test Driven Development* (TDD) [Beck 2001] como guia de implementação. TDD é uma estratégia de programação que visa à escrita de testes automatizados de unidades individuais do programa antes do desenvolvimento do código funcional, em pequenas e rápidas iterações [Janzen e Saiedian 2005]. Embora a palavra “teste” esteja presente no seu nome, TDD não é uma técnica para a aplicação de testes, mas sim uma prática que guia decisões de análise, de projeto e de programação [Beck 2001, Janzen e Saidian 2005, Jeffries e Melnik 2007]. Além dos testes construídos servirem como referência para o desenvolvimento de aplicações, também podem ser usados como referência para refatorações em um código, indicando ao desenvolvedor se mudanças realizadas no código geraram quebra da sua funcionalidade original [Janzen e Saidian 2005, Jeffries e Melnik 2007].

3.2.2 Processo de instanciação do GRENJ

O processo de instanciação do framework obtido com a reengenharia é o mesmo utilizado para a instanciação de aplicações com o uso da GREN. Uma vez que todo framework desenvolvido com base em uma linguagem de padrões pode ser instanciado com o auxílio da própria linguagem [Braga 2002], a GRN pode ser usada como base para a criação de sistemas a partir do framework [Durelli 2008]. O processo de instanciação de um sistema usando o GRENJ é definido pelos seguintes passos:

- análise da aplicação, na qual os requisitos devem ser confrontados com os padrões da GRN para verificar se o sistema se aplica ao domínio que a linguagem de padrões atende;
- classificação dos requisitos, em que cada requisito é classificado e representado por um padrão da GRN;
- implementação das classes de aplicação, em que as classes abstratas correspondentes aos requisitos do sistema devem ser implementadas, bem como os seus *hotspots* e alguns métodos de persistência.

A fase de implementação também pode ser apoiada pela prática de TDD, sendo que em primeiro lugar se desenvolvem os casos de teste para a classe específica que será construída, e então as classes são desenvolvidas até o ponto de se obter o sucesso destes testes.

Quanto aos *hotspots* presentes na estrutura do GRENJ, pode-se destacar três tipos de métodos a serem implementados nas classes de aplicação durante o processo de instanciação.

a) Métodos de informação de classe relacionada: Internamente, as classes da camada de negócios do GRENJ possuem relacionamentos entre si, referenciando-se por meio de atributos do tipo das classes abstratas. O engenheiro de aplicação deve informar ao framework, em tempo de instanciação, qual classe de aplicação representa aquela classe abstrata na aplicação instanciada. Na Figura 3.2 (a), é mostrado um exemplo desse tipo de implementação feito para a classe Filme. Pode-se observar que essa classe estende a funcionalidade da classe *Resource* do GRENJ. Dois métodos que informam ao framework as classes que representam certos papéis dentro do domínio:

`getResourceInstanceClass()` é informado que a classe FitaDeVideo representa a instancia do recurso Filme; `typeClasses()` informa que a classe filme é classificada pela classe Genero.

```
public class Filme extends Resource {

    private String atoresPrincipais;
    private int anoDoFilme;
    private String diretor;
    private Double precoDeLocacao;
    private char dubladoOuLegendado;

    (...)

    //necessary
    @Override
    public Class< ? extends ResourceInstance> getResourceInstanceClass() {
(a)
        return FitaDeVideo.class;
    }

    (...)

    //necessary
    @Override
    public Class[] typeClasses() {
(a)
        return new Class[] { Genero.class };
    }

    //necessary
    @Override
    public String[] typeFieldsInitialize() {
(b)     return new String[] { "genero" };
    }

    (...)

    //necessary
    @Override
    public QuantificationStrategy getQuantificationStrategyInstance() {
(c)
        return new InstantiableResource();
    }

    //necessary
    @Override
    public Class< ? extends MeasureUnity> getMeasureUnityClass()
        return null;
    }
```

Figura 3.2: Trecho de código com instanciação de classe do GRENJ.

b) Métodos de informação de campos referentes às classes relacionadas: a necessidade dessa implementação é similar à das classes relacionadas, mas há necessidade de informar o campo relacionado com a classe no banco de dados. Na Figura 3.2 (b), pode-se observar a implementação do método `typeFieldsInitialize()`com o campo “genero” representa o campo referente à classe Genero na tabela Filme.

c) Métodos de definição de estratégias: Esses métodos devem ser implementados para informar ao framework as estratégias escolhidas para o tratamento de algumas características do domínio. Um exemplo disso também pode ser visto na Figura 3.2 (c). Na utilização do framework GRENJ, é possível escolher dois tipos de estratégia para o tratamento de Recursos: *InstantiableResource*, para recursos instanciáveis, como o caso de locadoras que podem ter DVDs como instancias de Filmes; e *MeasurableResource*, para recursos mensuráveis, como bens de consumo vendidos em um mercado. Podemos ver em (c), na Figura 3.2, que a implementação do método `getQuantificationStrategyInstance()` informa ao framework a estratégia *InstantiableResource*, enviando um objeto da classe correspondente como retorno do método. Logo abaixo, pode-se observar o método `getMeasureUnityClass()` retornando null. Esse método deve retornar uma classe somente se a estratégia utilizada for a de recurso instanciável.

Os métodos de persistência a serem implementados serão melhor detalhados na seção a seguir, juntamente com a descrição da camada de persistência.

3.2.3 Camada de persistência do GRENJ

Seguindo o modelo desenvolvido por Braga (2002) a camada de persistência do GRENJ [Durelli 2008] foi implementada usando o padrão *Persistence Layer* [Yoder *et al.* 1998]. Esse padrão isola o tratamento de persistência dos dados de um sistema em classes que encapsulam esse comportamento [Yoder *et al.* 1998]. A classe `PersistentObject`, definida nesse padrão, contém operações relacionadas à persistência de dados e deve ser estendida por todas as classes da aplicação que serão persistidas. Para simplificar o entendimento deste trabalho, as classes da aplicação instanciada que devem ser persistidas serão chamadas de “classes de aplicação persistentes”.

A Figura 3.3 ilustra a organização de classes da camada de persistência do GRENJ e a sua relação com as classes presentes na camada de modelo do framework. Na camada de persistência, a classe `PersistentObject` funciona como uma interface para a camada de modelo. Assim, todas as classes do framework a serem persistidas devem herdá-la. Para efeito de organização, a Figura 3.3 mostra somente algumas classes da estrutura do framework que herdam diretamente a classe

`PersistentObject`. Pode-se observar também na Figura 3.3 a presença de classes de aplicação `Cliente` e `Filme` que, por extensão das classes do framework, também têm acesso aos métodos de persistência nele implementados.

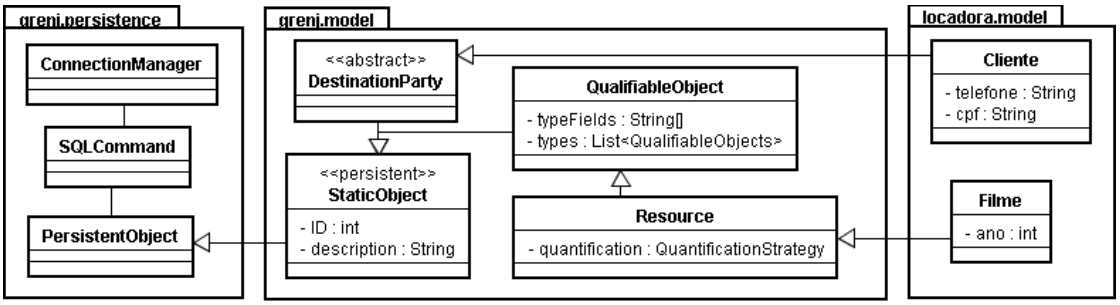


Figura 3.3: Diagrama de classes que mostra a relação da camada de persistência do GRENJ com as classes do modelo.

Além de estender a classe `PersistentObject` as classes que devem ser persistidas, tanto na estrutura do framework quanto na aplicação instanciada, devem implementar alguns métodos para que a persistência seja realizada adequadamente. A implementação desses métodos é feita diretamente nas classes de modelo, tanto do framework quanto da aplicação instanciada e resulta no entrelaçamento do interesse de persistência com a lógica de negócios do GRENJ. Esses métodos podem ser divididos em três grupos:

- Implementação de métodos de informação da estrutura da classe;
- Chamadas de métodos de informação do status da classe;
- Implementação do construtor para carregamento da classe com informações do banco de dados.

Os métodos de informação da estrutura da classe se referem à sobreposição de quatro métodos que servirão de base para a montagem das instruções SQL executadas nas operações de persistência. Esses métodos são:

- `insertionFieldClause()` – cláusula que contém os nomes das colunas para a operação de inserção;
- `insertionValueClause()` – cláusula que contém os nomes dos métodos que retornam os valores a serem inseridos nas respectivas colunas na banco de dados;

- `updateSetClause()` – cláusula que contém os nomes das colunas e os respectivos métodos que retornam valores de cada atributo;
- `updateWhereClause()` – cláusula usada quando um registro é atualizado.

A sobrescrição desses métodos deve ser realizada por todas as classes persistentes que declararem novos atributos, informando à camada de persistência qual a ordem dos campos que deve ser usada nessas instruções.

As chamadas de métodos de informação do status da classe referem-se à chamada dos métodos `setChanged()` e `setPersisted()`. O método `setChanged()` é chamado após qualquer mudança ocorrida em uma classe de aplicação persistente, enquanto o método `setPersisted()` deve ser chamado após a chamada de qualquer método que persista informações do objeto no banco de dados. Esses métodos são utilizados para notificar as alterações dos atributos, de forma que os valores atualizados possam ser posteriormente persistidos. Essa informação será verificada no momento da gravação dos dados, evitando, assim, que o framework execute qualquer gravação de informações sobre classes que não sofreram alterações.

A implementação do construtor refere-se à obrigatoriedade de todas as classes persistentes implementarem um construtor que recebe como parâmetros instâncias das classes `java.sql.ResultSet` e `grenj.util.Index` [Durelli 2008]. Nesta implementação, os valores do parâmetro do tipo `ResultSet` devem ser utilizados para alimentar os valores da classe em questão, e o parâmetro `Index` deve ser incrementado a cada valor passado. Esse construtor não será utilizado pelo engenheiro de aplicação durante a criação de aplicações específicas, sendo somente usado internamente pelo framework no momento da alimentação dos valores dessas classes.

Essas implementações devem ser realizadas tanto pelas classes presentes no framework quanto pelas classes específicas implementadas pelo engenheiro de software no momento da instanciação do GRENJ.

Embora a implementação do padrão *Persistence Layer* tenha por objetivo isolar o interesse de persistência da camada de modelo do GRENJ, as implementações de persistência descritas nessa Seção apresentam espalhamento de código de persistência na estrutura do framework. A Figura 3.4 apresenta um trecho de código da classe de

aplicação Cliente implementada com o uso do framework GRENJ. Pode-se observar em (a) que o método `setEndereco()`, após realizar a mudança do valor do atributo endereço, deve chamar o método `setChanged()`, informando a camada de persistência que houve alteração nas informações do objeto. Em seguida, em (b), pode-se observar a implementação do método `updateSetClause()`, que o engenheiro de aplicação informa a camada de persistência dos campos criados na classe Cliente, e como esses atributos devem ser persistidos. Essa implementação, além de apresentar espalhamento de código de persistência, também exige que o desenvolvedor conheça detalhes da linguagem utilizada nas instruções de interação com o banco de dados, bem como o formato dos conjuntos de caracteres a serem informados. Detalhes desse formato são a presença de uma vírgula no início da instrução ou a seqüência de caracteres especiais "\'" antes e depois da informação do valor a ser persistido na instrução.

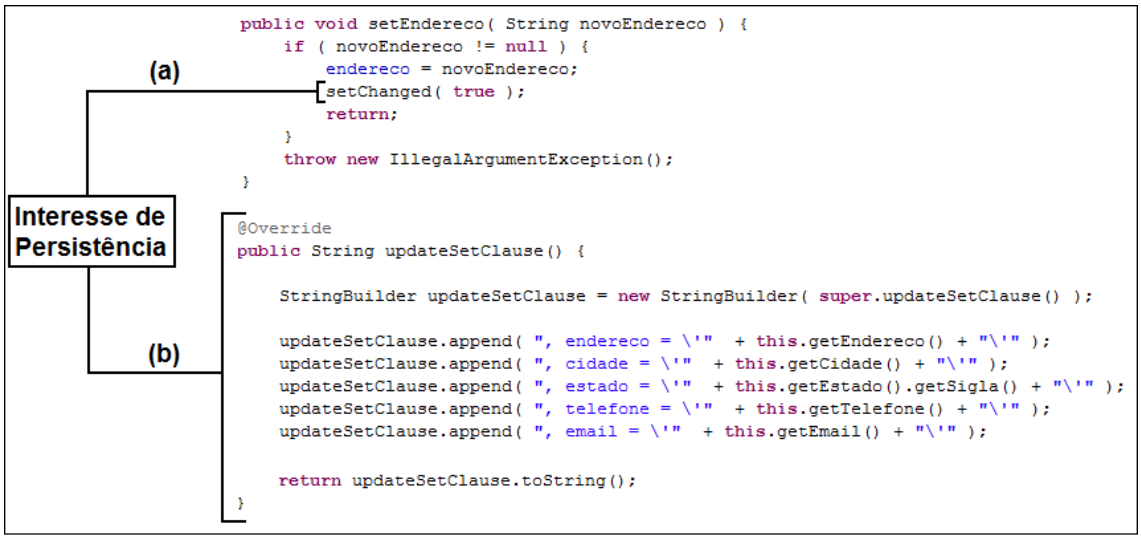


Figura 3.4: Trecho de código de classe de aplicação instanciada a partir do GRENJ, que apresenta espalhamento de código de persistência.

Esse espalhamento de código referente ao interesse de persistência apresentado nesta Seção pode ser isolado caso o mecanismo de persistência utilizado no GRENJ seja alterado. A seguir, será apresentada uma família de frameworks transversais desenvolvida por Camargo e Masiero (2008) que pode realizar esse isolamento adequadamente.

Outra obrigatoriedade do framework quanto à camada de persistência refere-se ao fato de que é obrigatório que o nome da classe persistente seja o mesmo que o nome

da tabela correspondente no banco de dados e que seus atributos tenham os mesmos nomes dos campos dessa tabela. Isso é necessário pois o padrão *Atribute Mapping Methods* do padrão *Persistence Layer* apresentado na Seção 2.4.1 do capítulo anterior não foi implementado para o GRENJ.

3.3 Famílias de frameworks transversais

Como visto no Capítulo anterior, Frameworks Transversais (FT) são estruturas construídas para isolar interesses de persistência específicos em códigos que apresenta entrelaçamento de código com algum interesse transversal. Camargo e Masiero (2008) ilustra três famílias de frameworks transversais, desenvolvidos na linguagem AspectJ, que realizam o tratamento de um interesse transversal em cada uma. Esses interesses são: Persistência, Segurança e Regras de Negócio. Cada família representa uma Linha de Produtos de Software que possui suas características implementadas também como FTs. Essas características devem ser compostas para produzir um framework transversal específico. Além disso, algumas características foram implementadas de modo que pudessem ser reusadas em outros ambientes que não o do FT de persistência propriamente dito. Este trabalho detalhará somente a família de frameworks de persistência utilizada no isolamento da persistência do GRENJ.

3.3.1 Família de frameworks transversais de persistência

Os produtos da família de FTs de Persistência, assim como outros frameworks desenvolvidos para o tratamento desse interesse, procuram facilitar a comunicação entre sistemas orientados a objetos e a estrutura relacional inerente à persistência [Camargo e Masiero 2008]. Além disso, esses produtos procuram manter o interesse de persistência o mais isolado possível da aplicação-base.

Em relação ao isolamento do interesse transversal de persistência, a separação ideal desse requisito seria tornar a persistência totalmente ortogonal ao sistema, ou seja, fazer com que as classes de aplicação persistentes não tivessem nenhuma consciência desse interesse, possibilitando assim que o programador, responsável pelos interesses funcionais do sistema, não tenha qualquer contato com esse tratamento [Al-Mansari *et al.* 2007]. Porém, como estudado por Rashid e Chitchyan (2003), o isolamento total da persistência não é possível de ser alcançado pela linguagem AspectJ, pois não existem pontos bem definidos a serem entrecortados por todas as operações de persistência.

Consulta e exclusão são exemplos de operações que não possuem pontos de junção que possam ser captados pela linguagem para a execução do comportamento transversal, exigindo que essas operações sejam chamadas explicitamente na estrutura do código-base. Outra característica dos produtos da família de FTs de persistência que fere a ortogonalidade desse interesse são as políticas de implementação das classes de aplicação persistentes. Essas políticas devem ser adotadas na modelagem das classes persistentes e do banco de dados para que o FT funcione corretamente. São elas [Camargo e Masiero 2008]:

1. toda classe de aplicação persistente deve possuir construtores com e sem parâmetros;
2. cada classe de aplicação persistente deve ter uma tabela correspondente no banco de dados com o mesmo nome da classe;
3. toda classe de aplicação persistente deve possuir métodos de acesso aos atributos (`sets` e `gets`);
4. cada atributo de uma classe de aplicação persistente deve possuir uma coluna na tabela correspondente no banco de dados com o mesmo nome do atributo.
5. todo método que altera um atributo do tipo `int` deve receber um parâmetro do tipo `Integer`.
6. classes que possuírem relacionamentos de cardinalidade um para um devem ser representados por atributos da classe.
7. classes que possuírem relacionamentos maior que um, devem ser representados por `Vectors` do tipo da classe.

A família de FTs de Persistência é composta por cinco frameworks transversais: Operações Persistentes, Conexão, *Pooling*, Memória auxiliar e Garantia de Políticas.

A Figura 3.5 mostra o diagrama de características da família de FTs de Persistência representado pelo modelo de linhas de produtos de software proposto por Gomaa (2004). Nessa figura, as características apresentadas na cor cinza foram implementadas como Frameworks Transversais ou simplesmente como aspectos na estrutura geral da Linha de Produtos.

Os frameworks Operações Persistentes e Conexão são as únicas características dessa família apresentadas como obrigatórias, sendo que todo FT de persistência gerado

a partir dessa família deve possuir esses dois FTs em sua estrutura. O FT de Operações Persistentes apresenta um conjunto de operações que devem ser herdadas pelas classes de aplicação persistentes para realizar a interação com o banco de dados, operações tais como inclusão, remoção, alteração e consultas. O módulo de Conexão, por sua vez, se preocupa com a definição dos pontos no sistema onde a conexão com o banco de dados deve ser aberta ou fechada.

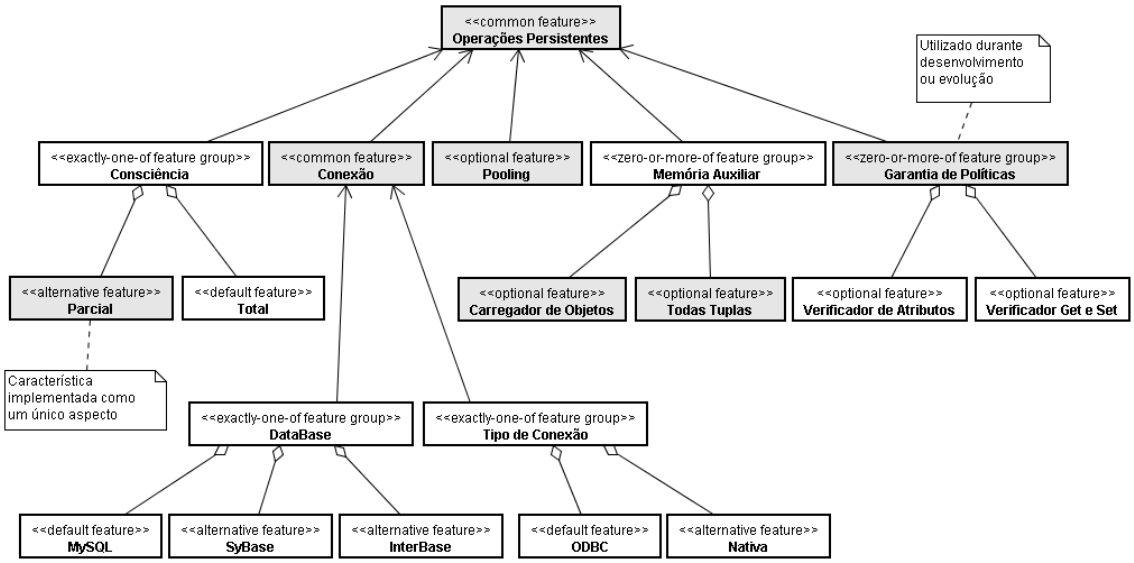


Figura 3.5: Diagrama de características do FT de Persistência [Camargo e Masiero 2008].

A característica Consciência apresenta-se como uma variabilidade do FT Operações Persistentes e define se o código-base terá consciência Total ou parcial da existência do FT. Uma dessas duas variabilidades deve obrigatoriamente estar presentes no FT. Caso a variabilidade escolhida seja a Total, todas as chamadas a operações de persistência devem ser indicadas no código-base pelo engenheiro de software. Já na opção de consciência parcial, o desenvolvedor deve indicar explicitamente somente as chamadas dos métodos de exclusão e de consulta, que não possuem pontos de junção bem definidos na linguagem AspectJ. Os métodos de inserção e alteração são automaticamente executados pelo framework com a captação dos construtores das classes, e da alteração de valores do objeto instanciado. A característica Conexão também possui variabilidades. A primeira é referente ao banco de dados (*DataBase*), e a segunda refere-se ao tipo de conexão. Essas variabilidades são definidas em tempo de instanciação.

Os FTs de Repositório de Conexões e Memória Auxiliar são características opcionais da Linha de Produtos, podendo ser utilizados na composição do framework para melhorar seu desempenho. O FT de Garantia de Políticas também é opcional e é utilizado somente durante o desenvolvimento do código-base, para garantir que certas políticas de modelagem atendam às exigências do FT de Persistência. Nenhum desses três FTs possui variabilidades, o que faz com que a utilização dessas características só possua a etapa de composição. As características opcionais Carregador de Objetos e Todas Tuplas da característica Memória Auxiliar, e as característica Verificador Get Set e Verificador de Atributos de Garantia de Políticas podem ser acopladas à estrutura do produto da família antes ou depois da composição com o código-base.

3.3.1.1 Estrutura do Framework Transversal de Persistência

As características de Operações Persistentes e de Conexão são o mínimo necessário para se utilizar um produto da família de FTs de persistência. Embora algumas características dessa família tenham sido implementadas de maneira genérica para que possam ser acopladas a qualquer código-base, essas duas em particular são totalmente dependentes entre si [Camargo e Masiero 2008]. A Figura 3.6 apresenta um diagrama de classes onde são apresentadas as classes e os aspectos presentes nos dois FTs (agrupados pelas linhas tracejadas), e a maneira como eles se relacionam. Os aspectos estão representados por retângulos cinza, e as setas que relacionam esses aspectos às classes ilustram a maneira como esses aspectos influenciam o comportamento de tais classes, seja por entrecorte (*crosscut*) ou pela inserção de métodos ou atributos (*inter type declaration*). Os métodos apresentados com o estereótipo <<*hook*>> são métodos abstratos e devem ser implementado pelo engenheiro de aplicação em uma classe concreta em tempo de composição.

A classe `TableManager` presente na Figura 3.6, assim como visto no padrão *Persistence Layer* (Seção 2.4.1) é responsável por montar e executar as instruções SQL dinamicamente. O aspecto *DirtyObjectController* é responsável por marcar objetos que foram alterados, fazendo as vezes dos métodos `setPersisted` e `setChanged` da *Persistece Layer*. É um sub-interesse implementado como um aspecto fixo dentro do sistema que entrecorta a classe `PersistentRoot`.

Quanto à utilização de aspectos no FT Operações Persistentes, um primeiro aspecto deve ser criado para indicar que uma classe de aplicação implementa a classe `PersistentRoot`, porém sem a necessidade de alterações invasivas no seu código. Neste caso, o aspecto utiliza o recurso de declaração intertipo da linguagem AspectJ. Ele também deve estender o aspecto `PersistentEntities`, que introduz na interface `PersistentRoot` um conjunto de atributos e métodos de persistência que serão herdados pelos seus sub-tipos.

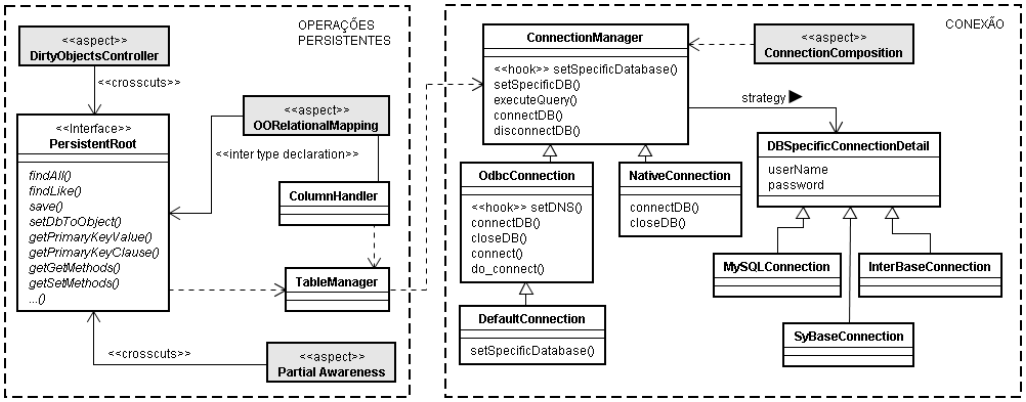


Figura 3.6: Estrutura de classes dos frameworks de Operações Persistentes e de Conexão [Camargo e Masiero 2008].

O segundo aspecto do FT refere-se ao mapeamento das classes que implementam `PersistentRoot` com o banco de dados. Esse mapeamento tem por objetivo inserir nos objetos dessas classes informações como o nome da tabela correspondente, os nomes das colunas presentes, etc. O seu funcionamento é encapsulado no aspecto `OORelationalMapping` e não exige inserção de código nos construtores das classes da aplicação que serão persistidas.

No FT de Conexão, `ConnectionComposition` é o único aspecto presente, responsável por iniciar e encerrar conexões com o banco de dados. Os pontos em que essas operações são executadas são informados pelo Engenheiro da Aplicação, com a criação de um aspecto concreto que estende `ConnectionComposition` e que contém os pontos de junção que serão entrecortados pelo aspecto.

3.3.1.2 Processo de Reúso do FT de Persistência

O processo de reúso de qualquer Framework Transversal (FT) possui duas etapas: Instanciação e Composição [Camargo e Masiero 2008].

A instanciação do FT de Persistência [Camargo e Masiero 2008] caracteriza-se pela definição de três variabilidades do sistema: o tipo de consciência do framework, o tipo de conexão com o banco de dados e o próprio banco de dados.

No caso do tipo de consciência, duas opções são possíveis: total ou parcial. A opção *default* da instanciação é a de consciência total. Para a escolha da opção de consciência parcial, basta que o aspecto `PartialAwareness` seja adicionado ao projeto. Esse aspecto entrecorta pontos das classes de aplicação persistentes para realizar a gravação de informações no banco de dados. Os pontos entrecortados são:

- a execução do construtor de uma classe com passagem de parâmetros, que realiza a gravação das informações dessa classe (método `save`);
- a execução de métodos set de uma classe, alterando o valor do objeto, que executa a atualização desta informação (método `update`).

Já as variabilidades de conexão são escolhidas pela extensão das classes derivadas de `ConnectionManager`. Pode-se escolher o banco de dados estendendo uma das classes que representam conexões com cada tipo de banco suportado pelo FT de Persistência (MySQL, InterBase ou SyBase). Caso seja estendida a classe `DefaultConnection`, tudo o que o engenheiro de software terá que informar será o *DNS* definido no Sistema Operacional.

Quanto à composição do FT de Persistência instanciado, ela é caracterizada pela inserção das operações de persistência nas classes de aplicação persistentes do código-base e pela definição dos pontos da aplicação em que a conexão com o banco de dados deverá ser aberta ou fechada. Isso é realizado implementando-se os aspectos abstratos correspondentes a cada mecanismo de composição.

O primeiro aspecto a ser concretizado é o `OORelationalMapping`. Este aspecto insere todas as operações de persistência na classe `PersistentRoot`, e o engenheiro de aplicação fica responsável por relacionar as classes de aplicação persistentes com a classe `PersistentRoot` com o uso de declarações inter-tipo (*intertype declarations*). A Figura 3.7 mostra um exemplo de implementação desse aspecto para uma aplicação hipotética na qual as classes `BaseSalary`, `Customer`, `Employee` e `Equipment` devem ser persistidas. Assim, o aspecto

`MyOORelationalMapping` é implementado herdando o aspecto `OORelationalMapping`, e usando os termos reservados `declare parents` para definir que as classes implementam a classe abstrata `PersistentRoot`. A partir disso, elas podem utilizar todos os métodos de persistência inseridos em `PersistentRoot`.

```
import persistence.OORelationalMapping;

public aspect MyOORelationalMapping extends OORelationalMapping {

    declare parents : BaseSalary        implements PersistentRoot;
    declare parents : Customer          implements PersistentRoot;
    declare parents : Employee          implements PersistentRoot;
    declare parents : Equipment         implements PersistentRoot;

}
```

Figura 3.7: Exemplo de código para inserção de métodos em classes de aplicação persistentes.

O próximo passo é informar ao framework de Conexão quais os pontos do código-base que deverão abrir e fechar a conexão com o banco de dados. Esses pontos são definidos na implementação dos *pointcuts* do aspecto abstrato *ConnectionCompositionRules*. A Figura 3.8 mostra um exemplo da implementação desse aspecto. Nessa implementação, o aspecto *MyConnectionCompositionRules* implementa os *pointcuts* informando que tanto a abertura quanto o fechamento da conexão acontecem na execução de um método `main`. O advice que executa o comportamento transversal que abre a conexão com a palavra reservada *before* (antes) do *joinpoint openConnection*. O comportamento que fecha a conexão é realizado com a palavra *after* (depois) do *joinpoint closeConnection*. Dessa maneira, define-se que a abertura da conexão será realizada antes da execução dos métodos definidos na concretização do pointcut *openConnection* e depois dos definidos para *closeConnection.*

A adição de novas características ao FT, como Memória Auxiliar, *Pooling* ou Garantia de Políticas pode ser realizada antes ou depois da composição do FT com o código-base. Essas características não possuem variabilidades e são acopladas à estrutura do FT de Persistência em si, e não no código-base.

```
import persistence.connection.ConnectionComposition;


public aspect myConnectionCompositionRules extends ConnectionComposition {

    public pointcut openConnection():
        execution (public static void *.main(..));
    public pointcut closeConnection():
        execution (public static void *.main(..));

    (...)
}
```

Figura 3.8: Exemplo de código para definição de pontos de abertura e fechamento da conexão com o banco de dados.

3.4 Considerações Finais

O framework de aplicação GRENJ auxilia desenvolvedores na construção de aplicações de gestão de recursos de negócio, promovendo reúso de projeto para sistemas dentro deste domínio. Além disso, por ser baseado em padrões de análise, promove também o reúso da análise desse domínio, facilitando a instanciação de aplicações específicas. Porém, a presença de interesses de persistência espalhados pelo seu código pode dificultar o entendimento de sua estrutura, aumentando o custo da instanciação de aplicações, além de responsabilizar o engenheiro de aplicação pela implementação de detalhes de persistência, tornando mais custoso o seu trabalho.

O Framework Transversal de Persistência apresentado neste capítulo se apresenta como uma alternativa para realizar o isolamento adequado do interesse de persistência no GRENJ. Em primeiro lugar por ter sido implementado em linguagem AspectJ, derivada da linguagem Java, e em segundo lugar por que foi desenvolvido baseado no padrão *Persistence Layer*, mesmo padrão de projeto usado na camada de persistência original do GRENJ.

O próximo capítulo apresentará o processo de substituição da camada de persistência originalmente implementada no GRENJ por um produto da família de frameworks transversais desenvolvido por Camargo e Masiero (2008) assim como as adaptações necessárias nos dois frameworks para que essa substituição seja feita.

CAPÍTULO 4
Manutenções Realizadas

4.1 Considerações Iniciais

O GRENJ [Durelli 2008] é um framework de aplicação que auxilia o desenvolvimento de sistemas de gestão de recurso de negócios e, como apresentado na Seção 3.2, possui interesses de persistência entrecortando as classes de negócio presentes em sua estrutura. Uma alternativa para a realização do isolamento desses interesses é o acoplamento de Frameworks Transversais à estrutura do GRENJ. Com esse intuito, foi feita uma manutenção evolutiva na camada de persistência desse framework de aplicação, acoplando a essa camada um FT de Persistência desenvolvido por Camargo e Masiero (2008). Este Framework Transversal realiza a modularização dos interesses de persistência em sistemas de software. Porém, não foram encontradas bibliografias que descrevam experiências do seu acoplamento com estruturas que devem ser especializadas, como os frameworks de aplicação.

Neste capítulo é apresentado o acoplamento do framework transversal de persistência ao framework de aplicação GRENJ. Também serão apresentadas as atividades de manutenção realizadas nos dois frameworks para possibilitar esse acoplamento. Na Seção 4.2 é apresentado o processo utilizado na realização do acoplamento dos dois frameworks. Na Seção 4.3 todas as manutenções realizadas

durante a execução do processo, tanto no GRENJ quanto no FT de Persistência, são apresentadas em detalhes. Na Seção 4.4 são apresentadas as considerações finais deste capítulo.

4.2 Processo de Acoplamento

Com o objetivo de melhorar a modularização do framework de aplicação GRENJ, separando a camada de modelo do interesse de persistência, foi realizada uma manutenção evolutiva em sua estrutura, em que a camada de persistência originalmente implementada foi substituída por um produto da família de frameworks transversais desenvolvidos por Camargo e Masiero (2008).

Dessa maneira, as classes do GRENJ que devem ser persistidas deixam de estender a camada de persistência original, mostrada em detalhes na Seção 3.2, e passam a ser entrecortadas pelo FT de Persistência, como mostrado na Figura 4.1. Nessa figura, são mostradas de forma esquemática as camadas do framework antes e depois da manutenção efetuada, bem como a relação entre essas camadas. Para identificar o novo framework de aplicação, a estrutura modificada será chamada de GRENJ-FT (Figura 4.1 (b)).

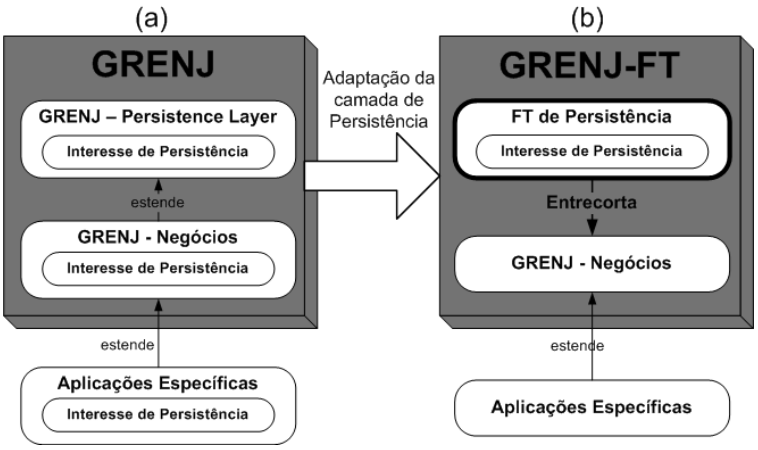


Figura 4.1: Arquitetura do GRENJ antes e depois da manutenção evolutiva.

Para o acoplamento com o GRENJ foi utilizado um produto da família de FTs de Persistência composto pelas características de conexão, de operações persistentes, e de *caching*. As características de conexão e de operações persistentes são obrigatórias em qualquer produto da família desenvolvida por Camargo e Masiero (2008). Já a característica de *caching* foi utilizada para melhorar a performance da persistência. Quanto às variabilidades do produto utilizado, foram escolhidas:

- Banco de dados MYSQL, uma vez que esse é o banco de dados originalmente oferecido pelo GRENJ. Essa variabilidade pode ser alterada diretamente no FT de Persistência sem afetar sua funcionalidade ou seu acoplamento com algum código-base;

- Tipo de conexão nativa, facilitando assim futuras alterações de conexão, caso queira-se alterar o tipo de banco de dados utilizado na instanciação do GRENJ;

- Consciência total. Essa variabilidade foi escolhida para evitar diminuição da performance do GRENJ. A escolha pela consciência parcial resultaria em um número excessivo de acessos ao banco de dados para que as informações fossem persistidas, pois características internas do framework de aplicação fazem com que classes sejam criadas e tenham seus valores alterados constantemente durante a execução de aplicações por ele instanciadas.

Após a definição do FT de Persistência a ser utilizado, foi feito o acoplamento com o framework de aplicação. Este acoplamento, como visto em detalhes na Seção 3.3, corresponde à definição dos pontos de abertura e fechamento de conexão (característica de Conexão), e a definição das classes que herdarão os métodos de persistência do Framework Transversal (característica de Operações Persistentes).

O processo de acoplamento da conexão pôde ser feito sem alterações no código do GRENJ, pois trata-se somente de indicar pontos do código-base onde a conexão com o banco de dados será aberta ou fechada. Na Figura 4.2 é mostrado um exemplo da realização do aspecto `ConnectionCompositionRules`, presente no FT de Persistência, onde são indicados os pontos de entrecorte para que o comportamento de abertura e fechamento da conexão seja executado.

```
public aspect myConnectionCompositionRules extends ConnectionComposition {

    declare precedence: myConnectionCompositionRules, Persistence;

    private pointcut pcStaticObject():
        execution (grenj.model.StaticObject.new());

    private pointcut pcBusinessResourceTransaction():
        execution (grenj.model.BusinessResourceTransaction.new());

    private pointcut pcAbstractCalculator():
        execution (grenj.model.AbstractCalculator.new());

    public pointcut openConnection():
        pcStaticObject() ||
        pcBusinessResourceTransaction() ||
        pcAbstractCalculator();
```

Figura 4.2: Trecho de código com implementação do aspecto ConnectionComposition.

Como pode ser observado na Figura 4.2, os pontos indicados para serem entrecortados por este framework são a criação e a destruição de classes que representam classes persistentes.

O acoplamento do FT Operações Persistentes com o código-base é feito por meio da implementação do aspecto `OORelationalMapping`, informando a ele quais classes do código-base devem implementar a interface `PersistentRoot`, tendo assim acesso aos métodos de persistência implementados no FT. Esse acoplamento foi realizado em paralelo à remoção da camada de persistência original do GRENJ, gerando assim algumas modificações em sua estrutura. Dessa maneira o processo utilizado na realização desse acoplamento considerou a arquitetura do GRENJ, bem como a maneira como suas classes estavam hierarquicamente organizadas. Como é mostrado na Figura 4.3, a implementação original da camada de persistência do GRENJ foi feita de maneira que as classes que seriam persistidas herdassem uma classe em comum, chamada `PersistentObject`.

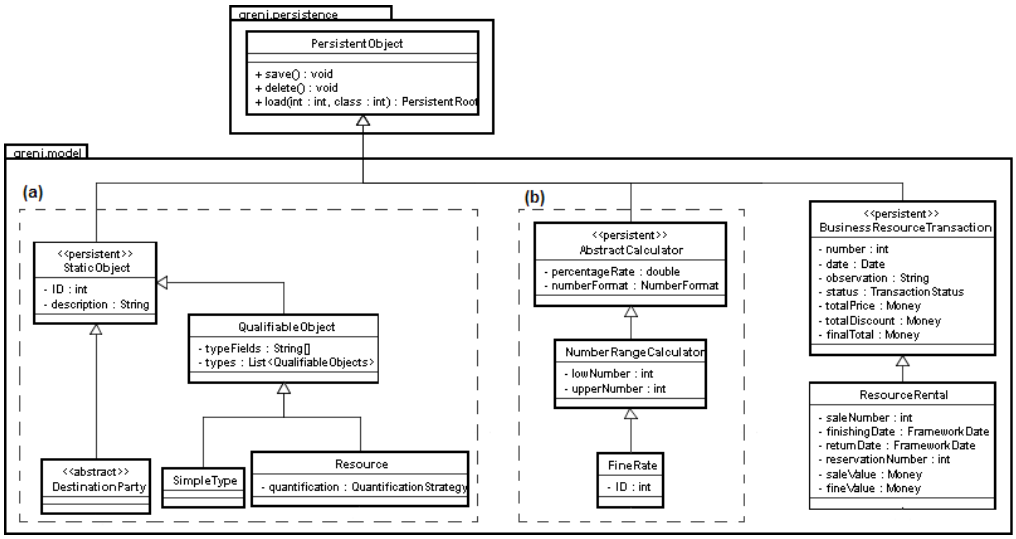


Figura 4.3: Diagrama de classes parcial do GRENJ.

Uma vez que a remoção da camada de persistência do GRENJ resulta em erros para as suas classes, o processo de substituição teve de ser feito de maneira controlada, garantindo que nenhuma funcionalidade seria comprometida durante o processo. Visando um melhor controle dessas modificações, o processo de manutenção seguiu uma abordagem incremental dividida por grupos de classes, chamados aqui de conjuntos de persistência. Cada conjunto de persistência é representado por uma classe

que herda diretamente a classe `PersistentObject`, chamada de raiz do conjunto de persistência, e pelas classes que a estendem na hierarquia do framework. Na Figura 4.3, são mostrados dois conjuntos de persistência como exemplo. O conjunto (a) é representado pela classe `StaticObject`, que representa a raiz do grupo, e suas classes-filha, como `QualifiableObject` ou `DestinationParty`. O conjunto (b) é representado pela raiz `AbstractCalculator`, e pelas classes `NumberRangeCalculator` e `FineRate`. O objetivo dessa divisão é garantir que as modificações ocorram em grupos separados de classes, de modo que as manutenções ocorridas em um grupo não afetem ou gerem erros em outros grupos.

Considerando a divisão apresentada, cada ciclo do processo incremental foi realizado em um conjunto de persistência da estrutura do GRENJ. Para cada conjunto de persistência foram efetuadas as seguintes atividades:

1. Remoção da relação de herança da classe raiz do conjunto de persistência com a classe `PersistentObject`. Essa modificação gerava erros nas classes do conjunto, pois estas deixavam de reconhecer os métodos de persistência presentes em sua estrutura;

2. Inserção de declaração inter-tipo no FT, fazendo com que a classe raiz do conjunto de persistência passe a implementar a interface `PersistentRoot` do FT de Persistência;

3. Remoção de código referente ao interesse de persistência que se encontrava entrelaçado nas classes do grupo hierárquico em questão;

4. Substituição dos métodos de persistência originais do GRENJ pelos correspondentes no FT de Persistência. Após este ponto, os erros gerados pela atividade do item 1 deixavam de existir;

5. Execução dos testes implementados na estrutura do GRENJ referentes às classes presentes no grupo hierárquico. Com isso, verificou-se se a funcionalidade do framework havia se mantido. Em casos de falhas nesses testes, eram feitas manutenções evolutivas adequadas para que eles obtivessem sucesso, indicando que a funcionalidade do GRENJ se mantinha. O próximo ciclo do processo só

era iniciado quando todos os testes referentes ao conjunto de persistência em questão tivessem sucesso.

Uma vez finalizado esse processo, tanto a característica de conexão quanto a de operações persistentes do FT de Persistência estavam acopladas ao GRENJ, tendo-se então o GRENJ-FT. Após o término da manutenção, ainda foi efetuada uma última alteração, onde foi criada uma camada de Operações Persistentes na estrutura do GRENJ, que será melhor detalhada na próxima Seção. Foram então realizadas instanciações desse novo framework para verificar se o processo de instanciação foi mantido em relação ao GRENJ, que serão apresentadas com detalhes no capítulo 5. As manutenções citadas nos itens 3, 4 e 5 desse processo, que correspondem às alterações realizadas em ambos os frameworks, serão comentadas a seguir.

4.3 Adaptações Efetuadas

Para que o FT de Persistência se acoplasse ao GRENJ mantendo exatamente as mesmas funcionalidades originais do framework de aplicação, foi necessário que algumas adaptações fossem realizadas na sua estrutura durante o processo de acoplamento, sendo que essas adaptações não se limitaram somente ao isolamento dos interesses de persistência da sua camada de modelo. Pôde-se notar, em primeiro lugar, que o FT utilizado não possuía todas as funções que o GRENJ necessitava para o tratamento da persistência. Sendo assim algumas melhorias foram feitas para que ele atendesse à essas exigências. Tais modificações foram feitas de modo que não fossem exclusivas do acoplamento com o GRENJ, mas para que pudessem ser utilizadas em futuros acoplamentos com outros códigos-base. Além disso, algumas exigências de persistência específicas do GRENJ foram implementadas em uma nova camada que entrecorta o FT de Persistência, isolando assim o interesse da camada de negócios e mantendo a generalidade do FT. Essas alterações são descritas com detalhes a seguir.

4.3.1 Adaptações do GRENJ

As manutenções realizadas na estrutura do GRENJ durante o processo de acoplamento com o FT de Persistência podem ser classificadas de duas maneiras:

- Adaptações de acoplamento: representadas por alterações necessárias para atender a exigências do FT, permitindo assim que o acoplamento fosse realizado adequadamente;

- Adaptações de isolamento: representadas por alterações que visam à separação do interesse de persistência, isolando-o da camada de modelo.

As adaptações de acoplamento foram necessárias porque o FT de Persistência, mais precisamente o módulo que representa a característica de Operações Persistentes, exige certas características do código-base para que o acoplamento ocorra de maneira adequada. Estas características devem existir em qualquer estrutura ao qual o FT se acople, e manutenções semelhantes às apresentadas devem ser realizadas sempre que esse FT se acoplar a um código-base que não as apresente. As adaptações de acoplamento foram as seguintes:

a) mudança do nome dos atributos que representavam identificadores das classes, que possuíam nomes diferentes em cada classe do framework, para ID, com tipo `int`. Originalmente o nome dos identificadores das classes persistentes do GRENJ não eram padronizados, o que fere a exigência do FT para o acoplamento;

b) todos os métodos `set()` que atribuem às classes valores do tipo `int` ou `double` foram alterados, passando agora a receber como parâmetro tipos `Integer` ou `Double`, respectivamente, pois os parâmetros dos métodos `set()` devem ser classes e não tipos primários;

c) o FT de Persistência trata as datas com o tipo `FrameworkDate`, dessa forma, os atributos que tratam desses valores com tipo `Date` foram alterados para esse tipo.

Ressalta-se que a uniformização do atributo de identificação das classes persistentes facilita o entendimento da estrutura do framework durante a sua instanciação. As demais alterações não resultaram em nenhuma modificação no processo de instanciação do GRENJ, mantendo sua generalização e seu escopo de aplicação inalterados.

Quanto às adaptações de isolamento, estas constituíam um dos objetivos principal do trabalho, pois tratam do isolamento da camada de persistência do GRENJ, tornando mais fácil a compreensão de sua estrutura e a sua utilização. Quanto a isso, foram realizadas duas modificações:

a) a utilização do framework transversal permitiu que fossem removidos trechos de código responsáveis pelo interesse de persistência que se encontravam entrelaçados nas classes de modelo do GRENJ. Esses trechos de código são referentes à chamada de métodos após alterações ou gravações das classes persistentes e à implementação de construtores responsáveis por alimentar informações do objeto por meio de dados recebidos de um `ResultSet` e métodos que informam à camada de persistência quais campos devem ser usados nas instruções de gravação e atualização. Na Figura 4.4 é mostrado um trecho de código da classe de aplicação `Cliente` antes e depois da manutenção. No código referente ao GRENJ pode-se observar que, na implementação do método `setEndereco()`, deve obrigatoriamente existir a chamada do método `setChanged()` logo após o atributo `endereco` receber um novo valor. Nesse mesmo código existe a implementação do método `updateSetClause()`, informando à camada de persistência quais campos devem ser utilizados nos métodos de alteração. No caso da implementação efetuada no GRENJ-FT, a chamada do método `setChanged()` não é mais necessária assim como não deve haver mais a implementação do método `updateSetClause()`.

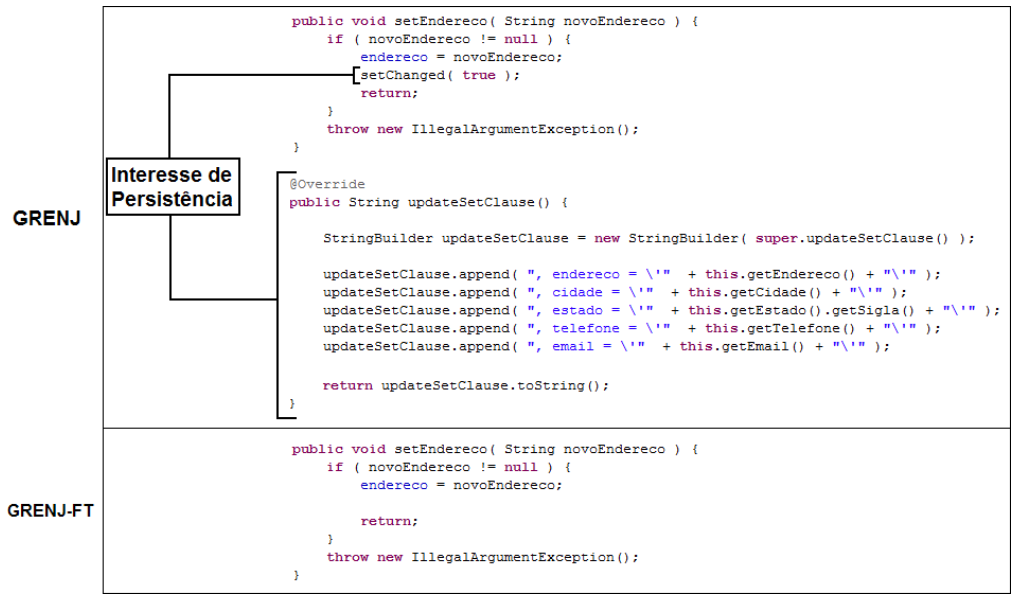


Figura 4.4: Mesmo trecho de código implementado com o uso do GRENJ e GRENJ-FT.

As modificações de isolamento garantiram maior inconsciência dos detalhes do interesse de persistência, tanto para as classes internas do framework quanto para as classes criadas nas aplicações instanciadas a partir dele. Os métodos apresentados na Figura 4.4, além de estarem presentes nas classes internas do GRENJ, também deveriam ser chamados ou implementados nas classes de aplicação criadas no momento da instanciação do framework. Assim, o trabalho de instanciação do GRENJ-FT torna-se mais fácil, pois o engenheiro de aplicação não tem a responsabilidade de implementar esses métodos.

b) a criação de uma camada de persistência no GRENJ que implementa o padrão Adapter [Gamma *et al.* 1995], pois os métodos de persistência implementados no GRENJ possuíam assinatura diferente da dos métodos correspondentes no FT. Um exemplo dessa diferença é a chamada do método `load()` presente no GRENJ, que é responsável por criar um objeto de determinada classe com informações recuperadas do banco de dados. Esse método solicita como parâmetro o número do identificador do registro que se quer carregar do banco de dados, e a classe a partir da qual se quer criar o objeto. O método `load()` é chamado de maneira estática pela classe `PersistentObject`, e retorna um objeto genérico do tipo `PersistentObject` que deve ser convertido para o tipo da classe específico que se quer instanciar. No caso do FT de Persistência, não existe um método correspondente que apresente esse comportamento. Os métodos de busca do FT devem ser chamados a partir de objetos instanciados da própria classe que se quer carregar, a partir do número identificador passado como parâmetro. Na Figura 4.5 são mostrados trechos de código em que são chamados métodos de persistência para o carregamento de um objeto segundo o mecanismo original do GRENJ, e segundo o FT de Persistência. No código referente ao GRENJ, é mostrada a chamada do método `loadAll()` pela classe estática `PersistentObject`. Já no referente ao GRENJ-FT, é criado um objeto do tipo `PersistentRoot`, e então a partir dele é chamado o método `findByField()`.

Para realizar a adaptação das chamadas dos métodos de persistência foi preciso identificar os métodos utilizados no GRENJ, apontar os correspondentes no FT de Persistência, e realizar a substituição da chamada no código do framework de aplicação. Notou-se que a simples substituição da chamada desses métodos no código do GRENJ representaria uma alteração invasiva no seu código, que o tornaria dependente do FT de

Persistência, pois os seus métodos estariam explicitamente declarados na estrutura do framework de aplicação. Para evitar essa dependência, e procurando generalizar as chamadas de métodos do GRENJ, foi criada uma camada de operações persistentes na qual é implementado o padrão de projeto Adapter [Gamma *et al.* 1995].

GRENJ

```
List<PersistentObject> transactionList = new ArrayList<PersistentObject>();

try {

    PersistentObject.loadAll( transactionList, "status = 1", "date", transactionClass );

} catch ( Exception e ) {
```

GRENJ-FT

```
List<PersistentObject> transactionList = new ArrayList<PersistentObject>();

try {

     Vector fields = new Vector();
     Vector values = new Vector();
     Vector clauses = new Vector();

     List<String> statement = breakCondition("status = 1");
     for(int i = 0; i <= statement.size(); i = i+4){
         fields.addElement(statement.get(i));
         values.addElement(statement.get(i+1));
         clauses.addElement(statement.get(i+2));
     }

     PersistentRoot obj = null;

     try{
         obj = (PersistentRoot) aClass.newInstance();
     }catch(Exception e){
         System.out.println("Erro na instanciação do objeto");
     }

     ResultSet rs = obj.findByField(fields, values, clauses);

     transactionList = fromResultsetToList(rs, obj);

 } catch ( Exception e ) {
```

Figura 4.5: Chamadas de métodos de persistência segundo a *Persistence Layer* e segundo o FT de Persistência.

Essa camada é composta de uma interface que contem as assinaturas de todos os métodos de persistência requeridos pelo GRENJ. Essa interface deve ser então concretizada de forma a se implementar os métodos de persistência de acordo com o mecanismo de persistência utilizado. Dessa maneira, as classes de modelo do GRENJ passam a herdar essa classe concreta tendo acesso às operações de persistência por meio dessa herança. A implementação desse padrão permitiu que todas as chamadas de métodos do GRENJ se mantivessem as mesmas, minimizando as alterações no código desse framework. Além disso, essa nova camada facilita o trabalho de futuras alterações no mecanismo de persistência do GRENJ, bastando, para isso, criar uma nova classe concreta que implemente a interface presente na camada de operações persistentes, e relacionar as classes do modelo com essa nova concretização. Na Figura 4.6 é mostrada a camada de operações persistentes inserida no GRENJ-FT e sua relação com as demais camadas. Pode-se observar que as classes de negócio do GRENJ, Figura 4.7, estendem a classe `PersistentObjectFT`, que concretiza a interface `PersistenceInterface`,

implementando os métodos de persistência de acordo com a lógica do FT de Persistência utilizado. As classes `PersistentObjectX` e `PersistentObjectY` são implementações hipotéticas da interface de operações persistentes o que possibilita a utilização de outros mecanismos de persistência, caso se queira substituir o FT de Persistência por algum outro meio.

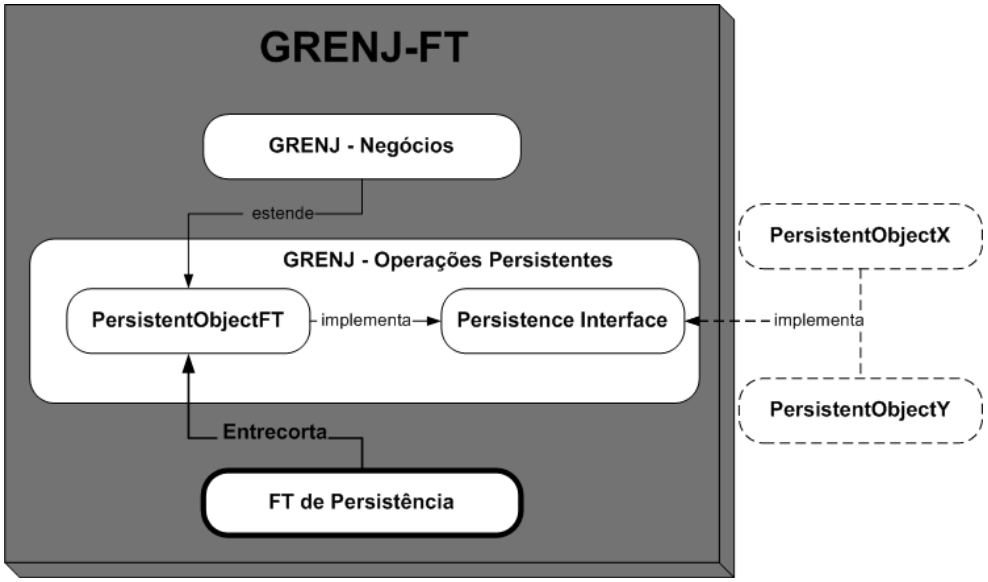


Figura 4.6: Camada de operações persistentes criada no GRENJ-FT.

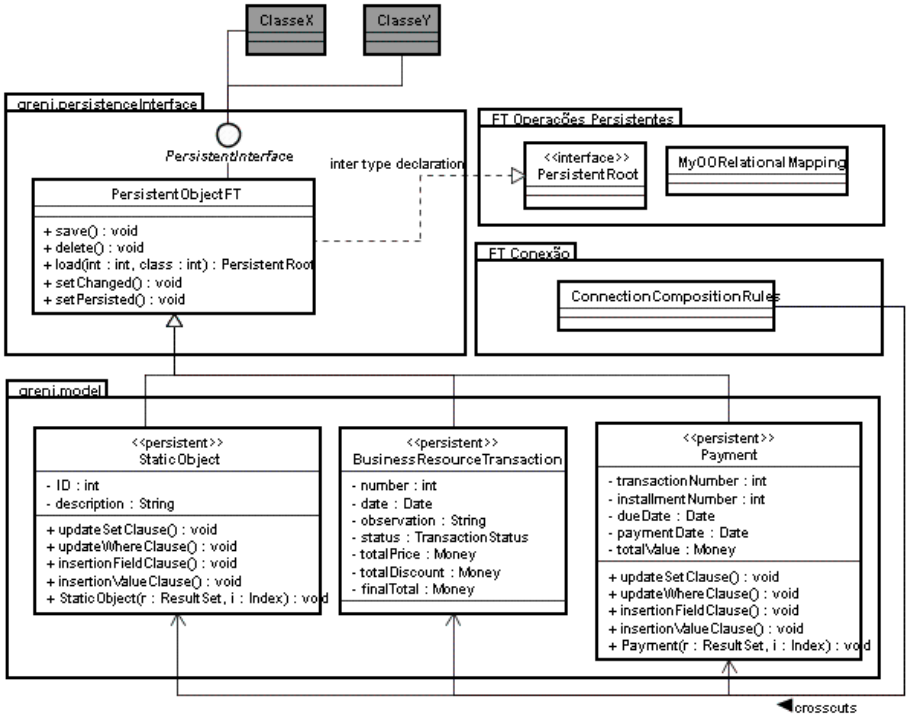


Figura 4.7: Diagrama de classes com a camada de operações persistentes criada no GRENJ-FT.

No caso da implementação realizada neste projeto, a classe concreta `PersistentObjectFT` é entrecortada pelo FT de Persistência, tendo então acesso aos métodos de persistência, que serão utilizados na implementação dos métodos definidos na interface.

4.3.2 Adaptações no Framework Transversal

Para que o Framework Transversal de persistência pudesse ser acoplado ao código do GRENJ, também foram necessárias algumas modificações em sua estrutura. Essas modificações foram necessárias pois as funcionalidades oferecidas por esse FT não atendiam a todas as necessidades de persistência do framework de aplicação GRENJ. Pode-se observar que as exigências não atendidas pelo FT podem estar presentes em qualquer outro sistema ao qual ele se acople, não sendo, portanto, uma exigência decorrente do fato do código-base ser um framework de aplicação. Todas as alterações feitas se caracterizaram como melhorias podendo ser utilizadas no acoplamento deste FT com qualquer outro código-base.

A primeira melhoria no FT de Persistência refere-se ao reconhecimento dos tipos de dados `Money` e de atributos enumeráveis (`Enum`).

O tipo de dado `Money` é nativo da linguagem Java e utilizado para o tratamento de valores monetários. Esse tipo de dado é utilizado no GRENJ principalmente no que se refere ao tratamento dos valores das transações geridas pelo sistema. Dessa maneira, a estrutura do FT foi alterada, fazendo com que ele reconhecesse o tipo de dado `Money` e o convertesse para um valor numérico (`Double`), gravando o seu valor original no banco de dados, e recuperando esse valor quando necessário.

Os atributos enumerados (`Enum`) presentes em diversos pontos do GRENJ são um exemplo de outro tipo de dados não reconhecido pelo GRENJ. Esses atributos foram usados para a identificação de campos multivalorados com valores pré-definidos. Por exemplo, o atributo `status`, tipo ENUM no GRENJ, da classe Resource, classifica o recurso na aplicação instanciada como “Disponível” (*Available*) ou “Indisponível” (*Unavailable*). A alteração feita no FT de Persistência permite que ele reconheça atributos desse tipo e que grave no banco de dados uma `String` referente ao valor representado.

Outra necessidade se referia à presença de atributos nas classes persistentes do GRENJ que não deveriam ser persistidos, ou seja, não deveriam ser reconhecidos pelo FT como campos na tabela correspondente àquela classe. Assim, foi criado um mecanismo para que o engenheiro de aplicação informe esses atributos ao FT em tempo de acoplamento. Esse mecanismo corresponde ao método abstrato `getNonPersistentFields()`, criado na interface `OORelationalMapping`. Esse método deve ser implementado no momento do acoplamento do FT com um código base, adicionando em uma lista do tipo `Hashtable` informações referentes aos campos que não devem ser persistidos. Na Figura 4.8 é mostrada um exemplo da implementação desse método. Pode-se observar que, para a classe `FitaDeVideo`, o campo auxiliar `p` não deve ser persistido; o mesmo ocorre com o campo auxiliar `numberFormat`, presente na classe `Multa` e `Taxa`.

Outra melhoria realizada no FT de Persistência foi a criação de métodos de persistência requeridos pelo GRENJ que não podiam ser supridos pelos existentes no FT. A Figura 4.9 apresenta como exemplo para esse caso, dois métodos: um que se refere à busca de registros que se enquadram em uma determinada cláusula de comparação, como intervalos de datas, ou valores maiores ou menores que determinado valor; outro que se refere à deleção de registros que se enquadram também a uma determinada comparação.

```
private Hashtable PersistentRoot.getNonPersistentFields(){
    Hashtable nonPersistableFields = new Hashtable();

    nonPersistableFields.put(Filme.class.getName(), new String[]{"types", "typeFields", "quantification"});
    nonPersistableFields.put(Carro.class.getName(), new String[]{"types", "typeFields", "quantification"});
    nonPersistableFields.put(Genero.class.getName(), new String[]{"types", "typeFields"});
    nonPersistableFields.put(FitaDeVideo.class.getName(), new String[]{"p"});

    nonPersistableFields.put(Multa.class.getName(), new String[] {"numberFormat"});
    nonPersistableFields.put(Taxa.class.getName(), new String[] {"numberFormat"});

    return nonPersistableFields;
}
```

Figura 4.8: Trecho da implementação do método getNonPersistentFields().

Todas as alterações realizadas no FT de Persistência, mesmo as motivadas por necessidades do GRENJ, se apresentaram como um aperfeiçoamento das funções por ele realizadas, podendo ser úteis a qualquer outro código-base que possuam essas mesmas necessidades. Dessa maneira, tem-se o aumento do escopo de aplicação do FT sem modificar o seu processo de reúso.

```
public ResultSet PersistentRoot.findByField(Vector field, Vector fieldValue, Vector comparisonClause){

    Vector clauseV = new Vector();
    Vector parameter = new Vector();

    parameter.addAll(fieldValue);
    for (int i=0; i < field.size(); i++){
        String clause = field.get(i) + " " + comparisonClause.get(i) + " ";
        clauseV.addElement(clause);
    }

    return TableManager.findlikeDB (this.getTableName(), clauseV, parameter);

}
public boolean PersistentRoot.delete(Vector field, Vector fieldValue, Vector comparisonClause){

    Vector clauseV = new Vector();
    Vector parameter = new Vector();

    parameter.addAll(fieldValue);
    for (int i=0; i <= field.size(); i++){
        String clause = field.get(i) + " " + comparisonClause.get(i) + " ";
        clauseV.addElement(clause);
    }

    return TableManager.deleteDB(this.getTableName(), clauseV, parameter);

}
```

Figura 4.9: Exemplos de métodos criados no FT de Persistência, visando suprir necessidades do GRENJ.

4.3.3 Criação de uma Camada Intermediária

Outra alteração na adaptação da camada de persistência do GRENJ foi a criação de uma camada intermediária para o tratamento de um interesse de persistência específico desse framework de aplicação: a presença de atributos no GRENJ cujos tipos representam *hotspots* do framework, ou seja, pontos que devem ser instanciados para a criação de uma aplicação específica. Logo, algumas classes do GRENJ possuem referências a classes genéricas que serão instanciadas pelo engenheiro de aplicação no momento da criação de uma aplicação específica. Dessa maneira, o FT de persistência não tem acesso à informação do nome da tabela correspondente àquela classe com o simples acesso ao atributo em questão, devendo consultar métodos do GRENJ, concretizados em tempo de instanciação, que informam o nome das classes da aplicação específica correspondente. Nota-se que este problema ocorre especificamente por conta do GRENJ ser um framework de aplicação, sendo que ainda vai ser instanciado em uma aplicação e que sua estrutura por si só, não possui todas as informações que o FT necessita.

Um exemplo detalhado dessa característica no GRENJ é encontrado na classe `ResourceRental`, estendida em sistemas de tratamento de Aluguel. Essa classe possui um atributo do tipo `DestinationParty`. De acordo com a estrutura do framework a classe `DestinationParty` representa o destino da transação, seja esta transação uma

compra, venda ou aluguel. Para a instanciação de uma aplicação hipotética, a classe `DestinationParty` pode ser estendida pela classe `Cliente`, sendo que essa possui novos atributos como telefone e e-mail. Da mesma maneira, a classe `ResourceRental` é estendida pela classe `Locacao`. O engenheiro de aplicação deve então, em tempo de instanciação, informar à classe `Locacao` que o atributo `DestinationParty` é representado na aplicação específica pela classe `Cliente`, por meio da implementação do método abstrato `getDestinationPartyClass()` presente na classe `ResourceRental`. Desta maneira é possível para a camada de persistência saber, no momento de carregar as informações para um objeto do tipo `Locacao`, que o atributo `DestinationParty` é representado pela classe `Cliente`, e que as informações devem vir da tabela correspondente a essa classe no banco de dados.

Logo, para que o FT de Persistência possua todas as informações da classe `Aluguel`, não é suficiente que ele acesse simplesmente os atributos desta classe, devendo também acessar o método `getDestinationPartyClass()`para ter acesso às informações destes campos na tabela correspondente ao aluguel. Um problema com o tratamento desse tipo de interesse é que, ao mesmo tempo que o comportamento responsável por verificar a relação da classe `DestinationPaty` com `Cliente` deveria ficar isolado da camada de modelo do GRENJ, ele também não poderia ser inserido diretamente na estrutura do FT de Persistência, por ser específico desse tipo de framework de aplicação. Com o objetivo de manter o isolamento desse interesse sem afetar a generalidade do FT de Persistência, o tratamento desse tipo de atributo foi implementado em uma nova camada, intermediária aos dois frameworks. Essa camada possui um aspecto que entrecorta certos métodos do FT de Persistência, obtendo o resultado da execução desses métodos e adicionando a esse resultado informações específicas segundo a lógica requerida pelo GRENJ. Na Figura 4.10 pode-se observar esquematicamente a camada intermediária que realiza esse comportamento. Pode-se observar que essa camada não está diretamente presente na estrutura do FT de Persistência, bem como não faz parte do GRENJ-FT, mostrando-se então como uma camada externa necessária para o acoplamento desses dois frameworks.

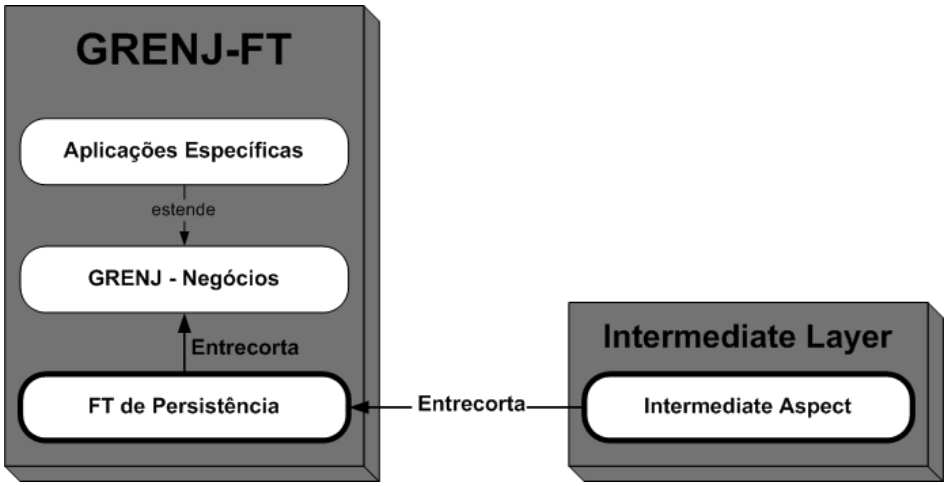


Figura 4.10: Camada intermediária para tratamento de persistência e sua relação com o GRENJ-FT.

Um exemplo de método entrecortado por essa camada intermediária pode ser vista na Figura 4.11, na qual é apresentada a recuperação de informações da classe `TransactionItem`, que possui um atributo genérico do tipo `Resource`. Nela, observa-se a existência de um *pointcut* que entrecorta o método `setDBToObject`() quando a classe entrecortada herdar a classe `TransactionItem`. Em seguida, é apresentado um *advice around* para o *pointcut* em questão. A chamada do método `proceed`() (Figura 4.11 (a)), referente à chamada original de `setDBToObject`, no início do *advice*, retornará um objeto de `TransactionItem`, mas com o atributo `resource` vazio. A chamada do método `getResourceClass`() (Figura 4.11 (b)), será implementado em tempo de instanciação do GRENJ, retornando a classe correta a se carregar. A seguir, o carregamento de um objeto genérico do tipo `Resource` (Figura 4.11 (c)), recupera as informações da tabela com o nome da classe de aplicação, informada como parâmetro (`resourceClass`). O atributo `resource` do objeto `transactionItem` é então alimentado (Figura 4.11 (d)) com as informações recuperadas em (c).

```
        // --------- PointCuts referentes ao método SetDBToObject ---------
        public pointcut pcSetDBToObject():
            execution (public PersistentRoot PersistentRoot.setDBToObject(ResultSet));

        public pointcut pcSetDBToObject_TransactionItem():
            pcSetDBToObject() &&
            this(grenj.model.TransactionItem);

         PersistentRoot around(): pcSetDBToObject_TransactionItem(){
(a)---------- PersistentRoot pr = proceed();
            PersistentObjectFT poft = new PersistentObjectFT();
            TransactionItem transactionItem = (TransactionItem) pr;

            Class< ? extends PersistentRoot> resourceClass = null;

            try {
(b)----------    resourceClass = transactionItem.getResourceClass();
            } catch (RuntimeException e) {

                e.printStackTrace();
            }

            Resource resource = null;
            ResultSet rs = pr.findlike("ID", pr.getID());

            try {
                if (rs.next()){

(c)----------------    resource = (Resource)poft.loadObject(rs.getInt("resource"), resourceClass);

                }
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            if (null != resource){
(d)----------  transactionItem.setResource(resource);
            }

            return pr;
        }
```

Figura 4.11: Trecho de código com exemplo de entrecorte de método do FT de Persistência.

4.4 Considerações Finais

Neste capítulo observou-se que um framework transversal pode ser utilizado para o encapsulamento do interesse transversal de persistência presente no GRENJ, desde que manutenções evolutivas sejam realizadas. Essas manutenções foram feitas visando manter a generalização e o escopo de atuação dos frameworks, bem como a sua funcionalidade.

Quanto às adaptações realizadas para que o acoplamento do FT fosse possível no GRENJ, pôde-se observar que elas resultaram em ganhos para ambos. No caso do FT de persistência, todas as melhorias realizadas podem ser mantidas em sua estrutura, sendo também utilizadas no acoplamento com qualquer outro código-base, não sendo este necessariamente um framework de aplicação. Quanto ao GRENJ, além do melhor isolamento do interesse de persistência facilitar a compreensão da sua estrutura e a sua instanciação, a interface implementada segundo o padrão de projeto *Adapter* deu maior flexibilidade à persistência do framework, facilitando futuras manutenções ou alterações no mecanismo de persistência utilizado.

A característica de Conexão do FT de Persistência mostrou-se plenamente funcional mesmo sem qualquer modificação em quaisquer das estruturas, pois possuía as informações de entrecorte necessárias para o seu acoplamento. Neste caso, o entrecorte pôde ser feito diretamente nas classes do framework, não havendo dependências desse módulo com informações da aplicação instanciada.

Importante observar que grande parte das alterações realizadas não foram decorrentes de incompatibilidades das estruturas dos frameworks, no caso, um framework transversal e um framework de aplicação. Em primeiro lugar, as alterações de isolamento realizadas no GRENJ correspondem ao objetivo do framework de aplicação. Já as adaptações de acoplamento foram realizadas visando obedecer a regras exigidas pelo FT de Conexão. Logo, independente do tipo de software ao qual o FT está sendo acoplado, alterações semelhantes devem ocorrer caso as exigências não estejam sendo atendidas. Quanto ao FT de Persistência, suas melhorias foram realizadas com o objetivo de atender a necessidades do código-base ao qual ele estava se acoplando. Assim, caso a funcionalidade requerida fosse encontrada em qualquer outro código-base, independente de sua arquitetura, essas manutenções seriam realizadas da mesma maneira.

A única modificação oriunda do fato do código-base ser um framework de aplicação foi a criação da camada intermediária para tratamento das informações da aplicação instanciada. Isso foi exigido por que o GRENJ não possui, por si só, as informações para que o FT relacionasse os atributos da aplicação entrecortada com os campos correspondentes em uma tabela. Logo este FT de Persistência, por depender diretamente da estrutura da aplicação à qual ele está acoplado, pode ter problemas no tratamento da persistência de aplicações desenvolvidas segundo a arquitetura do GRENJ.

O próximo capítulo apresentará três estudos de caso desenvolvidos por meio da instanciação do GRENJ-FT com o uso de requisitos obtidos para aplicações a serem geradas com o uso do GRENJ. Esses estudos têm a finalidade de verificar que o processo de instanciação de aplicações do GRENJ se manteve inalterado com a inclusão do FT de Persitência.

CAPÍTULO 5
Estudos de Caso

5.1 Considerações Iniciais

As adaptações realizadas no GRENJ apresentadas no capítulo anterior deram origem ao GRENJ-FT, framework de aplicação que possui um melhor isolamento do interesse de persistência em sua estrutura por ser acoplado a um framework transversal de persistência. Além disso, mantém a mesma funcionalidade do framework GRENJ, fato esse verificado com a execução dos mesmo casos de testes usados para sua construção.

Após a obtenção do GRENJ-FT, serão feitas instanciações nesse framework para se gerar novas aplicações. O processo de instanciação será realizado com o GRENJ-FT para verificar se tem o mesmo comportamento que com o GRENJ [Durelli 2008] ou se são necessárias alterações. Pretende-se também verificar o custo de se adaptar sistemas já existentes, desenvolvidos com o uso do GRENJ, de modo que possam utilizar o GRENJ-FT.

O processo de instanciação procurou cobrir as possibilidades mais recorrentes de sistemas instanciados que o GRENJ original prevê. Assim, este capítulo trata de três estudos de caso que contemplam os três tipos de transações básicas oferecidas pelo GRENJ, sendo elas venda, aluguel e manutenção. Além disso, a instanciação de cada um foi feita de maneira diferente, sendo que, em um dos casos, foi realizada a

instanciação de uma aplicação nova, e nos outros dois, foram feitas adaptações em sistemas originalmente instanciados com o apoio do GRENJ.

Na Seção 5.2 deste capítulo é mostrado um sistema para tratamento de vendas, instanciado utilizando somente o GRENJ-FT, a partir de requisitos preparados para um sistema originalmente instanciado pelo GRENJ. Na Seção 5.3 é apresentado um sistema de locadora, já instanciado com o uso do GRENJ [Durelli 2008], que foi adaptado para estender o GRENJ-FT, sendo estimado o custo dessa adaptação. Na Seção 5.4, é apresentado um sistema de manutenção em que foi utilizado o GRENJ, com a camada de interface, desenvolvida por Viana (2009). Na Seção 5.5 são apresentadas as considerações finais sobre o uso do GRENJ-FT.

5.2 Estudo de caso 1: Sistema para tratamento de vendas

O primeiro estudo de caso trata da instanciação de um sistema de vendas com o uso do GRENJ-FT, cujo intuito é criar uma instância que utilize o sexto padrão da GRN Comercializar o Recurso, responsável pelo tratamento de transação de compra e venda de recursos. Para essa instanciação foi considerado o seguinte conjunto de requisitos para um Sistema de Controle de Banca de Revistas (SCBR):

1. O sistema deve permitir a inclusão, alteração e remoção de itens. Esses itens podem ser revistas em quadrinhos, livros, jornais, ou qualquer outro produto vendido em bancas de revista. Cada item possui os seguintes atributos: código, descrição, valor e categoria.

2. O sistema deve permitir a inclusão, alteração e remoção de fornecedores de itens para a banca, com os seguintes atributos: código, nome, endereço, cidade, estado, telefone, email, fax e nome do contato.

3. O sistema deve permitir a inclusão, alteração e remoção de fornecimento de exemplares, com os seguintes atributos: número, data de entrada, fornecedor e valor total. Associada ao fornecimento existe uma relação de exemplares contendo os itens fornecidos, a quantidade e o valor unitário de cada um. O valor total do fornecimento é igual à soma do valor unitário de cada exemplar multiplicado pela quantidade fornecida. A quantidade de exemplares de um

item na banca corresponde à soma das quantidades de todos os fornecimentos desse item subtraído da soma das quantidades de todas as suas vendas.

4. O sistema deve permitir a inclusão, alteração e remoção de venda de exemplares. Cada venda possui os seguintes atributos: número, data, hora, valor total e forma de pagamento. Associada à venda existe uma relação de exemplares, contendo os itens vendidos, a quantidade e o valor unitário de cada um. O valor total da venda é igual à soma do valor unitário de cada exemplar multiplicado pela quantidade vendida. O sistema não deve permitir a venda de um item cuja quantidade de exemplares é igual a zero.

5. O sistema deve permitir apenas pagamento à vista para uma venda com as seguintes opções: 1) dinheiro; 2) débito automático; e 3) cartão de crédito.

O processo de instanciação do GRENJ, detalhado na Seção 3.2.2, define que a linguagem de padrões GRN pode ser utilizada como apoio na criação de um sistema, relacionando os padrões da linguagem aos requisitos do sistema. Dessa maneira, os requisitos foram confrontados com a GRN sendo que os padrões exibidos na Figura 5.1 atendem aos requisitos levantados.

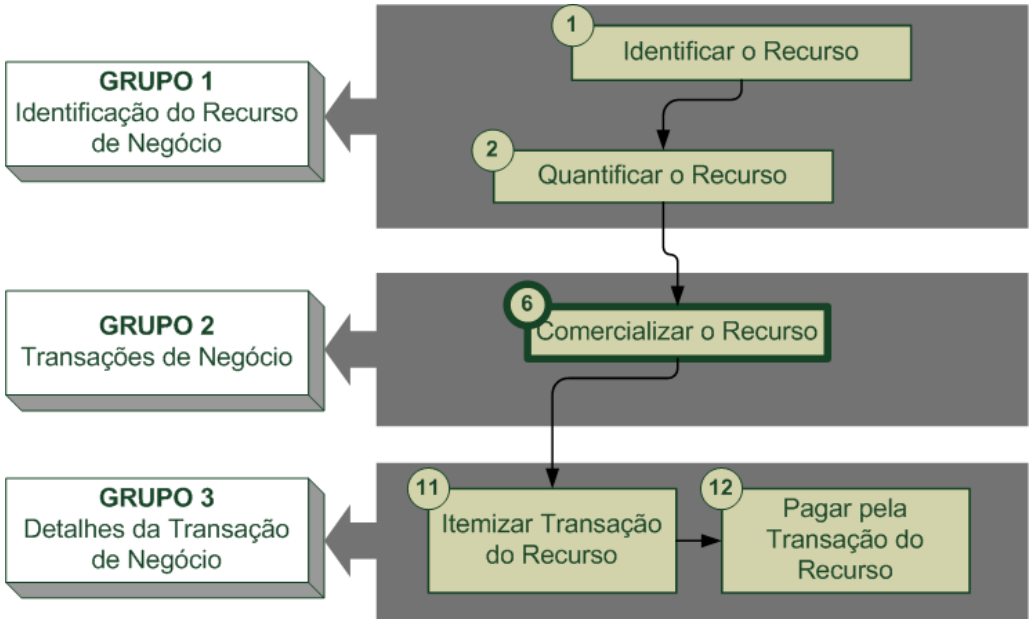


Figura 5.1: Padrões da GRN usados no sistema de controle de vendas.

Cada padrão identificado atende a um conjunto de requisitos do SCBR como mostrado na Tabela 5.1.

Tabela 5.1: Relação dos padrões da GRN com os requisitos do SCBR.

Padrões da GRN	Requisito do SCBR	Justificativa
1 e 2	1	os itens de venda representam o recurso a ser comercializado.
6	2, 3 e 4	trata da comercialização (compra e venda) dos recursos, bem como o gerenciamento das informações da origem (fornecedores) e as de destino (clientes) da transação.
11	4	refere-se à possibilidade de uma mesma transação envolver vários recursos
12	5	trata do pagamento da transação realizada

Uma vez relacionados os requisitos com os padrões da linguagem GRN, é possível definir as classes de aplicação que serão criadas de acordo com essa linguagem de padrões. Na Figura 5.2 é exibido o modelo de classes parcial para esse sistema.

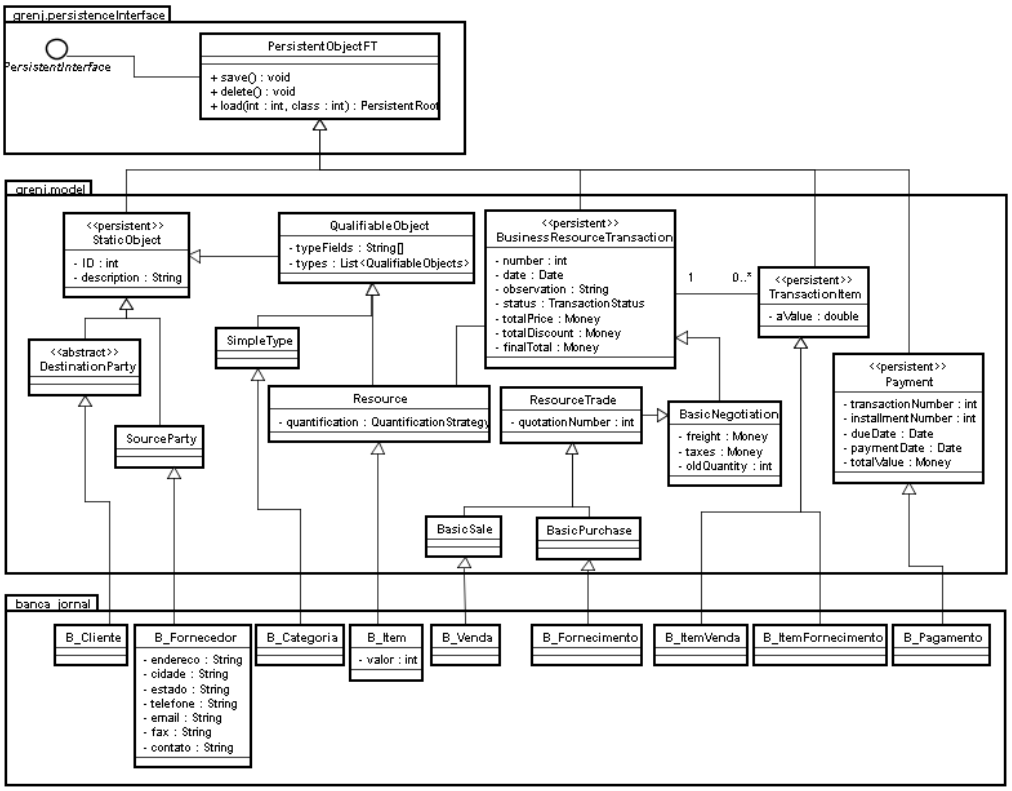


Figura 5.2: Modelo de classes do sistema de vendas com o GRENJ-FT.

Em seguida, foi realizada a implementação das classes identificadas. O processo de criação das classes foi mais simples do que é originalmente no GRENJ, pois bastou que o desenvolvedor se preocupasse com a criação das classes, representando seus atributos e implementando os métodos gancho exigidos pelo framework. Uma vez que o Framework Transversal de Persistência [Camargo e Masiero 2008] está presente na estrutura do GRENJ-FT, não existe mais a necessidade de se implementar os seguintes interesses de persistência nessas classes: o construtor que alimenta o objeto por meio de um `ResultSet`; chamadas de método `setChanged()` dentro dos métodos de escrita da classe; e métodos que informam à camada de persistência os novos campos criados nessa classe para a criação das instruções de operações com o banco de dados. No GRENJ [Viana 2009], esses últimos métodos são necessários na implementação das classes `B_Fornecedor` e `B_Item`, por possuírem atributos declarados em sua estrutura.

Para esse sistema não foi criada uma interface gráfica. A sua funcionalidade foi testada com o auxílio do JUnit, framework Java para auxílio de testes de unidade, que foi utilizado para a criação dos testes no desenvolvimento do GRENJ. Os casos de testes desenvolvidos para a aplicação SCBR foram satisfatórios logo na primeira tentativa, não havendo necessidade de adaptações na estrutura do GRENJ-FT para que o sistema tivesse o comportamento da funcionalidade solicitada pelos requisitos.

Na Figura 5.3 os casos de testes criados para a classe B_Fornecedor que irão verificar a funcionalidade da aplicação podem ser visualizados por meio de um plug-in desenvolvido para a IDE Eclipse. Quando o resultado desses casos de teste são satisfatórios, um ícone com um “v” é acrescentado na lista de casos de teste, com pode-se observar na Figura 5.4 (`B_TestFornecedor`). Na Figura 5.3 encontra-se um trecho de código referente aos testes listados na Figura 5.4 que confrontam as informações esperadas pelas operações com as recebidas pela execução das mesmas.

```
13 public class B_TestFornecedor {
14
16     public void testDefaultInitialization() {
34
35     @Test
36     public void testFindLike() {
37
38         System.out.println("Antes do construtor...");
39         B_Fornecedor fornecedor = new B_Fornecedor();
40         System.out.println("Depouis do construtor...");
41         ResultSet rs = fornecedor.findlike("ID", 1);
42         try {
43             if (rs.next()){
44                 fornecedor = (B_Fornecedor)fornecedor.setDBToObject(rs);
45             }
46         } catch (SQLException e) {
47             // TODO Auto-generated catch block
48             e.printStackTrace();
49         } catch (Exception e){
50             e.printStackTrace();
51         }
52         Assert.assertEquals( 1, fornecedor.getID() );
53         Assert.assertEquals( "Roberval Arantes", fornecedor.getDescription() );
54         Assert.assertEquals( "Rua das Raposas, 456", fornecedor.getEndereco() );
55         Assert.assertEquals( "Sao Carlos", fornecedor.getCidade() );
56         Assert.assertEquals( "SaoPaulo", fornecedor.getEstado() );
57         Assert.assertEquals( "(16) 3372-3864", fornecedor.getTelefone() );
58         Assert.assertEquals( "roberva@gmail.com", fornecedor.getEmail() );
59         Assert.assertEquals( "30755367-4", fornecedor.getFax() );
60     }
61
62     @Test
63     public void testInitializationWithValusFromDB() throws Exception {
64         PersistentObjectFT persistentObjectFT = new PersistentObjectFT();
65         System.out.println("Antes de executar o LOAD");
66         B_Fornecedor fornecedor = (B_Fornecedor) persistentObjectFT.loadObject( 1, B_Fornecedor.class );
67         System.out.println("Depois de executar o LOAD");
68
```

Figura 5.3: Trecho de código para os testes.

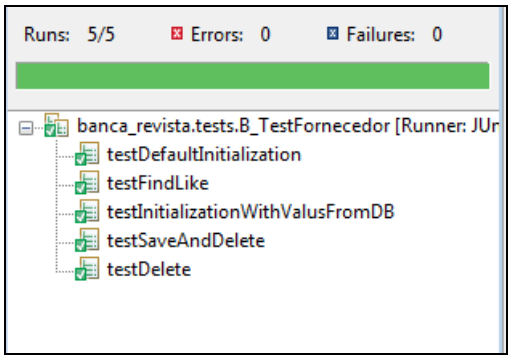


Figura 5.4: Testes para a classe Fornecedor.

5.3 Estudo de caso 2: Sistema para tratamento de aluguéis

Este estudo de caso apresenta a instanciação de um Sistema de Controle de Locação de DVDs (SCLDVDs) que foi originalmente obtido pela instanciação do GRENJ e suas classes foram construídas segundo o processo de instanciação descrito na Seção 3.2.2. Dessa maneira, as classes de negócio da aplicação e as classes de visão criadas para atendê-lo, carregam em sua estrutura espalhamento de código de persistência, devido ao fraco isolamento de interesses de persistência presentes no GRENJ.

O objetivo da realização deste estudo de caso, além de verificar o uso do padrão de locação de recursos oferecido pelo GRENJ em um sistema instanciado pelo GRENJ-FT, é verificar o custo da manutenção necessária para que sistemas criados com o uso

do GRENJ tenham o mesmo funcionamento com o GRENJ-FT. O sistema de aluguel foi construído de acordo com os seguintes requisitos:

1. A locadora realiza o aluguel de DVDs de filmes que podem ter uma ou mais cópias.
2. Cada filme possui um código, título e ano.
3. Cada DVD possui 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 a qual indica o valor diário da locação.
5. Os filmes também são classificados por gênero (comédia, terror, ação, etc.).
6. Os DVDs são alugados para os clientes cadastrados da locadora. As informações que o sistema deve manter sobre o cliente são: código, nome, telefone e CPF.
7. As informações de locação são: código, data de locação, data de devolução prevista, código do cliente, DVDs alugados, data de devolução efetiva e valor. Um cliente pode alugar mais de um DVD em uma mesma locação.
8. Se os DVDs não forem devolvidos na data de devolução prevista, o cliente deve pagar multa correspondente a um valor fixo multiplicado pelos dias de atraso na devolução.

Os padrões da GRN que atendem a esses requisitos são vistos na Figura 5.5. A Tabela 5.2 apresenta a relação dos padrões com os requisitos do SCLDVDs.

Na Figura 5.6 pode-se observar o modelo de classes do SCLDVDs. A camada de visão foi desenvolvida utilizando a biblioteca Swing da linguagem Java e portanto não aparece no modelo. As classes de modelo adicionam campos requeridos pelos requisitos:

- A classe `Cliente` possui as informações de telefone e CPF;
- A classe `Categoria` possui informação de valor;
- A classe `Filme` possui a informação de ano.

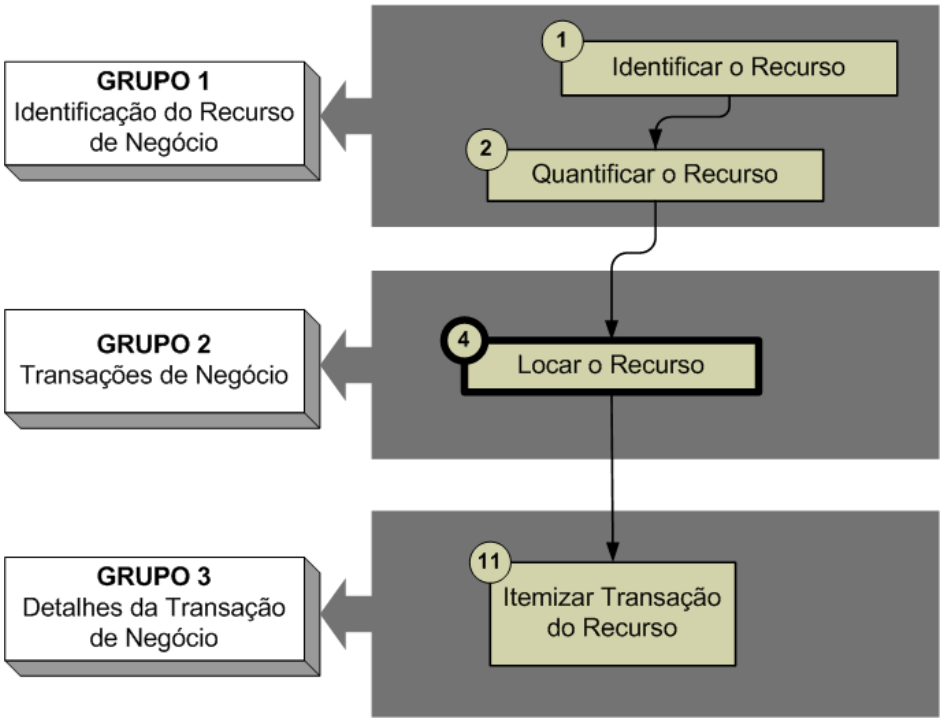


Figura 5.5: Padrões da GRN usados no sistema de controle de aluguel.

Tabela 5.2: Relação dos padrões da GRN com os requisitos do SCLDVDs.

Padrões da GRN	Requisito do SCLDVDs	Justificativa
1	2	identifica o recurso representado por Filme.
2	1, 3, 4 e 5	os detalhes da qualificação e classificação do recurso são realizados.
4	6, 7 e 8	trata do aluguel de DVDs e da análise para o tratamento de clientes, assumindo o papel de entidade de destino da transação.
11	6	Usado para se referir aos detalhamentos do requisito, em que devem ser definidas várias instâncias de produtos diferentes em uma mesma transação.

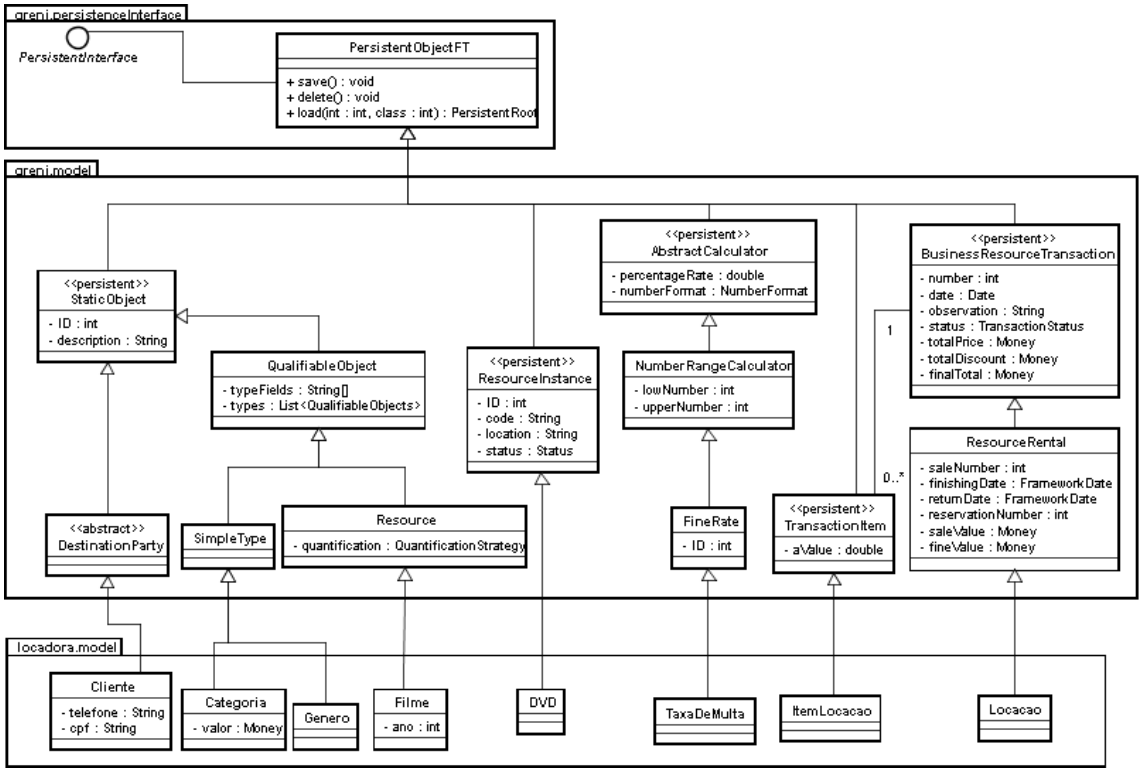


Figura 5.6: Modelo de classes do SCLDVDs.

Uma vez obtido o sistema, o framework GRENJ foi removido da aplicação, sendo acoplado o GRENJ-FT. Com esse processo, alguns erros referentes a métodos não encontrados foram observados nas classes da aplicação. Esses erros representavam o interesse de persistência espalhado pelo código do sistema e foram removidos mediante adaptações de seu código, correspondentes às adaptações de isolamento feitas nas classes de modelo do GRENJ durante a adaptação dos frameworks como apresentado na Seção 4.3. Para as classes de aplicação, foram realizadas as seguintes adaptações:

1. Remoção das chamadas de métodos `setPersisted()` e `setChanged()` de todas as classes de aplicação, e também das classes de visão após as chamadas de métodos `save()`;

2. Remoção das implementações dos métodos `insertionFieldClause()`, `insertionValueClause()` e `updateSetClause()` nas classes `Cliente`, `Categoria` e `Filme`;

3. Remoção da implementação dos construtores que recebem como parâmetros um `ResultSet` e um `Index`, responsáveis por alimentar as informações da classe com base nesse `ResultSet`;

4. Mudança dos parâmetros de entrada dos métodos `set()` de `int` para `Integer`;

5. Tratamento de datas com o tipo `FrameworkDate`.

A Tabela 5.3 mostra de maneira quantitativa o número de métodos removidos das classes do modelo da aplicação. A eliminação de métodos implementados nas classes de modelo (referentes aos itens 2 e 3 citados anteriormente) são representados pela coluna MI. A eliminação de chamadas de métodos, referente ao item 1, é representado pela coluna MC. Também é mostrado na tabela o número de linhas de código das classes, que foi alterado em conseqüência dessas modificações nas classes do sistema.

Tabela 5.3: Relação de métodos implementados e chamadas de métodos de persistência nas classes de modelo da aplicação.

SCLDVD						
Classes	**MI**		**MC**		**LoC**	
	GRENJ	**GRENJ-FT**	**GRENJ**	**GRENJ-FT**	**GRENJ**	**GRENJ-FT**
Category	3	0	2	0	83	59
Client	4	0	4	0	118	85
DVD	0	0	1	1	30	31
Movie	4	0	2	0	121	92
Gender	0	0	0	0	34	34
RentItem	2	0	0	0	52	43
Rent	0	0	0	0	167	170
FineTax	0	0	0	0	18	18
TOTAL	13	0	9	1	623	532

Pode-se observar na tabela que o número de métodos referentes a persistência foi reduzido a zero, não havendo mais a implementação de interesses de persistência nas classes de modelo da aplicação. Também se observa que o número de chamadas de métodos referentes à persistência foi reduzido de maneira expressiva. Com isso o

número de linhas de código dessas classes também foi reduzido, tornando-as mais simples e facilitando sua compreensão.

Quanto às alterações referentes aos itens 4 e 5, na Tabela 5.4 são mostrados o número de linhas de código alteradas para que o sistema pudesse ser utilizado como uma instância do GRENJ-FT. Na coluna LA são mostradas as linhas de código alteradas. Na coluna LoC são mostradas o total de linhas de código da classe, e na coluna % é mostrada a porcentagem que as alterações representam em relação ao total de linhas da classe.

Tabela 5.4: Alterações na camada de modelo do SCLDVD.

SCLDVD			
Classes	**LA**	**LoC**	**%**
Model Classes			
Category	3	59	5,1
Client	0	85	0
DVD	1	31	3,2
Movie	1	92	1,1
Gender	1	34	2,9
RentItem	0	43	0
Rent	3	170	1,8
FineTax	0	18	0
View Classes			
CategoryPanel	11	211	5,2
ClientPanel	12	222	5,4
DVDPanel	29	255	11,4
MoviePanel	26	271	9,6
GenderPanel	13	202	6,4
StandardPanel	3	82	3,6
RentPanel	36	481	7,4
TOTAL	152	2256	

Nessa tabela observa-se que as classes de modelo apresentam poucas mudanças, pois possuem poucas chamadas de métodos de persistência, sendo alteradas somente informações de tipos de dados. Já as classes de visão apresentam um número maior de

alterações porque possuem um número maior de chamadas de métodos de persistência. Essas chamadas devem ser alteradas para que correspondam aos métodos implementados na interface desenvolvida neste trabalho. Pode-se também observar que em nenhum dos dois tipos de classe a porcentagem de linhas de código alteradas foi maior que 11%, o que mostra que apenas uma pequena parte das classes desenvolvidas deve ser alterada para que a adaptação da aplicação seja realizada. Os códigos da classe `Cliente` antes e depois da substituição do framework em sua estrutura podem ser observados, respectivamente, nas Figuras 5.8(a) e (b).

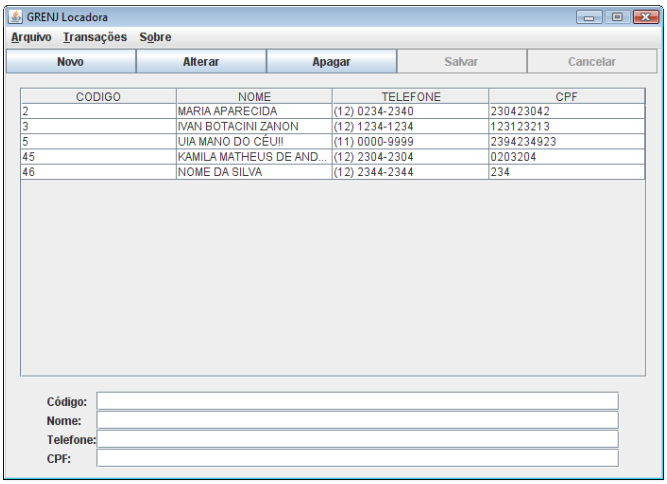
Figura 5.7: Código da classe de Cliente antes e depois das adaptações para o GRENJ-FT.

Os trechos de código que puderam ser removidos das classes de aplicação são os inscritos nos retângulos, Figura 5.8(a). Assim os interesses de persistência da classe de aplicação foram eliminados, permanecendo somente a responsabilidade de tratar as informações de `Cliente`.

Pode-se observar que as adaptações de isolamento (Capítulo 4) realizadas na adaptação do framework foram suficientes para que o sistema passasse a utilizar o GRENJ-FT no lugar do GRENJ. Com isso, pode-se observar que não só o framework utilizado, mas também as classes de aplicação, apresentaram menor entrelaçamento de código.

Duas telas do sistema em funcionamento são apresentadas respectivamente na Figura 5.7(a) e na Figura 5.7(b): o cadastro de clientes e um exemplo de locação com as informações exigidas pelo requisito sendo utilizadas.

(a)



(b)

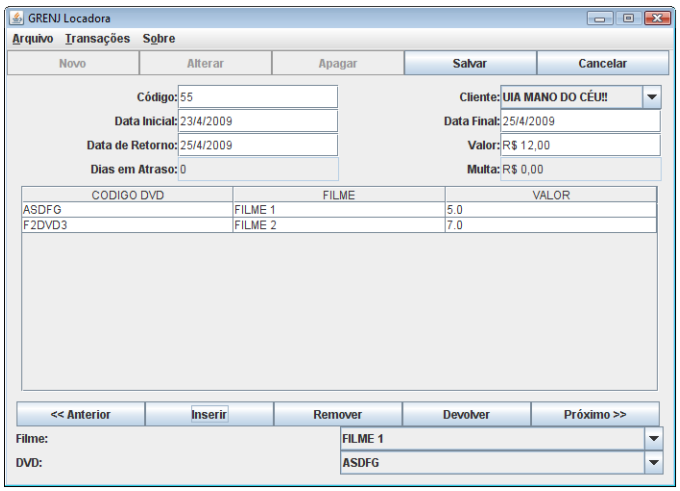


Figura 5.8: Exemplos de telas com o sistema de Locação de DVDs em funcionamento.

5.4 Estudo de caso 3: Sistema para tratamento de manutenção

O terceiro estudo de caso trata de um Sistema Assistência Técnica (SAT) para verificar o comportamento do padrão de manutenção da GRN no GRENJ-FT. As classes desse sistema foram geradas a partir do gerador de sistemas criado no GRENJ (Viana, 2009). Assim como no sistema de aluguel (Seção 5.3), esse sistema sofreu adaptações para funcionar com o GRENJ-FT. Os requisitos para esse sistema foram:

1. O sistema deve permitir a inclusão, alteração e remoção de clientes, com os seguintes atributos: nome, endereço, cidade, estado, telefone, fax, email e documento de identificação.

2. O sistema deve permitir a inclusão, alteração e remoção de aparelhos, com os seguintes atributos: código do aparelho, descrição e fabricante.

3. O sistema deve permitir a inclusão, alteração e remoção das diversas peças utilizadas nos consertos, com os seguintes atributos: código da peça, descrição, fabricante, categoria, valor e quantidade em estoque.

4. O sistema deve permitir a inclusão, alteração e remoção dos diversos tipos de tarefas que podem ser realizados em um conserto, como limpeza interna, troca de peças, etc. Cada tipo de tarefa possui um código, uma descrição e um valor.

5. O sistema deve permitir o processamento do pedido de conserto, com os seguintes atributos: número do conserto, data do pedido, aparelho, descrição.

6. O sistema deve permitir o processamento da execução do conserto, após a verificação do aparelho, com os seguintes atributos: número do conserto, aparelho, tipos de conserto necessários, peças necessárias, data prevista para finalização, desconto (se houver), valor total, descrição dos defeitos apresentados pelo aparelho e cliente.

7. O sistema deve permitir o processamento da finalização do conserto, após sua execução, com os seguintes atributos: número do conserto, aparelho, tipos de conserto realizados, peças utilizadas, data da finalização, desconto (se houver), valor total e forma de pagamento.

8. O sistema deve permitir as seguintes formas de pagamento: 1) a vista (dinheiro ou débito); 2) a prazo (cheque ou cartão de crédito).

Os padrões identificados para esses requisitos podem ser observados na Figura 5.9. Na Tabela 5.3 podem ser vistos os relacionamentos dos padrões com os requisitos do sistema.

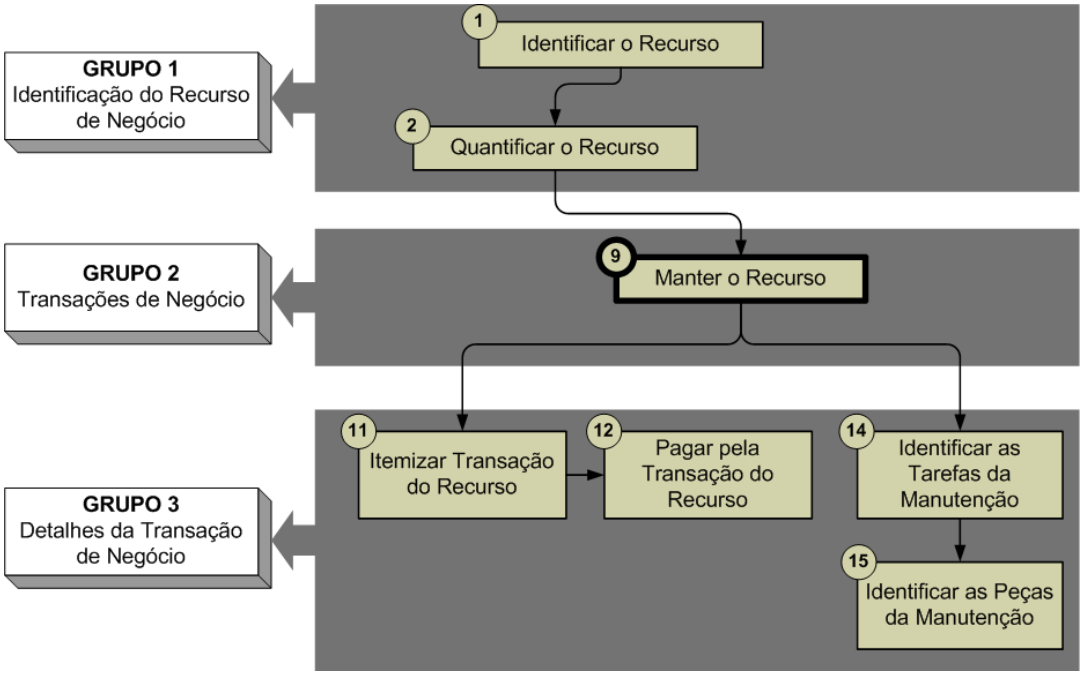


Figura 5.9: Padrões da GRN usados no sistema de controle de manutenção.

Tabela 5.5: Relação dos padrões da GRN com os requisitos do SAT.

Padrões da GRN	Requisito do SAT	Justificativa
1 e 2	2	identifica os produtos de manutenção
9	1, 5, 6 e 7	trata dos detalhes da manutenção do recurso
12	8	trata do controle do pagamento da manutenção após o seu término
14	4	identifica os serviços de manutenção prestados
15	3	tratando das peças envolvidas na manutenção efetuada

Os modelos de classes gerados a partir dos padrões de análise podem ser vistos na Figura 5.10.

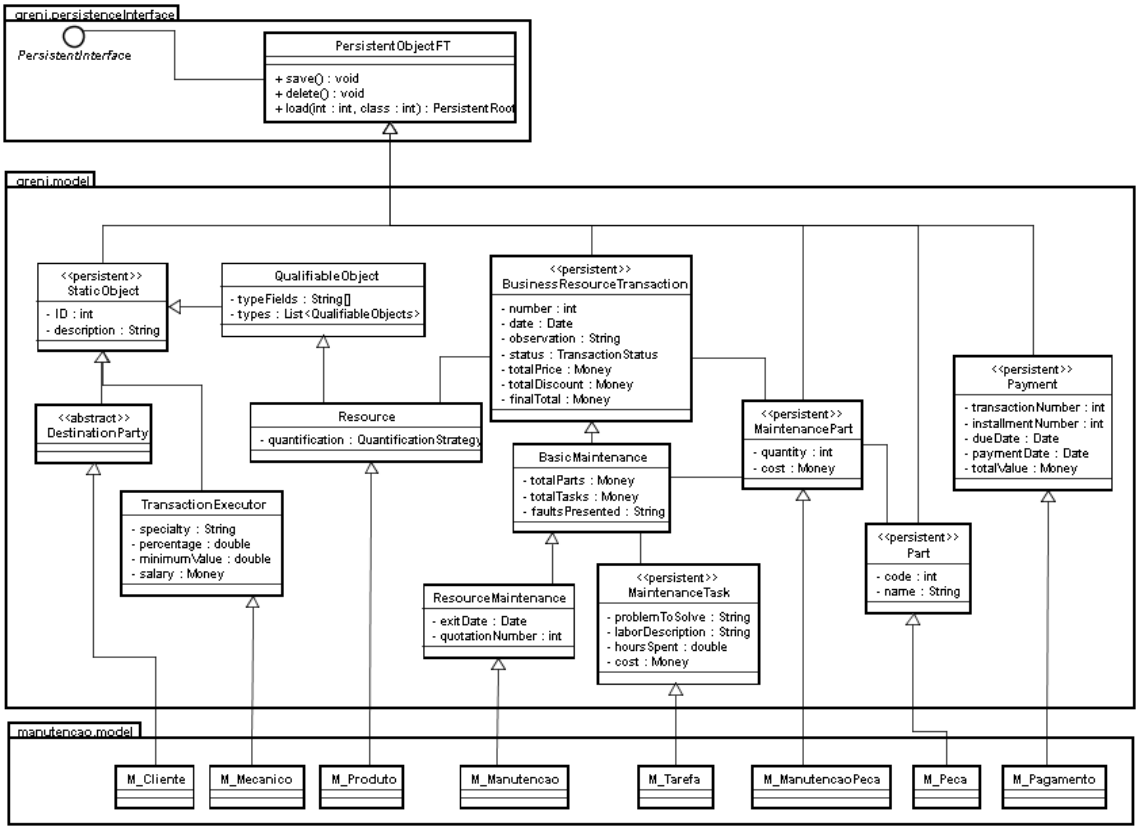


Figura 5.10: Diagrama de classes do sistema de manutenção com o GRENJ-FT.

As classes apresentadas na Figura 5.10 foram obtidas com o uso do gerador de aplicações desenvolvido por Viana (2009). Neste gerador, foram informados os padrões identificados, bem como os detalhes dos campos exigidos pelos requisitos, e com isso foram geradas classes para a aplicação de manutenção. Pôde-se observar que as modificações necessárias para o funcionamento desse sistema foram as mesmas apresentadas na Seção 5.3. A partir da remoção dos códigos referentes ao interesse transversal de persistência, presentes nas classes de aplicação do sistema de manutenção, essas já podem ser usadas para a execução das tarefas de acordo com o solicitados nos requisitos informados nesta seção.

Como o framework GRENJ-FT foi desenvolvido sem a camada de interface existente no GRENJ (Viana, 2009), as funcionalidades das classes geradas foram testadas com o uso de casos de testes gerados com o auxílio do framework JUnit e apresentaram sucesso em todos os casos.

5.5 Considerações Finais

Com os estudos de casos para as instanciações de sistemas apresentadas nesse capítulo foi possível verificar que as aplicações que utilizam os principais padrões da GRN podem ser instanciadas a partir do framework GRENJ-FT. Nesses casos obteve-se sucesso no atendimento dos requisitos que foram modelados pelos padrões da GRN e também foi mantido o comportamento de persistência esperado.

O processo de instanciação utilizando o GRENJ-FT permaneceu inalterado em relação ao do GRENJ. Todos os passos efetuados para a criação de aplicações com o GRENJ foram reproduzidos para as aplicações com o GRENJ-FT, gerando sistemas com a mesma funcionalidade. Logo, a camada de persistência acoplada ao GRENJ não interferiu no processo de instanciação e engenheiros de software acostumados ao uso desse framework não terão dificuldades em utilizar a versão GRENJ-FT gerada com este trabalho.

Também pôde-se observar que aplicações criadas com o uso do GRENJ podem ser adaptadas para o GRENJ-FT, mantendo-se a mesma a mesma funcionalidade. Essas adaptações apresentam um custo de manutenção baixo, pois somente são necessárias adaptações de isolamento, ou seja, a remoção da implementação do interesse de persistência que se encontra espalhado em seu código. A adaptação dos sistemas baseados no GRENJ para o GRENJ-FT proporciona melhor isolamento do interesse de persistência no código desses sistemas.

O próximo capítulo apresentará as conclusões deste trabalho, bem como as suas limitações e sugestões para trabalhos futuros.

CAPÍTULO 6
Conclusão

6.1 Considerações Finais

Neste trabalho de mestrado foi apresentada a adaptação realizada no framework de aplicação GRENJ [Durelli 2008], na qual a camada de persistência, implementada segundo o padrão de projeto *Persistence Layer* presente em sua estrutura, foi substituída por um framework transversal de persistência [Camargo e Masiero 2008]. O resultado dessa adaptação é um novo framework, chamado de GRENJ-FT. É possível observar que o GRENJ-FT possui melhor isolamento do interesse de persistência, eliminando a presença de códigos desse interesse na camada de negócios. Assim pode ser garantida melhor legibilidade ao seu código, o que facilita o processo de entendimento do framework durante a instanciação de novas aplicações, e também diminui o custo de sua manutenção. Além disso, as aplicações instanciadas a partir desse framework também deixam de ter espalhamento de código em suas próprias classes. Os desenvolvedores que utilizarem o GRENJ-FT para instanciar aplicações não terão que se preocupar com interesses de persistência na criação de sistemas específicos do domínio atendido pelo framework, o que diminui o custo dessa atividade.

Com este trabalho foi também possível observar a experiência do acoplamento de um framework transversal em uma estrutura genérica para o tratamento do interesse de persistência. Embora o acoplamento das estruturas tenha sido satisfatório, algumas

adaptações foram necessárias na estrutura dos dois frameworks, para que fosse realizado de maneira adequada. Essas adaptações tiveram por objetivo integrar os dois frameworks, sem alterar o escopo de atuação deles e sem torná-los dependentes um do outro. Assim, os dois frameworks continuam sendo genéricos: o FT de Persistência pode ser acoplado a qualquer outro código-base e o GRENJ pode receber qualquer tipo de mecanismo de persistência sem mudanças invasivas em seu código.

Os estudos de caso desenvolvidos mostraram que o framework GRENJ-FT utiliza o mesmo processo de instanciação e o mesmo escopo de atuação que o GRENJ. Além disso, foi verificado que sistemas já instanciados com o GRENJ podem também ser adaptados, passando a usar o GRENJ-FT. Essa adaptação não é significativa e proporciona a essas aplicações maior isolamento do interesse de persistência, com código mais legível e mais flexível a futuras manutenções.

6.2 Contribuições

O isolamento de interesses em estruturas de software proporciona melhor legibilidade ao código do framework, facilita a compreensão de sua estrutura e diminui o custo de sua manutenibilidade.

O GRENJ-FT, obtido por meio das modificações descritas neste trabalho, apresenta-se como um aperfeiçoamento do framework de aplicação GRENJ. Neste sentido, há diminuição do entrelaçamento do interesse de persistência com o interesse de negócios do framework de aplicação, realizando de maneira adequada o isolamento desses interesses. Logo a legibilidade do framework GRENJ-FT mostra-se melhor que a do GRENJ, facilitando a realização de futuras manutenções na estrutura do framework.

O processo de instanciação de novas aplicações com o uso do GRENJ-FT é facilitado, pois as classes de aplicação passam a não ter mais que implementar interesses de persistência, o que diminui o esforço por parte do engenheiro de software.

As adaptações realizadas no GRENJ para que o acoplamento com o FT fosse possível manteve inalterado o seu escopo de atuação, porém essas adaptações facilitam o acoplamento de outros FTs à sua estrutura. Outro aperfeiçoamento conseguido com essa manutenção é a possibilidade de se utilizar outros bancos de dados suportados pelo FT de Persistência, além do MySQL originalmente suportado pelo GRENJ.

O uso de um FT de Persistência no isolamento da persistência do GRENJ mostrou-se útil, pois possibilitou o reúso do conhecimento para o tratamento desse interesse, tornando a atividade de isolamento menor do que se fosse realizada com a construção de aspectos específicos para o GRENJ. Embora as adaptações realizadas tenham exigido conhecimento aprofundado do FT, sua utilização facilitou o processo de isolamento do interesse de persistência.

Com este trabalho também foi observado que é possível realizar o isolamento dos interesses de um framework de aplicação com o apoio de frameworks transversais, desde que algumas adaptações sejam feitas. Além de esse acoplamento ser possível, vê-se que a instanciação ou o escopo de atuação do framework de aplicação não precisa ser alterado por conta desse acoplamento.

A estrutura do Framework de Aplicação possuía características que impossibilitavam o seu acoplamento direto ao FT de Persistência, fazendo-se necessárias manutenções para a realização desse acoplamento. Já quanto ao FT, as alterações foram referentes somente ao acréscimo de funcionalidade para atender aos requisitos do código ao qual ele se acoplava. Ou seja, não foi identificado problema estrutural que o impedisse de se acoplar ao GRENJ. Logo, as adaptações realizadas no FT de Persistência se caracterizam como melhorias em sua estrutura. Essas adaptações permitem que esse framework se acople a um número maior de códigos-base, podendo tratar novos tipos de dados e já possuindo nativamente métodos prontos para atender a problemas específicos de alguns ambientes.

Por fim, o processo utilizado na realização dessa manutenção se mostra útil para futuras manutenções em frameworks de aplicação, uma vez que uma característica comum entre esses frameworks é a sua representação como um conjunto de árvores de classes.

6.3 Limitações

A criação do GRENJ-FT possibilitou a melhoria da legibilidade das classes e também das classes de aplicações instanciadas com o seu uso. Porém algumas limitações puderam ser observadas no trabalho realizado.

Embora a experiência de acoplamento do framework de aplicação com o framework transversal tenha apresentado sucesso, não é possível generalizar esse acoplamento, ou seja, garantir que ele é possível com a utilização de qualquer estrutura. No caso da adaptação realizada, foi apresentado o acoplamento de somente um framework transversal à estrutura de um framework de aplicação. Não se sabe qual o comportamento do framework transversal aqui apresentado se acoplado a algum outro framework de aplicação que apresente entrelaçamento do interesse de persistência. Tampouco pode-se ter certeza se as manutenções necessárias para esse acoplamento serão as mesmas aqui relatadas.

Em relação às melhorias alcançadas na estrutura do GRENJ, é importante notar também que, mesmo que as classes do GRENJ-FT apresentem uma diminuição no número de linhas de código, houve um aumento no número de classes existentes nesse framework, dada a presença do framework transversal em sua estrutura. A camada de persistência do GRENJ é composta por três classes principais que realizam todo o trabalho de persistência, e mais duas classes auxiliares. Já a estrutura do FT de Persistência é mais complexa, contando com quinze classes, mais quatro classes desenvolvidas para o acoplamento do FT com o código-base. Embora esses números não influenciem diretamente o desempenho dos frameworks ou mesmo na facilidade de sua compreensão, há aumento do tamanho físico do sistema, bem como das aplicações instanciadas a partir do framework.

Outra limitação deste trabalho está no fato de que a camada intermediária criada para o tratamento de alguns requisitos de persistência do GRENJ é específica para os dois frameworks utilizados, podendo somente ser utilizada para a integração dessas duas estruturas. O ideal seria que essa camada fosse generalizada para que pudesse atender a qualquer framework de aplicação que fosse acoplado a um framework transversal.

6.4 Trabalhos Futuros

De modo a dar continuidade a este trabalho, são sugeridos os seguintes trabalhos:

- O acoplamento de mais frameworks transversais à estrutura de um framework de aplicação, afim de verificar quais serão as alterações necessárias, além de observar o comportamento de mais de um FT em uma estrutura genérica;

- O acoplamento do FT de Persistência a mais frameworks de aplicação, verificando assim se as melhorias aqui realizadas em sua estrutura permitem que sejam atendidas devidamente às exigências de frameworks de aplicação em geral. Também pode-se observar se as alterações realizadas no seu acoplamento com outros frameworks são semelhantes às realizadas neste projeto;

- A utilização do GRENJ-FT juntamente com a camada de visão, desenvolvida para o GRENJ por Viana (2009). Como apresentado no Capítulo 3, a versão utilizada neste projeto possui somente as camadas de modelo e de persistência do framework. Viana (2009) se baseou neste mesmo framework para criar, em paralelo a este projeto, a camada de interface gráfica e o wizard para instanciação de aplicações. Logo, a versão do GRENJ que conta com a camada de visão ainda possui entrelaçamento de código de persistência assim como a versão utilizada neste trabalho, e pode ter o seu isolamento realizado caso seja integrado ao GRENJ-FT.

- A generalização da camada intermediária utilizada para realizar o tratamento de interesses de persistência específicos do GRENJ. Essa camada foi criada para que alguns comportamentos específicos da persistência do GRENJ fossem realizados sem que os frameworks se tornassem dependentes um do outro (Seção 4.3). Caso essa camada seja generalizada de modo a atender a quaisquer frameworks, ela pode ser utilizada para possibilitar o acoplamento de frameworks que possuem estruturas semelhantes ao GRENJ com FTs de Persistência.

- O desenvolvimento de um módulo responsável pela criação automática das tabelas utilizadas pelo FT de Persistência. Essas tabelas devem ser criadas pelo engenheiro de aplicação, fazendo corretamente as correspondências das estruturas das classes com as tabelas criadas. Uma vez que o FT tenha acesso às informações da estrutura do código-base que está entrecortando, é possível que ele as use para criar automaticamente a estrutura do bando de dados utilizado, diminuindo o esforço por parte do desenvolvedor.

- Por fim, espera-se que este trabalho possa auxiliar à criação de uma arquitetura de referência para a criação de frameworks de aplicação de forma a facilitar o

acoplamento de frameworks transversais em sua estrutura. O desenvolvimento de frameworks transversais ainda é uma área de pesquisa recente na Engenharia de Software, e FTs para o tratamento de outros tipos de interesses transversais podem ser desenvolvidos futuramente, podendo estes também ser integrados à estrutura de frameworks de aplicação para o tratamento de interesses transversais.

Referências Bibliográficas

AL-MANSARI, Mohammed; HANENBERG, Stefan; UNLAND, Rainer. Orthogonal persistence and AOP: a balancing act. **ACP4IS '07: Proceedings of the 6th Workshop on Aspects, Components, and Patterns For Infrastructure Software**, New York, NY, USA, p. 1-2. 12 mar. 2007.

BECK, Kent. Aim, fire [test-first coding]. **IEEE: Software**, Los Alamitos, California, v. 18, n. 5, p.87-89, Oct. 2001.

BENNETT, Keith H.; RAJLICH, Václav T.. Software maintenance and evolution: a roadmap. **International Conference On Software Engineering: Proceedings of the Conference on The Future of Software Engineering**, New York, NY, USA, p. 72-87. 2000.

BRAGA, Rosana Terezinha Vaccare. **Um processo para construção e instanciação de frameworks baseados em uma linguagem de padrões para um domínio específico.** 2003. 172 f. Tese (Doutorado) - ICMC, USP, São Carlos, 2003.

BRUGALI, Davide; SYCARA, Katia. Frameworks and pattern languages: an intriguing relationship. **ACM Computing Survey**, New York, NY, USA, v. 32, n. 1, p.2. 2000.

BUSCHMANN, Frank; HENNEY, Kevlyn.; SCHMIDT, Douglas C.. Past, Present, and Future Trends in Software Patterns. **IEEE: Software**, Los Alamitos, California, USA, v. 24, n. 4, p.31-37, Jul. 2007.

CAMARGO, V. V.; MASIERO, P. C. Frameworks Orientados a Aspectos**. Simpósio Brasileiro de Engenharia de Software**. Rio de Janeiro: Arndt von Staa/Pontifícia Universidade Católica, 2005.

CAMARGO, V. V.; MASIERO, P. C. A pattern to design crosscutting frameworks. **SAC '08: Proceedings of the 2008 ACM symposium on Applied computing**, p. 759-764, 2008.

CAO, Yan; YANG, Lina; YANG, Yanli; CHEN, Hua; LIU, Ning. Machine Tool Distributed Cooperative Design System Based on Extended MVC-Based Web

Application Framework and XML Interoperable Information Model. **ICICSE ’08: International Conference on Internet Computing in Science and Engineering**, p. 423-428, 28-29 Jan. 2008

CHOU, Li-Ping; SUN, Jenn-Dong; CHEN Mei-Ling. A New Application Framework for Intelligent Surveillance Sensor Networks. **IIHMSP 2007: Third International Conference on Intelligent Information Hiding and Multimedia Signal Processing**, Splendor Kaohsiung, Taiwan, v. 1, p. 589-591. 26-28 Nov. 2007.

CHUNG, Mokdong; CHOI, Jaehyuk; LEE, Kiyeal; RHYOO, Shi-Kook. Constructing Enterprise Application Framework for Secure RFID Application Using SPKI/SDSI. **ALPIT 2007: Sixth International Conference on Advanced Language Processing and Web Information Technology**, Henan, China, p. 572-577. 22-24 Aug. 2004.

CONSTANTINIDES, Constantinos A.; BADER, Atef; ELRAD, Tzilla H.; NETINANT, P.; FAYAD, Mohamed E.. Designing an aspect-oriented framework in an object-oriented environment. **ACM Computing Survey**, New York, NY, USA, v. 32, n. 1, p.41. Mar. 2000.

COPLIEN, James O. **Software Patterns**. New York, NY, USA: Sigs Books & Multimedia, 1996.

CORTES, Mariela; FONTOURA, Marcus; LUCENA, Carlos. Using refactoring and unification rules to assist framework evolution. **ACM SIGSOFT Software Engineering Notes**, New York, NY, USA, v. 28, n. 6, p.1-1, nov. 2003.

DAGENAIS, Barthélémy; ROBILLARD, Martin P. Recommending adaptive changes for framework evolution. **30th International Conference On Software Engineering**, Leipzig, Germany, p.481-490, 2008.

DEMEYER, S.; DUCASSE, S.; NEBBE, R.; NIERSTRASZ, O.; RICHNERL, T. Using Restructuring Transformations to Reengineer Object-Oriented Systems. **Software Composition Group**, University of Berne, 2000.

DIAZ, M.; ROMERO, S.; RUBIO, B.; Soler, E.; Troya, J. M. An aspect oriented framework for scientific component development. **PDP 2005: 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing**, Lugano, Switzerland, p. 290-296. fev. 2005.

Durelli, V. H. S. **Reengenharia Iterativa do Framework GREN**. 2008. Dissertação (Mestrado) – DC, UFSCar, São Carlos, 2008

FAYAD, Mohamed; SCHMIDT, Douglas C. Object-oriented application frameworks. **Communications Of The ACM**, New York, NY, USA, v. 40, n. 10, p.32-38, 1997.

FRAKES, W. B.; KANG, Kyo. Software reuse research: status and future. **IEEE Transactions On Software Engineering**, New York, NY, USA, v. 31, n. 7, p.529-536, jul. 2005.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Design patterns:** elements of reusable object-oriented software. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 1995.

GOMAA, Hassan. **Designing software product lines with UML**: from use cases to pattern-based software architectures. Redwood City, CA, USA: Addison Wesley Longman Publishing Co. Inc., 2004.

INTAKOSUM, S. From Use Cases to Framelets for Building Application Frameworks. **ITNG 2008: Fifth International Conference on Information Technology: New Generations**, Las Vegas, Nevada, USA, p. 1259-1260. apr. 2008.

JANZEN, D.; SAIEDIAN, H. Test-driven development concepts, taxonomy, and future direction. **Computer**, New York, NY, USA, v. 38, n. 09, p.43-50, sep. 2005.

JEFFRIES, Ron; MELNIK, Grigori. Guest Editors' Introduction: TDD- The Art of Fearless Programming. **IEEE: Software**, Los Alamitos, California, USA, v. 24, n. 3, p.24-30, may 2007.

JOHNSON, Ralph E. Frameworks = (components + patterns). **Communications Of The ACM**, New York, NY, USA, v. 40, n. 10, p.39-42, out. 1997.

KICZALES, Gregor; LAMPING, John; MENDHEKAR, Anurag; MAEDA, Chris; LOPES, Cristina Videira; LOINGTIER, Jean-Marc; IRWIN, John. Aspect-Oriented Programming. **ECOOP'97: 11th European Conference of Object-Oriented Programming**, Jyväskylä, Finland, v. 1241, p. 220-242. jun. 1997.

KIRCHER, M.; VOLTER, M. Guest Editors' Introduction: Software Patterns. **IEEE: Software**, Los Alamitos, California, USA, v. 24, n. 4, p.28-30, jul. 2007.

KLUMP, R. Understanding object-oriented programming concepts. **IEEE Power Engineering Society Summer Meeting**, Los Alamitos, California, USA, v.2, p. 1070-1074. jul. 2001.

KRUEGER, Charles W. Software reuse. **ACM Computing Surveys**, New York, Ny, Usa, v. 24, n. 2, p.131-183, jun. 1992.

KULOOR, C.; EBERLEIN, A. Aspect-oriented requirements engineering for software product lines. **10th IEEE International Conference And Workshop On The Engineering Of Computer-based Systems**, Huntsville, Alabama, USA, p. 98-107. apr. 2003.

LADDAD, Ranmivas. Aspect-oriented programming will improve quality. **IEEE: Software**, Los Alamitos, California, Usa, v. 20, n. 6, p.90-91, nov. 2003.

LI, Liu; QIANG, Tao. Research of JFSSH application framework based on J2EE. **CCDC 2008: Chinese Control and Decision Conference**, Yantai, China, p. 688-692. jul. 2008.

LIAO, Weidong; KOONSE, B. J. A layered Java application framework for supplying mathematical computing power to the distributed environment. **SEW 2007: 31st IEEE Software Engineering Workshop**, Columbia, MD, USA, p. 279-283. mar. 2007.

LIU, Yong; YANG, Aiguang. Research and application of software-reuse. **SNPD 2007: Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing**, Qingdao, China, p. 588-593. jul. 2007.

LOBATO, Cidiane; GARCIA, Alessandro; KULESZA, Uirá; VON STAA, Arndt; LUCENA, Carlos. Evolving and composing frameworks with aspects: the MobiGrid case. **ICCBSS 2008: Seventh International Conference on Composition-Based Software Systems**, Madrid, Spain, p. 53-62. fev. 2008.

LOPES, Sergio; TAVARES, Adriano; MONTEIRO, João; SILVA, Carlos. Design and description of a classification system framework for easier reuse. **ECBS '07: 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems**, Tucson, Arizona, USA, p. 71-82. mar. 2007.

MAT JANI, H.; LEE, S. P. Applying case reuse and rule-based reasoning (RBR) in object-oriented application framework documentation: Analysis and design. **Conference On Human System Interactions**, Krakow, Poland, p. 597-602. 25 may 2008.

PACIOS, S. F. **Uma abordagem orientada a aspectos para desenvolvimentos de linhas de produtos de software**. Monografia de qualificação, ICMC-USP, São Carlos – SP, 2007.

PARK, Jaeyong. **Perfective and corrective UML pattern-based design maintenance with design constraints for information systems**. 2007. 229 f. Tese (Doutorado) - George Mason University, Fairfax, VA, USA, 2007.

PARSONS, David; RASHID, Awais; SPECK, Andreas; TELEA, Alexandru. A "Framework" for object oriented frameworks design. **TOOLS: Technology of Object-Oriented Languages and Systems**, 1999.

POHL, Klaus; BÖCKLE, Günter; VAN DER LINDEN, Frank. **Software Product Line Engineering**: Foundations, Principles, and Techniques. New York, NY, USA: Springer-Verlag Berlin Heidelberg, 2005.

PRESSMANN, Roger S. **Engenharia de Software**. 6. ed. São Paulo, Brasil: McGraw-Hill, 2006. 752 p.

RASHID, Awais; CHITCHYAN, Ruzanna. Persistence as an aspect. **AOSD '03: Proceedings of the 2nd international conference on aspect-oriented software development**, New York, NY, USA, p. 120-129. 2003.

ROTHENBERGER, M. A.; DOOLEY, K. J.; KULKARNI, U. R.; NADA, N. Strategies for software reuse: a principal component analysis of reuse practices. **IEEE Transactions On Software Engineering**, Los Alamitos, CA, USA, v. 29, n. 9, p.825-837, set. 2003.

SAKOWICZ, B.; MURLEWSKI, J.; LABUS, A; Napieralski, A. JWay - Model-Driven J2EE Application Framework. **MIXDES'07: 14th International Conference on Mixed Design of Integrated Circuits and Systems**, p. 703-706. jun. 2007.

SCHMIDT, Douglas C.; FAYAD, Mohamed; JOHNSON, Ralph E. Software patterns. **Communications Of The Acm**, New York, NY, USA, v. 39, n. 10, p.37-39, 1996.

SHIVA, Sajjan G.; SHALA, Lubna Abou. Software reuse: Research and practice. **ITNG'07: Fourth International Conference on Information Technology**, Las Vegas, Nevada, USA, p. 603-609. abr. 2007.

VANHAUTE, B.; WIN, B. D.; DECKER, B. D. Building Frameworks in AspectJ. **ECOOP 2001: Workshop on Advanced Separation of Concerns**, Budapest, Hungry, 2001.

YODER, J.; JOHNSON, R.; WILSON, Q. Connecting Business Objects to Relational Databases. **Proceedings of the 5th Conference on the Pattern Languages of Programs**, 1998.

YUE, Dianmin; WU, Xiaodan; BAI, Junbo. RFID Application Framework for pharmaceutical supply chain. **IEEE/SOLI 2008: IEEE International Conference on Service Operations and Logistics, and Informatics**, Beijing, China, p. 1125-1130. out. 2008.

ZIADI, Tewfik; JÉZÉQUEL, Jean-marc; FONDEMENT, Frédéric. Product line derivation with UML. **Proceeding Software Variability Management Workshop**, University Of Groningen Department Of Mathematics And Computing Science, p. 94-102. 2003.

ZIMMERMANN, Olaf; ZDUN, Uwe; GSCHWIND, Thomas. Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method. **WICSA 2008: Seventh Working IEEE/IFIP**

Conference on Software Architecture, Vancouver, BC, Canada, p. 157-166. fev. 2008.