Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um Arcabouço Orientado por Aspectos Para Implementação Automatizada de Persistência

César Francisco de Moura Couto
Orientador: Roberto da Silva Bigonha
Co-Orientador: Marco Túlio de Oliveira Valente

Belo Horizonte

23 de Junho de 2006

César Francisco de Moura Couto

Um Arcabouço Orientado por Aspectos Para Implementação Automatizada de Persistência

Dissertação de Mestrado apresentada ao Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte

23 de Junho de 2006

Agradecimentos

Agradeço ao Departamento de Ciência da Computação da Universidade Federal de Minas Gerais a oportunidade de participar do seu programa de mestrado.

Aos professores Carlos Camarão, Mariza Bigonha, Nívio Ziviane e Wagner Meira os ensinamentos nas matérias que tive oportunidade de cursar.

Aos amigos, Ademir de Alvarenga Oliveira, Eduardo Santos Cordeiro, Fábio Tirelo, Italo Giovani Abdanur Stefani, João Rafael Moraes Nicola, Kristian Magnani dos Santos, Leonardo Teixeira Passos, Mário Celso Candian Lobato, Rodrigo Antonio de Paiva, Tays Cristina do Amaral Pales Soares, Thiago Henrique Braga e Wagner Salazar Pires a ajuda e companheirismo.

Ao meu orientador Roberto da Silva Bigonha os ensinamentos, atenção, disponibilidade, paciência e puxões de orelha.

Ao meu co-orientador Marco Túlio de Oliveira Valente os quase quatro anos de ensinamentos. Sem ele o meu futuro profissional não seria o mesmo. Obrigado mesmo professor Marco Túlio.

Ao pessoal da Paiva Piovesan e Atan Sistemas que me apoiaram sempre.

Agradeço aos meus irmãos Mila, Carol e Celinho, amigos e familiares o incentivo.

Agradeço à minha linda e doce Cinthia, que teve paciência nos momentos difíceis e que sempre esteve ao meu lado.

Finalmente agradeço a meu querido pai, Célio César, e a minha querida mãe, Cecília, que sempre confiaram em mim.

Resumo

Esta dissertação apresenta um arcabouço orientado por aspectos para a implementação automatizada de persistência. A implementação de persistência em bancos de dados relacionais pode ser dividida nos seguintes sub-interesses: controle de conexões, controle de transações, controle de sincronização de objetos e recuperação de dados. O arcabouço proposto inclui uma ferramenta capaz de gerar aspectos para implementar cada um desses sub-interesses.

Abstract

This thesis presents an aspect oriented framework to support automated implementation of data persistence. The implementation of persistence in relational databases can be classified in the following concerns: connection control, transaction control, object synchronization control and data retrieval. The proposed framework includes a tool that generates aspects for each of these concerns.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O desenvolvimento orientado por objetos parte do princípio que cada objeto deve permitir a encapsulação de dados e métodos com o objetivo de implementar determinados requisitos (*concerns*). Entretanto, na maioria dos sistemas, existem determinados requisitos que devem estar implementados em diversos componentes do mesmo. Estes requisitos são de difícil modularização e são conhecidos como requisitos transversais (*crosscuting concerns*) [29, 51, 27]. Exemplos clássicos de requisitos transversais são *logging*, autenticação, persistência de dados, controle de transação, dentre outros.

A implementação de requisitos transversais em linguagens orientadas por objetos leva a duas condições adversas conhecidas como dispersão de código (*code scattering*) e entrelaçamento de código (*tangled code*) [29, 51, 30]. A dispersão de código se caracteriza quando o código referente a um requisito encontra-se espalhado por vários componentes do sistema. O entrelaçamento de código se caracteriza quando códigos referentes a diversos requisitos localizam-se na mesma unidade funcional.

As implicações destes problemas são diversas. Primeiramente, observa-se que esses problemas dificultam o rastreamento do programa, pois a implementação simultânea de vários requisitos em um único módulo torna obscura a correspondência entre os requisitos e suas implementações, resultando em um mapeamento pobre entre os dois. Além disso, outro ponto importante é que este modelo possui baixo grau de reúso, pois, dado que um único módulo implementa vários requisitos, outros sistemas que precisarem de implementar funcionalidades semelhantes podem não ser capazes de utilizar prontamente o módulo, diminuindo a produtividade do desenvolvedor.

Por estes pontos, percebe-se ainda que o código pode apresentar pouca qualidade interna, com baixa coesão modular e alto grau de acoplamento de módulos [38]. Todos estes problemas podem levar à produção de código de difícil evolução ou baixa extensibili-

dade [37], visto que modificações posteriores no sistema podem ser dificultadas pela pouca modularização.

Dentre os requisitos que claramente possuem um comportamento transversal, destaca-se o suporte a persistência de dados. O requisito de persistência pode ser dividido em sub-requisitos, tais como controle de conexões, controle de transações, controle de sincronização de objetos e recuperação de dados. Esses sub-requisitos quando mapeados para módulos de implementação de um sistema, certamente ocasionam problemas de dispersão e entrelaçamento de código. Para mostrar o quanto a persistência se dispersa e entrelaça no código funcional de um sistema, observe a classe com suporte a persistência, conforme a Listagem 1.1. Essa classe possuirá métodos para criação, atualização e remoção de dados do dispositivo de persistência (linhas 19 a 21). Logo, esses métodos necessitarão de controle de conexão e transação. Além disso, essa classe deverá possuir métodos para recuperação de dados, como `findPrimaryKey`, `findAll`, dentre outros (linhas 24 a 26). A implementação desses métodos torna-se trabalhosa quando feita da forma convencional, ou seja, quando é necessário implementar diretamente classes e métodos responsáveis por essa recuperação.

A grande vantagem de se separar o requisito de persistência do restante do código é permitir a sua modularização. Métodos para criação, atualização e remoção de dados poderão ser implementados e invocados fora de módulos funcionais do sistema. Assim, os problemas de dispersão e entrelaçamento de código descritos anteriormente podem ser minimizados. Além disso, o desenvolvedor passa a não se preocupar com a implementação de vários requisitos ao mesmo tempo, podendo se concentrar na implementação dos módulos de requisitos funcionais. O resultado é um aumento de produtividade do desenvolvedor e diminuição de erros na lógica dos requisitos funcionais.

Assim, não surpreende que a implementação de persistência seja um exemplo clássico de emprego de orientação por aspectos [14, 43, 49, 48]. Programação Orientada por Aspectos (POA) [29, 51, 30, 28, 23] é um novo paradigma de programação que tem por objetivo modularizar requisitos de um sistema que não podem ser modularizados por meio de programação orientada por objetos. A POA vem para separar os requisitos transversais do sistema dos requisitos não-transversais, melhorando, ou até mesmo eliminando, a dispersão e o entrelaçamento de código.

De acordo com Johnson [26], um arcabouço (*framework*) é um projeto reutilizável das partes de um sistema representado por um conjunto de classes abstratas e a maneira que instâncias dessas classes interagem entre si. Em outras palavras, é uma aplicação que pode ser especializada para produzir outros aplicativos. O seu propósito é fornecer recursos para auxiliar um desenvolvedor de sistemas na implementação de aplicações. Além disso, com o aparecimento da programação orientada por aspectos, surgiram novos conceitos e mecanismos que podem ser utilizados no desenvolvimento de arcabouços. Vários autores

```
1  public class Cliente implements Persistent{
2
3      private String nome;
4      private String cpf;
5      private String email;
6      public Cliente(..){..}
7
8      /* contrutora responsavel pela recuperacao de dados */
9      public Cliente(ResultSet rs){..}
10
11     public String getNome(){..}
12     public void setNome(String nome){..}
13     public String getCpf(){..}
14     public void setCpf(String cpf){..}
15     public String getEmail(){..}
16     public void setEmail(String email){..}
17
18     /*metodos de criacao, atualizacao e remocao*/
19     public void create(..){..}
20     public void update(..){..}
21     public void delete(..){..}
22
23     /*metodos para recuperacao dos dados*/
24     public Cliente findByPrimaryKey(..){..}
25     public Cliente findByCpf(..){..}
26     public ArrayList findAll(..){..}
27 }
```

Listagem 1.1: Cliente.java

[12, 43, 49, 52] têm usado o conceito de arcabouço orientado por aspectos como sendo um arcabouço orientado por objetos que também usa estruturas de aspectos em sua implementação. Assim, arcabouços aparecem como uma alternativa para implementação de persistência por meio de aspectos em banco de dados relacionais.

No entanto, os arcabouços já propostos para implementação de persistência por meio de aspectos demandam um esforço considerável de programação, exigindo a definição de diversos aspectos e interfaces. A implementação de tais aspectos é normalmente tediosa e, portanto, sujeita a erros. Além disso, existe uma forte dependência entre os aspectos implementados e as classes da aplicação base. Logo, modificações nessas classes freqüentemente implicam em redefinição dos aspectos de persistência. Na prática, tais problemas

podem dificultar a adoção desses arcabouços por parte de desenvolvedores de sistemas de grande porte. Daí a importância de ferramentas para geração de código que auxiliem os usuários a criar os aspectos de persistência específicos de uma determinada aplicação.

Nesta dissertação, apresenta-se um arcabouço de persistência orientado por aspectos. Diferentemente de outras soluções [49, 43, 14], o arcabouço proposto inclui um gerador de aspectos, o qual é responsável por criar de forma automática todos os aspectos responsáveis por persistir objetos de uma aplicação alvo em um meio de armazenamento não-volátil. Para gerar tais aspectos, exige-se apenas que o usuário do Arcabouço informe parâmetros como os seguintes: pontos do sistema onde deve-se estabelecer a conexão, métodos que atualizam o estado de objetos persistentes e o mapeamento de tais objetos para tabelas de um banco de dados relacional. Esses parâmetros são todos informados por meio de uma interface gráfica, dispensando assim o usuário de dominar uma linguagem orientada por aspectos. Os aspectos gerados pelo Arcabouço são implementados em AspectJ [28]. Na versão atual do sistema, assume-se que o meio de armazenamento persistente é um banco de dados relacional [20]. No entanto, nada impede que o arcabouço seja utilizado com outros meios de armazenamento. Por fim, o sistema não requer que a aplicação alvo obedeça a uma arquitetura particular (por exemplo, que seja um sistema *Web*, um sistema três camadas, etc. O Arcabouço é genérico o suficiente para permitir que persistência seja incorporada a quaisquer aplicações orientadas por objetos desenvolvidas em Java.

1.1 Objetivos

Este trabalho de dissertação de mestrado teve como objetivo geral avaliar o uso de técnicas de programação orientada por aspectos na implementação do requisito transversal de persistência, englobando controle de conexão, controle de transação e sincronização de objetos. Para tanto, foi construído um arcabouço orientado por aspectos para implementação de tal requisito, o qual atende às seguintes exigências:

- fornece mecanismos para controle da conexão com o banco de dados;
- fornece mecanismos para controle de transação garantindo a integridade dos dados;
- fornece mecanismos para sincronização entre dados voláteis e dados persistentes;
- permite que alterações feitas em dados voláteis sejam persistidas de forma transparente para o programador;
- inclui um gerador de aspectos, o qual é responsável por criar de forma automática todos os aspectos responsáveis pelo controle de conexão e transação com o banco e por persistir objetos de uma aplicação alvo em um meio de armazenamento não-volátil;

- pode ser usado em sistemas gerais desenvolvidos em Java;
- implementado em AspectJ [28], uma versão da linguagem Java para a programação por aspectos.

1.2 Contribuições

As principais contribuições deste trabalho são listadas abaixo:

- apresentação e avaliação de um conjunto de arcabouços já existentes na literatura para a implementação de persistência;
- projeto e implementação de um arcabouço orientado por aspectos para modularização de persistência;
- projeto e implementação de uma ferramenta que viabiliza:
  - geração automática dos aspectos para controle de conexão, transação e sincronização de objetos;
  - especialização automática dos aspectos abstratos que compõem o Arcabouço;
  - geração automática de métodos para inserção, atualização, exclusão e recuperação de registros no banco de dados;
  - facilidade de uso.
- avaliação da utilização do Arcabouço e teste da ferramenta em um sistema de comércio eletrônico.

1.3 Estrutura da Dissertação

Este trabalho está organizado da seguinte maneira:

- Capítulo 2 discute o conceito de requisitos transversais e mostra como estes encontram-se entrelaçados junto a lógica de negócios de sistemas e dispersos por todo o código. Além disso, é feita uma apresentação do paradigma de programação orientado por aspectos e uma descrição de AspectJ.

- Capítulo 3 define arcabouços orientados por objetos e apresenta os benefícios e problemas relacionados com o uso de arcabouços. Além disso, definem-se arcabouços orientados por aspectos, apresentado as diferenças em relação aos arcabouços orientados por objetos.

- Capítulo 4 apresenta arcabouços orientados por aspectos para implementação de persistência. Logo em seguida, são apresentados outros trabalhos relacionados com a modularização de persistência.

- Capítulo 5 apresenta o arcabouço proposto para a implementação de persistência. É feito um detalhamento do requisito de persistência e uma descrição simplificada do arcabouço. Logo em seguida, são apresentados os aspectos abstratos para controle de conexão, controle de transação, sincronização dos objetos e recuperação dos dados.

- Capítulo 6 descreve a ferramenta desenvolvida para geração automática de aspectos. Fornece-se uma visão geral do funcionamento da ferramenta, bem como toda a configuração dos arquivos XML necessários para a geração dos aspectos. Logo em seguida, é apresentado como a ferramenta gera os aspectos que especializam os aspectos abstratos do Arcabouço.

- Capítulo 7 apresenta como estudo de caso um Sistema de Comércio Eletrônico. Descreve-se a forma como a ferramenta de geração cria os aspectos para este sistema.

- Capítulo 8 apresenta a importância do uso do arcabouço para o processo de desenvolvimento de *software*, juntamente com uma comparação entre o arcabouço e os trabalhos relacionados descritos no Capítulo 4. Além disso, avalia-se a aplicabilidade, facilidade de uso e extensibilidade do arcabouço.

- Capítulo 9 apresenta as conclusões desta dissertação e perspectivas de trabalhos futuros.

Capítulo 2

Programação Orientada por Aspectos

Um módulo é um artefato de programação que pode ser desenvolvido e compilado separadamente de outras partes que compõem o programa [42, 17, 53]. Na programação estruturada, a modularização se limita à implementação de abstrações dos comandos (expressões, funções e procedimentos) necessários à realização de uma tarefa. Com o advento da orientação por objetos, pode-se obter um grau mais elevado de modularização, por ser possível realizar abstrações de estruturas de dados, tipos e de suas operações por meio da implementação de interfaces.

Um projeto de desenvolvimento de sistemas é constituído de diversas fases. Uma das fases básicas é a definição dos requisitos do sistema. Requisitos de um sistema se dividem em requisitos funcionais, que constituem o objetivo do sistema, e requisitos não-funcionais, que compreendem partes específicas do projeto, muitas vezes não tendo relação direta com o problema em questão [30]. A Figura 2.1 apresenta a visão de um sistema composto de múltiplos requisitos como lógica de negócios, persistência e registro de operações que se tornam entrelaçados à medida que são inseridos nos módulos de implementação. Como cada módulo de implementação trata diversos requisitos pertencentes ao sistema, conseqüentemente a independência dos requisitos não pode ser mantida.

Uma outra visão de como os requisitos podem ser decompostos em um sistema é apresentada na Figura 2.2. Imagine um requisito de sistema sendo decomposto em um espaço n-dimensional de requisitos onde cada um forma uma dimensão [30]. A Figura 2.2 apresenta três dimensões formadas pela lógica de negócio e pelos requisitos transversais *logging* e persistência.

O objetivo desta visão de sistema é destacar que requisitos devem ser mutuamente independentes e conseqüentemente devem ser implementados sem afetar uns aos outros. Por exemplo, alterar o requisito de persistência de banco de dados relacional para banco

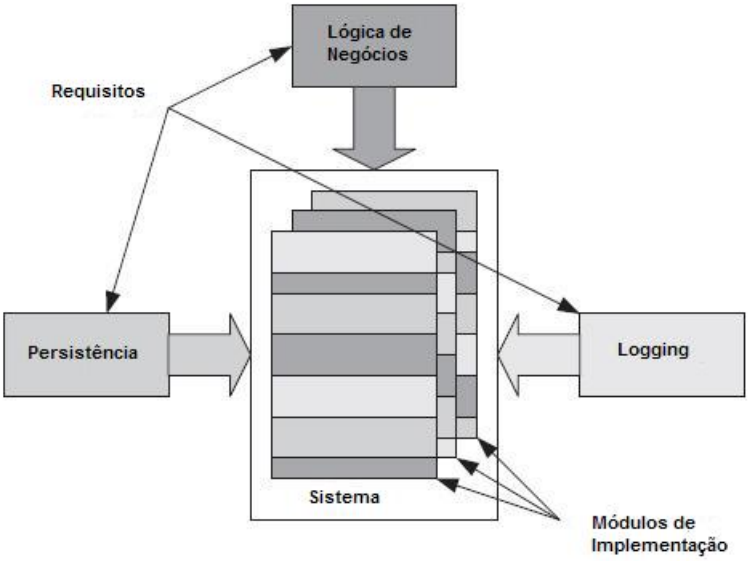

Figura 2.1: Requisitos de Sistemas (figura extraída de [30])

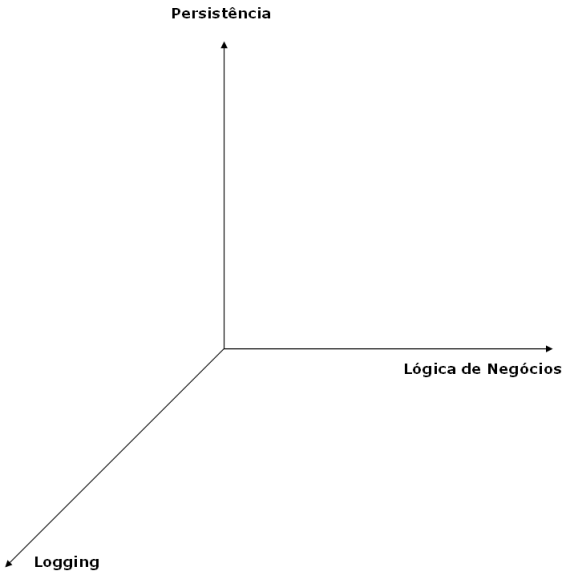

Figura 2.2: Visão N-dimensional de Requisitos de Sistemas (figura extraída de [30])

de dados orientado por objetos não deveria afetar a lógica de negócio ou algum requisito de segurança.

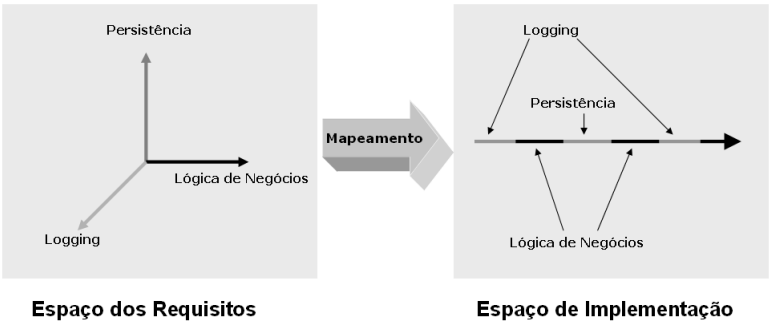

Figura 2.3: Mapeamento entre o Espaço N-dimensional e a Dimensão Única (figura extraída de [30])

Separar, identificar e classificar requisitos de um sistema é um importante exercício no desenvolvimento de software. Uma vez feito isso é possível tratar de forma independente cada requisito em módulos de implementação. Entretanto, tal tratamento em linguagens orientadas por objeto tradicionais pode levar ao problema conhecido de falta de modularização (no caso de requisitos transversais).

A implementação de sistemas requer do engenheiro de software o mapeamento entre o espaço n-dimensional dos requisitos para o espaço de dimensão única de implementação. A Figura 2.3 apresenta este mapeamento. As técnicas atuais de implementação tendem a misturar os requisitos em uma única dimensão levando à dispersão e ao entrelaçamento de código. A dispersão de código se caracteriza quando o código referente a um requisito encontra-se espalhado por vários componentes do sistema. O entrelaçamento de código ocorre quando os códigos referentes a diversos requisitos localizam-se na mesma unidade funcional. A dispersão e o entrelaçamento de código podem acarretar problemas como dificuldade de manutenção, forte acoplamento entre objetos, baixa reutilização de código e dificuldade na rastreabilidade do código.

```
1  public class Logger{
2      public static void log(String message){
3          System.out.println(message);
4      }
5      public static void logEntry(String message){
6          System.out.println("Entering "+message);
7      }
8      public static void logExit(String message){
9          System.out.println("Exiting "+message);
10     }
11 }
```

Listagem 2.1: Logger.java

```
1  public class Authenticator{
2      private static String user;
3      public static boolean authenticate(){
4          //Authentication code
5      }
6      public static void setUser(String user){
7          this.user = user;
8      }
9  }
```

Listagem 2.2: Authenticator.java

Para demonstrar melhor os problemas de entrelaçamento e dispersão do código, considere as classes `Logger` e `Authenticator` nas Listagens 2.1 e 2.2 [51].

A classe `Logger` possui métodos com a função de registrar informações na entrada e na saída de métodos de uma classe. A classe `Authenticator` possui métodos com a função de verificar se o usuário tem permissão para executar determinada tarefa. Seja o exemplo de utilização dos métodos dessas classes na Listagem 2.3.

A dispersão de código ocorre nas diversas chamadas aos métodos da classe `Logger` (`log`, `logEntry` e `logExit`). No exemplo, as chamadas a estes métodos encontram-se somente em uma unidade funcional do sistema, mas poderiam estar dispersas por todas as unidades. O entrelaçamento de código encontra-se na unidade funcional `doSomething`, onde se alternam as chamadas dos métodos das classes `Logger` e `Authenticator`.

Requisitos transversais como os apresentados anteriormente de fato encontram-se entrelaçados em módulos de implementação junto a lógica de negócios do sistema e dispersos por

```
1  public class SomeClass{
2      public void doSomething(int aParameter){
3          if(!Authenticator.authenticate()){
4              Logger.log("Authentication failed");
5          }else{
6              Logger.logEntry("doSomething");
7              //execute method's operations
8              Logger.logExit("doSomething");
9          }
10     }
11 }
```

Listagem 2.3: SomeClass.java

todo o código, resultando em um acoplamento indesejado. Assim, é importante que exista alguma técnica capaz de modularizar requisitos transversais sem que ocorra a dispersão e o entrelaçamento de código.

Programação orientada por aspectos (POA) [29] foi desenvolvida para acrescentar nas metodologias já existentes, tais como programação orientada por objetos (POO) e programação procedimental, conceitos e construções que auxiliem na modularização de requisitos transversais [30].

Analisando os paradigmas POA e POO, observa-se que a principal diferença no que diz respeito à gerência de requisitos transversais é que na POA a implementação de cada requisito funcional fica independente do comportamento transversal que está sendo introduzido dentro dela. Por exemplo, uma regra de negócio não tem consciência de que as operações estão sendo registradas em um arquivo de *log* ou que existe um mecanismo de controle de autenticação.

2.1 Desenvolvimento Orientado por Aspectos

O desenvolvimento de *software* orientado por aspectos é realizado em três fases distintas: decomposição, implementação e recomposição dos requisitos [28]. Estas fases são descritas a seguir:

- a decomposição de requisitos consiste em particioná-los com o intuito de identificar requisitos funcionais e requisitos transversais. Por exemplo, em um sistema de comércio eletrônico, requisitos como cadastro de produtos, cadastro de clientes e fechamento de pedidos são denominados requisitos funcionais. Já requisitos como autenticação

e registro de operações, controle de conexão, controle de transação e sincronização de objetos são denominados requisitos transversais. Em [5], os requisitos transversais são classificados em requisitos de infra-estrutura, de serviços e de paradigmas. Requisitos de infra-estrutura consistem em requisitos responsáveis por coordenação (escalonamento e sincronização), distribuição (tolerância a falhas, comunicação, serialização e replicação) e persistência. Os requisitos de serviços são representados pelos requisitos de segurança, recuperação de erros, operações de retrocesso, rastreamento e registro de operações. Exemplos de requisitos de paradigma são os padrões *visitor* e *observer* [18];

- a etapa de implementação dos requisitos deve ser feita de forma independente. No exemplo de um sistema de comércio eletrônico, deve-se implementar as regras de negócio, *logging* e autenticação, em módulos distintos. A implementação de cada requisito em módulos distintos pode ser feita em classes ou aspectos. Por exemplo, a implementação de *logging* pode ser feita por meio da implementação de classes e interfaces específicas para o problema, possivelmente organizadas por meio de padrões de projetos. É possível, ainda, utilizar bibliotecas externas, tal como Log4J [32];

- após a implementação de cada requisito são definidas as regras para a recomposição do sistema. Estas regras são implementadas em módulos denominados aspectos. Os aspectos definem como os requisitos transversais são compostos para formar o sistema. Esse processo é denominado costura (*weaving*), ou seja, o código referente a cada requisito é inserido no ponto de junção definido pela regra implementada no aspecto.

A Figura 2.4 ilustra as etapa do processo de desenvolvimento de *software* orientado por aspectos. Primeiramente, é feita a decomposição dos requisitos e depois a implementação dos mesmos em módulos individuais. A etapa final consiste em combinar as implementações usando os aspectos para formar o sistema final.

2.2 Benefícios da Programação Orientada por Aspectos

Dentre os benefícios esperados pelo emprego de POA em sistema reais destacam-se os seguintes [30]:

- **diminuição das responsabilidades dos módulos:** a POA permite que um módulo seja responsável somente pelo seu requisito, não sendo responsável por algum outro requisito transversal, possibilitando uma melhora na rastreabilidade. Por exemplo, um módulo que acessa um banco de dados não é responsável pelo *pool* de conexões;

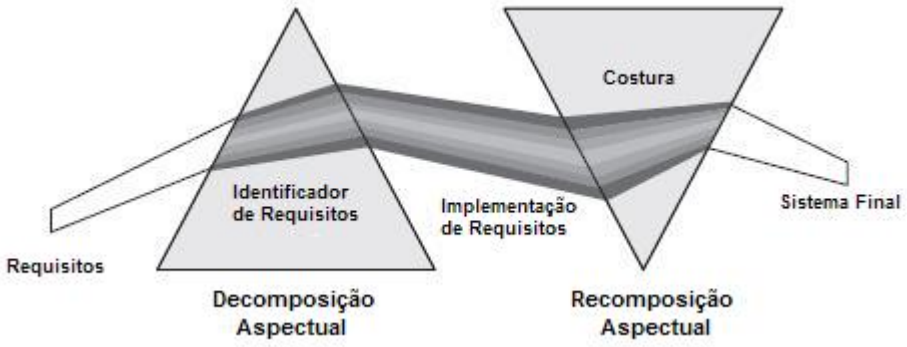

Figura 2.4: Desenvolvimento de Software Orientado por Aspectos (figura extraída de [30])

- **modularização:** a POA fornece um mecanismo para implementar cada requisito separadamente com o mínimo de acoplamento. O resultado é uma implementação modularizada mesmo na presença de requisitos transversais e um sistema com muito menos duplicação de código, uma vez que a implementação de cada requisito é separada. Além disso, há uma maior facilidade de entendimento e manutenção do sistema;

- **facilidade na evolução do sistema:** a POA modulariza os requisitos transversais em aspectos e faz com que módulos funcionais não tenham consciência de que esses requisitos estão sendo interceptados. Adicionar uma nova funcionalidade a um aspecto que modulariza um requisito transversal não requer alterações no módulos funcionais. Além disso, quando um novo módulo é adicionado ao sistema, os aspectos existentes podem entrecortar o mesmo, evitando retrabalho;

- **amarração tardia de implementação de requisitos:** com a POA, o engenheiro de *software* pode adiar a implementação de alguns requisitos, uma vez que é possível implementá-los posteriormente em aspectos separados. Novos requisitos com características transversais podem ser manipulados criando novos aspectos;

- **reusabilidade de código:** a chave para uma maior reusabilidade de código está em uma implementação com alto grau de coesão. A POA permite a implementação com alta coesão, pois implementa cada aspecto como um módulo separado.

- **redução do custo da implementação:** a POA permite a redução do custo da implementação, porque evita o custo de modificação dos muitos módulos que implementam requisitos transversais. Outro fator que possibilita a redução do custo de implementação é o aumento da produtividade do desenvolvedor. Quando um desen-

volvedor está focado na implementação de um único requisito, sem se preocupar com outros, a sua produtividade aumenta.

2.3 Linguagens Orientadas por Aspectos

Uma linguagem de programação orientada por aspectos é composta por vários recursos, dentre os quais destacam-se os seguintes: pontos de junção (*joinpoints*), conjuntos de junção (*pointcuts*) e regras de junção (*advices*). Pontos de junção são pontos de execução do programa, por exemplo, chamadas de métodos. Conjuntos de junção são pontos de junção agrupados, definindo um contexto. Por último, regras de junção referem-se ao código dos requisitos transversais que devem ser executados em pontos de junção.

Linguagens orientadas por aspectos devem possuir ainda recursos para definir e identificar pontos de junção e uma forma para interferir na execução de pontos de junção. Esses recursos devem estar agrupados em aspectos. O compilador de aspectos é denominado *weaver* e sua função é realizar o processo de costura do código (*weaving*), entrelaçando os códigos das regras de junção nos pontos de junção especificados nos aspectos.

A Figura 2.5 ilustra o uso de uma linguagem orientada por aspectos que fornece um compilador responsável por realizar a costura.

A linguagem AspectJ permite a introdução do conceito de aspectos em Java. Seus mecanismos de transversalidade permitem realizar interferências tanto na estrutura estática das classes de um programa Java quanto no comportamento do programa durante a sua execução. Dentre outras linguagens que implementam orientação por aspectos, destacam-se HyperJ [41], AspectC++ [3], AspectC [10], Jiazzi [36] e AspectWerkz [4]. Existem também propostas para POA independente de linguagem [31].

A linguagem HyperJ [41] foi desenvolvida pela IBM para o suporte à metodologia de separação multi-dimensional e integração de requisitos em Java. O objetivo desta metodologia é permitir que cada requisito seja implementado em uma dimensão distinta, de modo que o ambiente de compilação faça a composição, de forma semelhante à feita em AspectJ. A linguagem AspectC++ [3] é uma implementação de orientação por aspectos em C++, com projeto baseado na linguagem AspectJ. A linguagem Jiazzi [36] é uma extensão de Java que permite a compilação separada de módulos denominados *units*. Estas *units* funcionam de forma semelhante a aspectos em AspectJ, podendo alterar a estrutura e o comportamento de classes. AspectC [10] é uma extensão orientada por aspectos de C. AspectWerkz [4] implementa transversalidade estática e dinâmica, permitindo que o processo de costura de código seja feito durante a compilação ou durante a execução. A sua implementação é flexível, permitindo que código transversal seja implementado como combinação de código Java com anotações e arquivos XML.

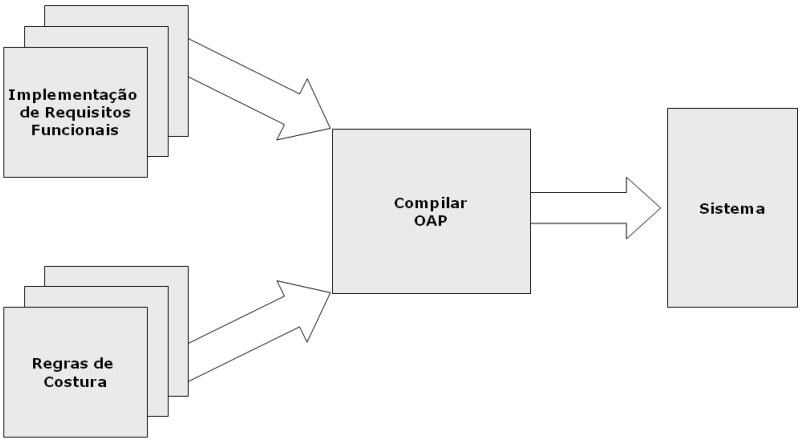

Figura 2.5: Compilador de Aspectos (figura extraída de [30]

O poder de expressão e recursos especiais destas linguagens para desenvolvimento de sistemas baseados em aspectos situam-se em um mesmo patamar de equivalência. Entretanto, AspectJ destaca-se por sua maior difusão.

2.4 AspectJ

AspectJ [29, 23, 30, 51] é uma extensão da linguagem Java com suporte a programação orientada por aspectos. AspectJ estende Java com dois tipos de implementações transversais [29, 51]: transversalidade dinâmica (*dynamic crosscuting*), que permite definir implementação adicional em pontos bem definidos do programa, e transversalidade estática (*static crosscuting*) que permite definir novas operações em assinaturas estáticas de classes e interfaces de um programa em Java.

Atualmente, o desenvolvimento do ambiente de execução da linguagem é um projeto Eclipse [19], cujo *website* contém ferramentas para o desenvolvimento de programas orientados por aspecto.

2.4.1 Transversalidade Dinâmica

Transversalidade dinâmica consiste em introduzir um novo comportamento na execução de um programa. Portanto a transversalidade dinâmica modifica o comportamento do sistema alterando seu fluxo de execução. Por exemplo, se deseja-se especificar que uma certa ação será realizada antes da execução de algum método, pode-se especificar o ponto do programa e a ação que será executada nesse ponto em um módulo separado.

2.4.2 Transversalidade Estática

Transversalidade estática consiste em introduzir modificações nas estruturas estáticas do programa como classes, interfaces e aspectos do sistema. A transversalidade estática não modifica o comportamento do sistema. Em AspectJ, é possível implementar os seguintes tipos de transversalidade estática:

- introdução de campos e métodos em classes e interfaces;
- modificação da hierarquia de tipos;
- declaração de erros e advertências de compilação;
- enfraquecimento de exceções.

2.4.3 Elementos Transversais

Os principais recursos de AspectJ são os seguintes [51]:

- **pontos de junção** (*join points*)**:** são pontos bem definidos da execução de um programa como, por exemplo, chamadas de métodos ou execução de blocos de tratamento de exceções. Pontos de junção podem também possuir contexto associado;
- **conjuntos de junção** (*pointcuts*)**:** são formados por um conjunto de pontos de junção e as informações de contexto destes pontos. Eles permitem a especificação de coleções de pontos de junção e a exposição de seus contextos para a implementação de regras de junção. Por exemplo, o designador de conjunto de junção `call(void Point.setX(int))` identifica todos os pontos de junção formados por chamadas do método de assinatura `void Point.setX(int)`. A linguagem AspectJ permite a especificação dos seguintes pontos de junção:

  - chamadas de métodos e construtores por meio do designador `call`. A construção `call(void Point.getX(int))` representa os pontos de junção que são chamadas do método `getX` da classe `Point` sem valor de retorno e com um parâmetro inteiro. Já a construção `call(Point.new(int,int))` representa os pontos de chamadas ao construtor da classe `Point`;
  - execuções de métodos e construtores por meio do designador `execution`. A construção `execution(void Point.getX(int))` representa todas as execuções do método `getX` da classe `Point` sem valor de retorno e com um parâmetro inteiro. Já a construção `execution(Point.new(int,int))` designa os pontos de junção de execução do construtor da classe `Point` que recebe dois parâmetros inteiros. A diferença entre os designadores `call` e `execution` é que o `call` representa o momento das chamadas de métodos e construtores quando a passagem de parâmetros está sendo feita, já o `execution` representa o momento da execução dos métodos e construtores quando a passagem de parâmetro já foi feita;
  - acesso a campos de classes e objetos via conjuntos de junção que capturam acessos de leitura e escrita a atributos estáticos e não-estáticos de uma classe. A captura da leitura de um campo utiliza o operador `get` e a de escrita, o operador `set`, ambos parametrizados com a assinatura do campo. Por exemplo, a construção `get(PrintWriter System.out)` representa os acessos de leitura do campo `out` da classe `System`, ao passo que `set(int MyClass.x)` representa as operações de escrita do campo `x` da classe `MyClass`;
  - execuções de tratadores de exceção via conjuntos de junção que capturam a execução dos blocos `catch`. Eles podem ser definidos por meio do operador `handler`, parametrizado pelo nome da classe de exceção. Exemplo `handler(NumberFormatException)`;
  - execuções de inicialização de classes via conjuntos de junção que capturam a execução do código especificado no bloco `static` dentro das definições de classes. A identificação é feita por meio do operador `staticinitialization`, parametrizado pelo nome da classe.

- **reflexão computacional:** A linguagem AspectJ suporta reflexão computacional via um objeto especial chamado `thisJoinPoint` que permite obter informações em tempo de execução sobre pontos de junção. Esse tipo de reflexão pode ser importante, por exemplo para definição de aspectos para registro de *log* de operações e depuração do código;

- **regras de junção** (*advices*)**:** são construtos que fornecem código para expressar a ação transversal nos pontos de junção que são capturados pelos conjuntos de junção

[30]. AspectJ permite definir regras de junção que são executadas antes (`before`), após (`after`) ou no lugar de (`around`) pontos de junção. A regra de junção `around` é especial por permitir redirecionar a execução para outro ponto, continuar a execução original ou causar uma alteração no contexto da execução. A Figura 2.6 ilustra o diagrama de seqüência que apresenta a regra de junção `around` que entrecorta o método `inserirCliente(..)`, inserindo chamadas aos métodos que inicializam e finalizam uma transação. Além disso, este diagrama apresenta a regra de junção `after` que entrecorta a construtora da classe `Cliente` e persiste o objeto criado no banco de dados;

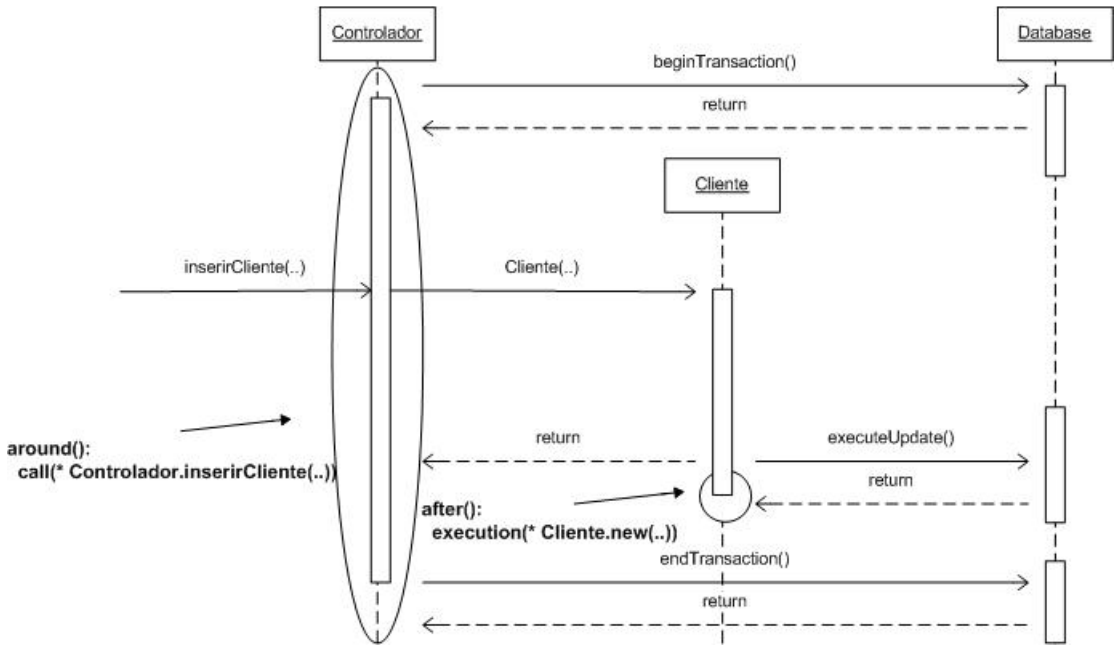

Figura 2.6: Diagrama de Seqüência

- **aspectos** (*aspects*)**:** Um aspecto é uma unidade modular de implementação de requisitos transversais. Aspectos são definidos por declarações, que são similares a declarações de classes em Java. Declarações de aspectos podem declarar conjuntos de junção e regras, incluir métodos e campos e podem estender classes ou outros aspectos. Ao contrário de classes, aspectos não podem ser instanciados. Para obter uma referência para um aspecto, utiliza-se o método estático `aspectOf()`, presente em todos os aspectos. O controle de visibilidade de membros de aspectos em AspectJ é igual ao controle de visibilidade de membros de classes em Java, sendo possível utilizar os modificadores `public`, `private` ou `protected`, ou ainda definir visibilidade de pacotes, se nenhum dos três modificadores for utilizado.

```
1 public class SomeClass{
2     public void doSomething(int aParameter){
3         //execute the method operations
4     }
5 }
```

Listagem 2.4: SomeClass.java

Aspectos podem ser estendidos ou especializados, sendo neste caso denominados subaspectos, formando uma hierarquia de aspectos. Não é permitida a definição de hierarquias múltiplas de aspectos. As regras para redefinição de conjuntos e regras de junção em subaspectos são similares às regras de redefinição de métodos em Java. Um aspecto pode ser abstrato, isto é, pode possuir conjuntos de junção abstratos. A semântica de conjuntos de junção abstratos é semelhante a de métodos abstratos: as decisões de implementação são adiadas para aspectos concretos que estendam aspectos abstratos.

A linguagem AspectJ possui outras construções para a implementação de requisitos transversais. Entretanto, não cabe nesse momento enfocar tais construções. O objetivo dessa seção foi simplesmente mostrar os recursos principais da linguagem que serão utilizados nesta dissertação.

2.4.4 AspectJ: Exemplo

Esta seção descreve exemplos de uso dos elementos transversais descritos na seção 2.4.3. Suponha que o código do método `doSomething` da classe `SomeClass`, definido na Listagem 2.3, seja agora implementado usando AspectJ como apresentado na Listagem 2.4.

Analisando o código, observa-se que todos os requisitos transversais foram removidos do método `doSomething`. Dessa forma, o método conterá somente a funcionalidade alvo para que foi projetado. As classes `Logger` e `Authenticator` não serão modificadas, pois encapsulam de forma correta os requisitos existentes. Dois aspectos serão criados para aplicar transversalmente os requisitos das classes `Logger` e `Authenticator`. O código do aspecto `LoggingAspect` é mostrado na Listagem 2.5.

O código para o aspecto `AuthenticatingAspect` é descrito na Listagem 2.6.

Ambos os aspectos implementam os requisitos transversais que entrecortavam o código do método `doSomething` através dos *advices*. Os modificadores `before`, `after` e `around` definem o ponto do código em que os requisitos serão executados. No aspecto `LoggingAspect`, os *advices* `before(): loggableCalls()` e `after(): loggableCalls()` definem, respectivamente, que antes e depois da execução dos métodos públicos do programa, com ex-

```
1  public aspect LoggingAspect {
2      pointcut publicMethods(): execution(public * *(..));
3      pointcut logObjectCalls(): execution(* Logger.*(..));
4      pointcut loggableCalls(): publicMethods() && ! logObjectCalls();
5
6      before(): loggableCalls() {
7          Logger.logEntry(thisJoinPoint.getSignature().toString());
8      }
9      after(): loggableCalls() {
10         Logger.logExit(thisJoinPoint.getSignature().toString());
11     }
12 }
```

Listagem 2.5: LoggingAspect.aj

ceção dos pertencentes à classe `Logger`, os métodos `logEntry` e `logExit` serão executados. Para o aspecto `AuthenticatingAspect`, o *advice* `around(): methodsAuthentication()` contém o código para autenticar a execução do método `doSomething`.

Como descrito na Seção 2.4.3, aspectos podem ser estendidos ou especializados formando uma hierarquia de aspectos. Para exemplicar o uso de aspectos abstratos e concretos observe na Listagem 2.7 o aspecto abstrato `AbstractLogging` e o aspecto concreto `EcommerceLogging` que especializa os conjuntos de junção abstratos definidos no aspectos `AbstractLogging`.

No aspecto concreto, o conjunto de junção `logPoints` foi especializado para capturar todas as chamadas métodos das classes que fazem parte do pacote `ecommerce`. A implementação do método `getLogger()` retorna o registro de operações que está especificado para o sistema Ecommerce. É possível criar outros subaspectos que especializam o aspecto `AbstractLogging` e fornecem definições específicas.

2.5 Comentários Finais

Programação orientada por aspectos é um novo paradigma de programação que tem por objetivo modularizar requisitos transversais de um sistema, que por sua vez não podem ser modularizados por meio de programação orientada por objetos. A POA vem para separar os requisitos transversais do sistema dos requisitos não transversais, reduzindo, ou até mesmo eliminando, a dispersão e o entrelaçamento de código.

A programação orientada por aspectos não é um substituto para a programação orien-

```
1  public aspect AuthenticatingAspect {
2      pointcut methodsAuthentication(): execution(
3                      *SomeClass.doSomething(..));
4      void around(): methodsAuthentication(){
5          boolean authenticated = Authenticator.authenticate();
6          if(authenticated){
7              proceed();
8          }else{
9              Logger.log("Autenticacao falhou");
10         }
11     }
12 }
```

Listagem 2.6: AuthenticatingAspect.aj

tada por objetos. Os requisitos funcionais de um sistema continuarão a ser implementados por meio da POO. A orientação por aspectos simplesmente adiciona novos conceitos à POO, facilitando a implementação de requisitos transversais e retirando grande parte da atenção dada a tais requisitos nas fases iniciais de desenvolvimento. Com efeito, o desenvolvedor pode se concentrar na implementação dos módulos de requisitos funcionais do sistema, permitindo que a implementação dos requisitos transversais seja feita separadamente.

Além disso, a evolução de sistemas é facilitada, visto que se novos aspectos forem criados para implementar novos requisitos transversais, as classes do programa podem permanecer inalteradas. Da mesma forma, ao adicionar novas classes ao programa, os aspectos existentes também são transversais a estas classes.

Cumpre mencionar que, como principal ponto negativo, tem-se que a orientação por aspectos pode quebrar o encapsulamento de módulos, ao permitir acesso a detalhes de sua implementação. Se mal utilizada, pode afetar negativamente o funcionamento de módulos funcionais que já foram testados e homologados.

Outro ponto negativo é que linguagens de programação orientadas por aspectos tem dificuldade com raciocínio modular [55]. Raciocínio modular significa que podemos pensar a respeito de um módulo, independentemente do contexto em que este módulo está inserido e que podemos deduzir o seu comportamento a partir de seus componentes. As linguagens de programação orientadas por aspectos violam estes princípios.

Entretanto, estes problemas deverão ser resolvidos em futuro próximo, considerando os benefícios que o uso POA pode trazer para a área de desenvolvimento de *software*.

```
1  public abstract aspect AbstractLogging {
2      public abstract pointcut logPoints();
3      public abstract Logger getLogger();
4      before() : logPoints() {
5          getLogger().log("Before: " + thisJoinPoint);
6      }
7  }
8  public aspect EcommerceLogging extends AbstractLogging {
9      public pointcut logPoints()
10                         : call(* ecommerce..*(..));
11     public Logger getLogger() {
12         return Logger.getLogger("ecommerce");
13     }
14 }
```

Listagem 2.7: AbstractLogging.aj e EcommerceLogging.aj

Capítulo 3

Arcabouços

De acordo com Johnson [26], um arcabouço (*framework*) é um projeto reutilizável das partes de um sistema representado por um conjunto de classes abstratas e a maneira que instâncias dessas classes interagem entre si. Em outras palavras, é uma aplicação que pode ser especializada para produzir outros aplicativos [21]. O seu propósito é fornecer recursos para auxiliar um desenvolvedor de sistemas na implementação de aplicações [26]. Outra definição feita também por Johnson apresenta o arcabouço como sendo o esqueleto de uma aplicação que pode ser especializado de acordo com as intenções de um desenvolvedor. Segundo Johnson, essas duas definições não são conflitantes, pois a primeira descreve a estrutura de um arcabouço, enquanto a segunda descreve o propósito de um arcabouço.

Arcabouço é uma técnica de reúso orientada por objetos que tem sido utilizada com eficácia na construção de sistemas. A reusabilidade é definida como a propriedade de um software ser utilizado em novos sistemas. Uma das formas de reúso é o de código por meio de componentes. Componente é um conjunto de classes que estão intimamente relacionadas e que oferece uma funcionalidade específica. Os componentes podem ser facilmente utilizados para a criação de novos sistemas. Uma outra forma de reúso é o de projeto, que consiste na utilização de padrões de projeto [22]. Arcabouços permitem tanto o reúso de código quanto o reúso de projeto.

A instanciação de um arcabouço para uma dada aplicação pode ser feita com a criação de subclasses específicas para essa aplicação que especializam as classes abstratas do arcabouço. Outra maneira de especialização consiste no uso de componentes que acompanham o arcabouço. Esses componentes são implementações das classes abstratas do arcabouço que permitem ao usuário escolhas entre os componentes para construir sua aplicação. Assim, o reúso de código em um arcabouço pode ser feito pelo uso dos componentes existentes ou por meio da herança aproveitando boa parte da implementação da superclasse.

O uso do arcabouço no desenvolvimento de um sistema determina a arquitetura desse sistema. São definidas a estrutura geral do sistema juntamente com a sua divisão em classes e objetos, o fluxo de controle de execução, bem como as responsabilidades de cada classe e como as mesmas interagem entre si. Diferentemente do que acontece com o uso de bibliotecas, onde o engenheiro de software deve definir a estrutura geral do sistema, o seu fluxo de controle e os pontos das chamadas aos componentes da biblioteca.

Outra característica fundamental de um arcabouço é a inversão de controle [35]. A inversão de controle se caracteriza quando classes pertencentes ao arcabouço contêm invocação de métodos de classes que especializam classes abstratas do arcabouço. Esses métodos executados são específicos para o sistema que está sendo desenvolvido.

Arcabouços devem ser construídos obedecendo o Princípio da Inversão de Dependência [33]. Esse princípio diz que módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações, e abstrações não devem depender de detalhes, são os detalhes que devem depender de abstrações. Quando módulos de alto nível dependem de módulos de baixo nível, torna-se difícil o seu reúso em diferentes contextos. Entretanto, quando módulos de alto nível são independentes, o seu reúso torna-se viável. Para exemplificar um programa que infringe esse princípio, observe o exemplo apresentado na Figura 3.1 com três módulos, Copiar, LerTeclado e EscreverImpressora.

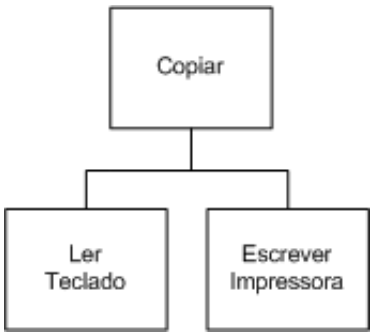

Figura 3.1: Programa Copiar

Uma implementação para este programa é mostrada na Listagem 3.1.

Os módulos de baixo nível LerTeclado e EscreverImpressora são módulos reúsaveis. Estes módulos poderiam ser utilizados em vários outros programas que utilizam acesso ao teclado e a impressora. No entanto, o módulo de alto nível Copiar não poderia ser usado em nenhum outro contexto que não envolva teclado e impressora. Analisando este problema, observa-se que o módulo de alto nível Copiar é dependente de módulos de baixo nível como LerTeclado e EscreverImpressora. Para que o módulo Copiar seja reusável deve-se torná-lo independente. A Figura 3.2 apresenta a solução para este problema.

```
1 void Copiar(){
2     char c;
3     while ((c = LerTeclado()) != EOF)
4         EscreverImpressora(c);
5 }
```

Listagem 3.1: Método Copiar

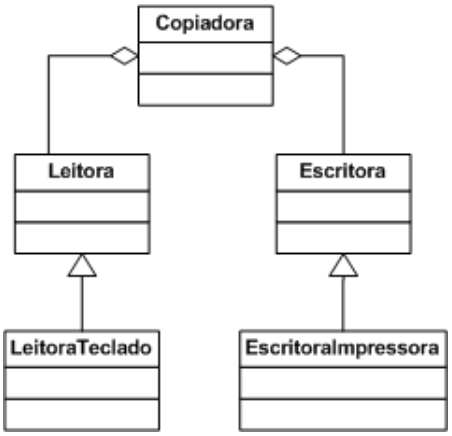

Figura 3.2: Copiar Reusável

A classe `Copiadora` utiliza recursos das classes abstratas `Leitora` e `Escritora`. A classe `Copiadora` contém o método `copiar`, apresentado na Listagem 3.2, que recebe como parâmetro instâncias de classes que estendem as classes abstratas `Leitora` e `Escritora`, como por exemplo `LeitoraTeclado` e `EscritoraImpressora`. Esse método possui um comando de iteração que obtém os caracteres através do método `ler` e envia esses caracteres através do método `escrever`. O método `copiar` torna-se reusável independente do tipo de leitor ou escritor. Novas classes de leitores e escritores podem ser criadas, bastando que estas estendam `Leitora` e `Escritora`. Assim, a classe `Copiadora` se torna independente das classes `LeitoraTeclado` e `EscritoraImpressora`, provendo a inversão de dependência.

Outra característica importante é que arcabouços possuem *hot spots* e *frozen spots* [46]. *Hot spots* são pontos de instanciação do arcabouço, ou seja, pontos nos quais o usuário deve ligar os módulos de implementação específicos do seu sistema com os módulos de implementação do arcabouço. Por exemplo, especialização de alguma classe ou aspecto abstrato do arcabouço. *Frozen spots* são partes fixas já implementadas do código, que disponibilizam recursos para algum requisito. Por exemplo, um arcabouço pode possuir partes fixas implementadas referente ao requisito de controle de transação. Assim, um

```
1  public abstract class Leitora{
2      public abstract char ler();
3  }
4  public abstract class Escritora{
5      public abstract void escrever(char c);
6  }
7  public class Copiadora{
8      void copiar(Leitora r, Escritora e){
9          char c;
10         while ((c = r.ler()) != EOF)
11             e.escrever(c);
12     }
13 }
```

Listagem 3.2: Método Copiar

sistema que utiliza este arcabouço, poderá fazer uso deste recurso.

Arcabouços são mais abstratos e flexíveis que os componentes, porém mais difíceis de serem usados. Por outro lado, os arcabouços são mais concretos e fáceis de serem reutilizados, porém menos flexíveis e aplicáveis que os padrões de projetos. Quanto mais específico for um arcabouço, provavelmente mais fácil será utilizá-lo. Se um arcabouço for genérico, ou seja, puder ser aplicado a diversas tipos de aplicação, conseqüentemente, terá mais parâmetros e opções a serem configuradas, tornando a sua utilização mais difícil. Definitivamente, arcabouço é uma técnica de reúso orientada por objetos que tem sido utilizada com eficácia na construção de sistemas. De certa forma, já faz parte da cultura dos desenvolvedores a sua utilização.

3.1 Arcabouços Caixa Branca e Caixa Preta

Arcabouços podem ser classificados em sistemas do tipo caixa branca e sistemas do tipo caixa preta [21, 25]. Arcabouços caixa branca utilizam características das linguagens orientadas por objetos como herança, amarração dinâmica e redefinição de métodos como técnicas de extensão. Já arcabouços caixa preta suportam extensibilidade por definição de interfaces para componentes.

Arcabouços caixa branca demandam que o engenheiro de software conheça em detalhes a estrutura interna do arcabouço. Em contraste, os arcabouços caixa preta são estruturas que usam composição de objetos ao invés de herança, conseqüentemente o desenvolvedor

não precisa ter um profundo conhecimento sobre a estrutura interna do arcabouço. Este fato os torna mais fáceis de serem usados porém com uma menor flexibilidade em relação aos arcabouços caixa branca.

3.2 Arcabouços Orientados por Aspectos

Com o aparecimento da programação orientada por aspectos e da linguagem AspectJ, surgiram novos conceitos e mecanismos que podem ser utilizados no desenvolvimento de arcabouços. Vários autores [12, 43, 49, 52] têm usado o conceito de arcabouço orientado por aspectos como sendo um arcabouço orientado por objetos que também usa estruturas de aspectos em sua implementação.

Hanenberg [52] define um arcabouço orientado por aspectos como sendo uma coleção de aspectos concretos e abstratos. Segundo Hanenberg, além dos métodos-gancho (*hook methods*), que são conceitos amplamente utilizados em arcabouços orientados por objetos, os conjuntos de junção (*pointcuts*) também podem ser especializados, permitindo que um interesse seja acoplado a inúmeros módulos de implementação.

Camargo e Masiero [13] definem arcabouço orientado por aspectos, do ponto de vista de estrutura, como sendo um conjunto formado por unidades básicas de programação orientada por objetos (classes), cuja presença não é obrigatória, e unidades básicas de programação orientada pos aspectos (aspectos). Do ponto de vista do propósito, um arcabouço orientado por aspectos pode ser definido como um sistema semi-completo e reutilizável que pode ser instanciado por um desenvolvedor de aplicações.

Quanto à natureza do arcabouço orientado por aspectos, Camargo e Masiero definem dois tipos: Arcabouços Transversais e Arcabouços de Aplicação. Um arcabouço transversal possui mecanismos de composição abstratos e variabilidades correspondentes a um único interesse transversal, como por exemplo: persistência, distribuição, segurança e regras de negócio. Já um arcabouço de aplicação é um arcabouço que implementa uma arquitetura genérica de um domínio e sua instanciação produz uma aplicação deste domínio. Por exemplo, um arcabouço de aplicação para construção de sistemas de comércio eletrônico.

3.3 Benefícios e Problemas do Uso de Arcabouços

Os benefícios do uso de arcabouços estão relacionados com a reusabilidade de código. Alguns desses benefícios são:

- introduzir maior qualidade e confiabilidade nas aplicações;

- reduzir a manutenção;
- evitar a recriação e reavaliação das soluções antes criadas;
- requer menor tempo de desenvolvimento, uma vez que aumenta a produtividade dos desenvolvedores, que podem dedicar seus esforços às funcionalidades alvo do sistema.

Os principais problemas do uso de arcabouços são devidos à sua complexidade. Alguns desses problemas são:

- o desenvolvimento de um arcabouço com qualidade, flexibilidade, extensibilidade e reusabilidade pode ser uma tarefa árdua;
- aprender a usar um arcabouço é difícil, o que pode demandar um tempo considerável de treinamento;
- as estruturas genéricas, que melhoram a flexibilidade e a extensibilidade do arcabouço, dificultam a depuração de código, pois essas estruturas não podem ser depuradas separadamente de suas instanciações.

3.4 Comentários Finais

O uso de arcabouços no desenvolvimento de um sistema determina sua estrutura geral, juntamente com a sua divisão em classes, fluxo de controle de execução, bem como as responsabilidades de cada classe e como as mesmas interagem entre si. Arcabouços podem ser classificados em sistemas caixa branca, que utilizam características das linguagens orientadas por objetos, como herança, amarração dinâmica e redefinição de métodos, e em sistemas caixa preta, que suportam extensibilidade por definição de interfaces para componentes. Além disso, arcabouços possuem *hot spots* que ligam o arcabouço com o sistema que está sendo construído e *frozen spots* que disponibilizam recursos para algum determinado requisito.

Com o aparecimento da programação orientada por aspectos e da linguagem AspectJ, surgiram novos conceitos e mecanismos que podem ser utilizados no desenvolvimento de arcabouços. Alguns autores têm usado o conceito de arcabouço orientado por aspectos. Arcabouços orientados por aspectos são definidos como sendo um arcabouço orientado por objetos que também usam estruturas de aspectos em sua implementação.

O objetivo desse capítulo foi apresentar os conceitos e características de arcabouços orientados por objetos e orientados por aspectos, com o intuito de classificar o arcabouço implementado nessa dissertação. O arcabouço apresentado nos capítulos a seguir é um

arcabouço orientado por aspectos, do tipo caixa branca e que pode ser classificado quanto a sua natureza como um arcabouço transversal.

O arcabouço implementado é orientado por aspectos pelo fato de usar estruturas de aspectos em sua implementação. É do tipo caixa branca, por possuir características como herança, amarração dinâmica e sobrescrever métodos como técnica de extensão. O usuário fica responsável pela redefinição dos aspectos abstratos para controle de conexão, controle de transação e sincronização dos objetos, enquanto o Arcabouço fica responsável pela arquitetura do sistema e fluxo de controle. Entretanto, como o Arcabouço possui uma ferramenta de geração automática de aspectos, a tarefa de redefinição dos aspectos abstratos fica a cargo da ferramenta e não do usuário.

Os *hot spots* do Arcabouço implementado nesta dissertação se encontram nos conjuntos de junção abstratos pertencente aos aspectos abstratos do arcabouço. Os *frozen spots* se encontram dentro de partes concretas dos aspectos abstratos do arcabouço, como por exemplo, as regras de junção. Usando a definição de arcabouço proposta por Camargo e Masiero [13], este arcabouço pode ser considerado um arcabouço transversal, ou seja, possui mecanismos de composição abstratos e variabilidades correspondentes a um único interesse transversal: persistência.

Capítulo 4

Arcabouços Orientados por Aspectos para Implementação de Persistência

Neste capítulo, são apresentados três arcabouços orientados por aspectos para implementação de persistência:

- arcabouço proposto por Soares, Borba e Laureano [49];
- arcabouço proposto por Camargo, Ramos, Penteado e Masiero [14];
- arcabouço proposto por Rashid e Chitchyan [43].

4.1 Arcabouço Proposto por Soares, Borba e Laureano

Soares, Borba e Laureano argumentam que AspectJ é bastante útil para modularizar requisitos de persistência e distribuição e propõem um arcabouço para implementação de distribuição e persistência usando AspectJ. Descrevem ainda como este arcabouço foi utilizado para reestruturar um sistema *Web* denominado *Health Watcher*.

Para a remoção do código de persistência da implementação do sistema *Health Watcher* usando Java puro, foi criado um conjunto de aspectos responsáveis pela implementação da funcionalidade de persistência para todas as operações que acessam bancos de dados. Dentre esses aspectos, destacam-se o aspecto abstrato `AbstractPersistenceControl` responsável pela execução do processo de inicialização de persistência, `PersistenceControlHW` que especializa o aspecto anterior para o sistema *Health Watcher* e o aspecto concreto `UpdateStateControl`, responsável pela sincronização dos objetos no banco de dados.

O aspecto `AbstractPersistenceControl` possui um conjunto de junção abstrato responsável pela definição do ponto do sistema no qual será necessário o controle de persistência. Já o aspecto `UpdateStateControl` é responsável pela sincronização dos dados *dirty*, ou seja, dados que precisam ser persistidos no banco. As classes `Complain` e `HealthUnit` são definidas no aspecto `UpdateStateControl` como as classes persistentes do sistema *Health Watcher*. Além disso, define-se um conjunto de junção responsável por armazenar em uma estrutura de dados os objetos *dirty* do sistema. A sincronização dos objetos é feita percorrendo esta estrutura de dados e invocando o método `synchronizedObject` implementado por cada classe persistente.

No trabalho de Soares, Borba e Laureano, também são apresentados aspectos responsáveis pelo controle de transação. Para tal requisito foi criado o aspecto abstrato `AbstractTransactionControl` e a especialização desse aspecto `TransactionControlHW`. O aspecto `AbstractTransactionControl` possui um conjunto de junção abstrato responsável pela definição dos métodos transacionais do sistema, ou seja, métodos que necessitam de controle de transação. Regras são definidas para que antes desses métodos seja iniciado o controle da transação e depois seja finalizado por meio de confirmações *commit*, caso não ocorra algum erro, e *rollback*, caso alguma exceção seja levantada.

O trabalho mostra portanto que AspectJ é útil para separação dos requisitos transversais de persistência e distribuição dos requisitos funcionais do sistema. São mostrados também vantagens da implementação com AspectJ em relação a Java. Um simples arcabouço pode ser extraído da implementação de persistência usando AspectJ devido à existência de aspectos abstratos que podem ser especializados para qualquer tipo de sistema que siga a mesma arquitetura do `Health Watcher`.

O arcabouço proposto, no entanto, deixa para o programador a implementação de alguns métodos importantes, como é o caso do método `synchronizedObject`, que deve ser implementado dentro das classes persistentes do sistema. O método `synchronizedObject` poderia ser criado por uma ferramenta de geração que obteria as informações necessárias para criação acessando tabelas de um banco de dados. Após a geração desse método, o mesmo poderia ser introduzido na classe usando os recursos da transversalidade estática de AspectJ. Além disso, a criação de aspectos que especializam os aspectos abstratos `AbstractPersistenceControl` e `AbstractTransactionControl`, responsáveis pelo controle de persistência e transação, também é feita de forma manual, sem suporte de uma ferramenta de geração código. Para sistemas maiores, tal processo torna-se tedioso para o desenvolvedor e sujeito a erros, podendo ocasionar aumento nos custos da etapa de construção do sistema. Os próprios autores reconhecem que seria bastante útil possuir uma ferramenta que automatize a criação dos aspectos necessários. Além disso, não são apresentadas no artigo formas para a recuperação dos dados no banco. Tal requisito é de grande relevância, já que em um sistema real o uso de consultas SQL em um banco de dados é

muito comum.

4.2 Arcabouço Proposto por Camargo, Ramos, Penteado e Masiero

Camargo, Ramos, Penteado e Masiero propõem um projeto baseado em aspectos para o padrão Camada de Persistência orientado por objetos proposto por Yoder e outros em [54]. Além disso, são propostas diretrizes que auxiliam na obtenção de um projeto baseado em aspectos para um sistema orientado por objetos que tenha sido desenvolvido usando padrões de projeto.

De acordo com o artigo, o padrão camada de persistência pode ser implementado separando-se interesses funcionais dos interesses que referem-se somente ao padrão, facilitando dessa forma o entendimento do padrão e melhorando sua reusabilidade. Para tal, realiza-se uma comparação detalhada da implementação baseada em aspectos com uma orientada por objetos.

O padrão camada de persistência é responsável por cuidar da interface entre objetos da aplicação e tabelas do banco de dados. Esse padrão é composto de um conjunto de dez sub-padrões, dos quais somente cinco (Camada Persistente, CRUD, Gerenciador de Tabelas, Métodos de Mapeamento de Atributos e Gerenciador de Conexão) são utilizados na implementação apresentada no artigo. Camada Persistente consiste no salvamento e recuperação de objetos em um banco de dados relacional. CRUD consiste em operações como criação, leitura, atualização e remoção, necessárias para a persistência de objetos. O Gerenciador de Tabelas permite o mapeamento de um objeto para sua tabela no banco e o Método de Mapeamento de Atributos efetua o mapeamento entre atributos do objeto e colunas das tabelas. Gerenciador de Conexão cuida da conexão do sistema com a base de dados.

Para o emprego dos sub-padrões, foi escolhido um sistema de Oficina Eletrônica, onde a principal funcionalidade é controlar o conserto de produtos eletrônicos. Foi feita uma primeira implementação desse sistema utilizando os cinco sub-padrões orientados por objetos e, logo em seguida, uma implementação utilizando técnicas de orientação por aspectos. Ao término do trabalho foi feita uma análise comparativa entre as implementações, ressaltando os benefícios da solução orientada por aspectos.

O sistema de Oficina Eletrônica em questão é composto de entidades persistentes com Cliente, Conserto do Aparelho, Tipo do Aparelho e Técnico, que conseqüentemente, se tornam classes do sistema. Para cada classe persistente foi adicionado o código referente ao sub-padrão Gerenciador de Tabelas. Observou-se o entrelaçamento e dispersão de código

ao empregar tal sub-padrão, o que desapareceu na implementação orientada por aspectos, permanecendo somente os atributos e métodos do código funcional.

A implementação do sistema usando sub-padrões orientados por aspectos deu origem a criação de código adicional significativo ao padrão. Foram necessários três aspectos abstratos (`AspectStructure`, `AbstractAspectConnection` e `AspectConnection`) que podem ser especializados para qualquer tipo de sistema e cinco aspectos específicos, um para cada classe definida como persistente. Os aspectos específicos de cada classe entrecortam chamadas a seus construtores. O aspecto `AspectConnection` entrecorta as `servlets` que compõem o sistema para efetuar a conexão com o banco. O aspecto `AspectStructure` insere os atributos e métodos referentes ao Gerenciador de Tabelas.

O trabalho mostrou que as classes do sistema ficaram livres de atributos e métodos de persistência eliminando problemas de entrelaçamento e dispersão de código e tornando as classes mais reusáveis, já que deixaram de ser dependentes do padrão. Além disso, buscou-se sempre generalizar os aspectos criados para aumentar seus níveis de reúso. No entanto, não foi possível essa generalização para as classes persistentes do sistema, tendo que ser criado um aspecto para cada uma. Tal criação quando aplicada de forma manual pode ser viável em pequenos sistemas, mas para sistemas maiores torna-se bastante tediosa. Tal problema poderia ser resolvido se fosse utilizada uma ferramenta capaz de gerar de forma automática estes aspectos específicos.

4.3 Arcabouço Proposto por Rashid e Chitchyan

Rashid e Chitchyan apresentam uma experiência na separação de requisitos de persistência de um sistema de Controle de Publicações usando técnicas de programação orientada por aspectos. A principal funcionalidade desse sistema é armazenar detalhes de publicações, como nome e email do autor, local de publicação, item de publicação (artigo, livro, etc), dentre outros. Além disso, mostra-se que os aspectos criados para o sistema de Controle de Publicações podem ser especializados para outros tipos de sistemas, possibilitando a criação de um arcabouço para implementação de persistência.

Os autores definiram as partes da persistência que são independentes do sistema por meio da criação de conjuntos de junção abstratos que devem ser concretizados por aspectos especializados. Basicamente dois aspectos são responsáveis pela modularização do código referente ao requisito de persistência: o aspecto concreto `ApplicationDatabaseAccess`, responsável por definir as classes persistentes, e o aspecto abstrato `DataBaseAccess`, responsável pelo estabelecimento de conexão com banco, controle de transação, sincronização dos objetos e recuperação dos dados. O controle de transação foi implementado usando recursos de reflexão computacional de Java e a recuperação dos dados foi implementada

por meio de recursos de reflexão computacional de AspectJ.

O arcabouço proposto modulariza todos sub-interesses relacionados com a implementação de persistência (controle de conexão, controle de transação, sincronização de objetos e recuperação dos dados). Porém, a principal desvantagem deste arcabouço é que o mesmo não possui uma ferramenta de geração de aspectos para especialização dos aspectos abstratos. Além disso, cabe ao programador a implementação dos métodos referentes a criação e atualização de registros no banco. Essa implementação poderia também ser feita por uma ferramenta de geração que extrairia as informações necessárias a partir das tabelas do banco de dados.

4.4 Outros Trabalhos

Hibernate é uma ferramenta para ambiente Java, responsável pelo mapeamento entre objetos de um sistema e sua representação em um banco de dados relacional [24]. Hibernate não cuida somente desse mapeamento, mas também fornece facilidades para recuperação de dados e pode reduzir o tempo de desenvolvimento de software.

Hibernate utiliza arquivos XML de configuração para estabelecer conexão com o banco de dados e para fazer o mapeamento entre objetos do sistema e sua representação com uma tabela no banco de dados. Para exemplificar a configuração de um arquivo XML observe o exemplo da classe `Cat` na Listagem 4.1, retirado de [24].

O arquivo de configuração correspondente para esta classe é apresentado na Listagem 4.2. Neste arquivo, é feito o mapeamento entre o objeto e seus atributos e a tabela e os campos. Após a configuração, já é possível o uso do requisito de persistência. O código apresentado na Listagem 4.3 representa a persistência de um objeto do tipo `Cat` no banco de dados.

Na listagem 4.3, a invocação de métodos para obter a sessão, abrir transação, persistir o objeto criado, atualizar por definitivo no dispositivo de persistência e fechar a sessão, são responsabilidades do usuário do Hibernate. A obtenção da sessão acontece na invocação do método `currentSession`, a abertura de transação ocorre em `beginTransaction`, a persistência é feita quando o método `save` é executado, as atualizações são persistidas na execução do método `commit` e por último o fechamento de sessão através do método `closeSession`. No entanto, a implementação desses métodos está embutida no arcabouço, ou seja, são *frozen spots* pertencentes ao Hibernate.

A grande desvantagem do Hibernate em relação a solução proposta nessa dissertação, é que o Hibernate não modulariza o requisito de persistência. Observe nas linhas 1, 2, 9, 10 e 11, requisitos transversais sendo entrelaçados às regras de negócio. Outro fator

```
1  public class Cat {
2
3      private String id;
4      private String name;
5      private char sex;
6      private float weight;
7
8      public Cat() {}
9      public String getId() {return id;}
10     private void setId(String id) {this.id = id;}
11     public String getName() {return name;}
12     public void setName(String name) {this.name = name;}
13     public char getSex() {return sex;}
14     public void setSex(char sex) {this.sex = sex;}
15     public float getWeight() {return weight;}
16     public void setWeight(float weight) {this.weight = weight;}
17
18 }
```

Listagem 4.1: Cat.java

importante é que o Hibernate não permite facilidade na alteração do modelo de dados. Qualquer alteração feita no modelo, seja, alterar tipos de atributos de tabelas do banco, adicionar ou remover atributos em uma tabela e adicionar ou remover tabelas, requer do usuário uma nova atualização do arquivo XML de configuração. Enquanto que na solução proposta nesta dissertação, qualquer alteração feita no modelo, basta gerar novamente o código para refletir tal alteração nos módulos de implementação do sistema. Além disso, Hibernate não possui uma ferramenta para configuração ou geração de código automática. Toda a configuração deve ser feita manualmente pelo desenvolvedor.

EJB (*Enterprise Java Beans*) é um módulo da arquitetura J2EE (*Java 2 Enterprise Edition*) [47] usado para o desenvolvimento de aplicações cliente/servidor [34]. *Enterprise Beans* são componentes que constituem a arquitetura EJB e são usados para encapsular a lógica de negócios da aplicação, ou seja, os requisitos funcionais. Um *enterprise bean* reside em um EJB *Container*. Um EJB *Container* fornece o ambiente de execução para *enterprise beans*, incluindo serviços como segurança, controle de transação, persistência e balanceamento de carga [11]. Esses serviços fornecidos pelo EJB *Container* facilitam o trabalho do programador, já que o mesmo não precisa se preocupar em implementá-los.

Persistência é um serviço implementado pelo *Container-Managed Persistence* (CMP) ou pelo *Bean-Managed Persistence* (BMP). Suponha que um desenvolvedor necessita atualizar

```
1  <?xml version="1.0"?>
2  <!DOCTYPE hibernate-mapping
3      PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
4      "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
5  <hibernate-mapping>
6      <class name="net.sf.hibernate.examples.quickstart.Cat" table="CAT">
7          <id name="id" type="string" unsaved-value="null" >
8              <column name="CAT_ID" sql-type="char(32)" not-null="true"/>
9              <generator class="uuid.hex"/>
10         </id>
11         <property name="name">
12             <column name="NAME" length="16" not-null="true"/>
13         </property>
14         <property name="sex"/>
15         <property name="weight"/>
16     </class>
17 </hibernate-mapping>
```

Listagem 4.2: Arquivo hibernate.cfg.xml

objetos que estão *dirty*, ou seja, precisam ser persistidos no dispositivo de persistência. Atualizações automáticas podem ser executadas no banco de dados pelo serviço CMP disponibilizado pelo EJB *Container*. Para assegurar que isso aconteça os objetos devem ser definidos como *entity beans* [47]. Esses *beans* são mapeados para tabelas de um banco de dados relacional por meio de um arquivo XML de configuração.

Assim, percebe-se que o código de uma aplicação desenvolvida em J2EE facilita o trabalho do programador, já que o mesmo passa a se preocupar somente com requisitos funcionais. Além disso, uma aplicação desenvolvida nessa arquitetura possui um código mais modular do que as desenvolvidas convencionalmente, visto que o código que seria responsável pela persistência é retirado das classes.

EJB é classificado como um arcarbouço caixa preta, ou seja, suporta extensibilidade por definição de interfaces. Por um lado, arcabouços deste tipo fazem com que o desenvolvedor não precise ter um profundo conhecimento sobre a estrutura interna do arcabouço, facilitando o seu uso. Por outro lado, não utilizam herança e são inflexíveis na utilização dos serviços disponibilizados. O *container* EJB não é programável, ou seja, o serviços disponibilizados não podem ser facilmente modificados pelo usuário. Por exemplo, o serviço CMP usa banco de dados relacional como dispositivo de persistência. EJB não fornece mecanismo para que o usuário possa modificar o dispositivo de persistência de acordo com a sua necessidade. Além disso, EJB não fornece mecanismos para configuração ou geração de código automática. Toda a configuração de persistência deve ser feita manualmente pelo desenvolvedor.

```
1  Session session = HibernateUtil.currentSession();
2  Transaction tx= session.beginTransaction();
3
4  Cat princess = new Cat();
5  princess.setName("Princess");
6  princess.setSex('F');
7  princess.setWeight(7.4f);
8
9  session.save(princess);
10 tx.commit();
11 HibernateUtil.closeSession();
```

Listagem 4.3: Persistência de Um Objeto

Outra desvantagem é que o EJB não permite facilidade na alteração do modelo de dados. Qualquer alteração no modelo, requer do usuário uma nova atualização do arquivo XML de configuração e nos *entity beans*. Enquanto que na solução proposta nesta dissertação as alterações feitas no modelo de dados apenas requerem nova geração do código para refletir tal alteração nos módulos de implementação do sistema.

AspectJ2EE [11] é uma linguagem orientada por aspectos direcionada para a implementação de servidores de aplicações J2EE reconfiguráveis. O intuito de AspectJ2EE é permitir que o *container* EJB seja menos rígido, mais flexível e extensível. Na versão tradicional do EJB, quando um desenvolvedor deseja adicionar algum requisito transversal, para o qual EJB não fornece suporte, a implementação desse requisito ocasiona dispersão e entrelaçamento de código. A idéia é fornecer uma extensão das funcionalidades do *container* EJB, de tal forma que a adição de uma funcionalidade não prejudique a modularização. Usando AspectJ2EE, os serviços da plataforma J2EE são substituídos por um biblioteca de aspectos. Esses serviços tornam-se flexíveis e extensíveis.

As principais diferenças de AspectJ2EE para AspectJ são apresentadas a seguir:

- os aspectos de AspectJ podem ser aplicados a qualquer classe Java, enquanto que os aspectos de AspectJ2EE podem ser aplicados somente a *enterprise beans*;
- a linguagem AspectJ2EE tem suporte específico para a composição de aspectos. O processo de *weaving* é diferente do AspectJ. AspectJ2EE utiliza os mesmos mecanismos empregados pelos servidores de aplicação J2EE para combinar serviços com a lógica de negócio dos *enterprise beans*;
- AspectJ2EE permite modularizar requisitos transversais que impactam simultaneamente módulos cliente e servidores de uma aplicação distribuída. Por exemplo,

AspectJ2EE disponibiliza o serviço de criptografia, isso impacta o cliente (que tem que criptografar as mensagens) e o servidor (que tem que realizar o processo inverso).

AspectJ2EE emprega uma sintaxe semanticamente equivalente a do arquivo XML de configuração usado para especificar o serviço EJB. Um exemplo desse arquivo é mostrado na Listagem 4.4.

```
1  <entity id="Account">
2    <ejb-name>Account</ejb-name>
3    <home>aspectj2ee.demo.AccountHome</home>
4    <remote>aspectj2ee.demo.Account</remote>
5    <ejb-class>aspectj2ee.demo.AccountBean</ejb-class>
6    <aspect>
7      <aspect-class>aspectj2ee.core.Lifecycle</aspect-class>
8    </aspect>
9    <aspect>
10     <aspect-class>aspectj2ee.core.Persistence</aspect-class>
11     <value name="primKeyClass">java.lang.String</value>
12     <value name="primKeyField">id</value>
13     <value name="fieldMap">id:ID ,balance:BALANCE</value>
14   </aspect>
15   <aspect>
16     <aspect-class>aspectj2ee.core.Security</aspect-class>
17     <pointcut name="secured">execution(*(..))</pointcut>
18     <value name="requiredRole">User</value>
19   </aspect>
20   <aspect>
21     <aspect-class>aspectj2ee.core.Transactions</aspect-class>
22     <value name="reentrant">false</value>
23     <pointcut name="requiresnew">execution(deposit(..)) ||
24                                  execution(withdraw(..))</pointcut>
25     <pointcut name="required">execution(*(..)) &&
26                               !requiresnew()</pointcut>
27   </aspect>
28 </entity>
```

Listagem 4.4: Deployment Descriptor AspectJEE

Observe na listagem que `Lifecycle`, `Persistence`, `Security` e `Transactions` são aspectos aplicados ao *bean* `Account`. A idéia desse novo *deployment descriptor* é permitir que serviços como persistência, segurança, controle de transações e ciclo de vida do *bean* sejam configuráveis e novos serviços sejam adicionados.

O propósito de AspectJ2EE é unir a programação orientada por aspectos com J2EE. AspectJ2EE foi construída a partir da experiência e das técnicas de programação normal-

mente empregadas em J2EE. Entretanto trata-se de uma nova linguagem de programação orientada por aspectos, com o mesmo nível de complexidade de AspectJ. Além disso, AspectJ2EEE foi projetada para implementação de qualquer requisito não-funcional que seja transversal. A solução proposta nesta dissertação, por outro lado, foi projetada especificamente para apoiar a implementação de persistência. Portanto, o seu uso e aprendizado são mais simples e rápidos.

4.5 Comentários Finais

Após a análise dos arcabouços orientados por aspectos para implementação de persistência, observou-se as seguintes deficiências:

- os arcabouços descritos para implementação de persistência por meio de aspectos demandam um esforço considerável de programação, exigindo a definição de diversos aspectos e interfaces manualmente. A implementação de tais aspectos é normalmente tediosa e, portanto, sujeita a erros;
- Hibernate não modulariza o requisito de persistência e não permite facilidade na alteração do modelo de dados. Qualquer alteração feita no modelo requer do usuário uma nova atualização do arquivo XML de configuração. Além disso, Hibernate não possui uma ferramenta para configuração ou geração de código automática. Toda a configuração de persistência deve ser feita manualmente pelo desenvolvedor;
- EJB é classificado como um arcarbouço caixa preta. Por um lado, arcabouços deste tipo fazem com que o desenvolvedor não precise ter um profundo conhecimento sobre a estrutura interna do arcabouço, facilitando o seu uso. Por outro lado, não utilizam herança e são inflexíveis na extensão dos serviços disponibilizados. Além disso, o EJB não permite facilidade na alteração do modelo de dados. Qualquer alteração no modelo, requer do usuário uma nova atualização do arquivo XML de configuração. Por último, o EJB também não fornece mecanismos para configuração ou geração de código automática. Toda a configuração deve ser feita manualmente pelo desenvolvedor;
- AspectJ2EE é uma nova linguagem de programação orientada por aspectos, com o mesmo nível de complexidade de AspectJ. Além disso, AspectJ2EEE foi projetada para implementação de qualquer requisito não-funcional que seja transversal, tornando o seu uso e aprendizado mais complexo e lentos.

Com intuito de minimizar as deficiências nos arcabouços orientados por aspectos, os capítulos seguintes descrevem o GAP. O GAP é um arcabouço que lida com o problema de

modularização encontrado na implementação do requisito de persistência e automatiza o processo de especialização dos aspectos abstratos, por meio de uma ferramenta de geração de código.

Capítulo 5

GAP: Um Arcabouço para Implementação de Persistência

5.1 Arcabouço Proposto

O arcabouço orientado por aspectos proposto nesta dissertação, chamado de GAP (*Generator of Aspect for Persistence*), é responsável pela implementação de persistência de dados de sistemas que utitilizam bancos de dados relacionais. O requisito de persistência pode ser dividido nos seguintes sub-interesses: controle de conexões, controle de transações, controle de sincronização de objetos e recuperação de dados. Persistência de dados é um requisito que possui comportamento transversal e conseqüentemente, ocasiona os problemas de dispersão e entrelaçamento de código. Para mostrar o quanto o código de persistência se dispersa e entrelaça no código funcional de um sistema, observe o exemplo da classe `Cliente`, na Listagem 5.1.

A classe `Cliente` possui código referente a sincronização do objeto nas linhas de 14 a 16, prejudicando a coesão modular, além do código para recuperação de dados nas linhas de 19 a 21, com os métodos como `findByPrimaryKey`, `findAll`, dentre outros. A implementação desses métodos torna-se trabalhosa quando feita da forma convencional, ou seja, quando é necessário manipular diretamente classes e métodos responsáveis por essa recuperação, como por exemplo a classe `ResultSet` da API de Java.

Outro exemplo de dispersão e entrelaçamento de código se encontra na classe `ControladorCliente` na Listagem 5.2, onde o método `cadastrarCliente` contém código referente ao controle de conexão e transação nas linhas 5 e 9.

```
1  public class Cliente implements Persistent{
2      ...
3      /* Contrutora responsavel pela recuperacao dos dados*/
4      public Cliente(ResultSet rs){..}
5
6      public String getNome(){..}
7      public void setNome(String nome){..}
8      public String getCpf(){..}
9      public void setCpf(String cpf){..}
10     public String getEmail(){..}
11     public void setEmail(String email){..}
12
13     /*metodos de insercao, atualizacao e remocao*/
14     public void create(..){..}
15     public void update(..){..}
16     public void delete(..){..}
17
18     /*metodos para recuperacao dos dados*/
19     public Cliente findByPrimaryKey(..){..}
20     public Cliente findByCpf(..){..}
21     public ArrayList findAll(..){..}
22     ...
23 }
```

Listagem 5.1: Cliente.java

A fim de resolver o problema de modularização de persistência e facilitar a construção de métodos para a recuperação de dados, auxiliando o engenheiro de software na construção de sistemas, o GAP fornece as seguintes funções:

- mecanismos para o controle de conexão com o banco de dados;
- mecanismos para o controle de transação garantindo a integridade dos dados;
- mecanismos para a sincronização entre dados voláteis e dados persistentes;
- mecanismos para facilitar a recuperação de dados;
- geração automática, via uma ferramenta proposta neste trabalho, dos aspectos concretos que especializam os aspectos abstratos do GAP.

```
1  public class ControladorCliente{
2      ...
3      public static Cliente cadastrarCliente(..) throws Exception{
4          try{
5              beginTransaction(..);
6              Cliente cliente = clienteDAO.cadastrarCliente(..);
7              return cliente;
8          }finally{
9              endTransaction(..);
10         }
11     }
12     ...
13 }
```

Listagem 5.2: ControladorCliente.java

O GAP é composto por aspectos abstratos e classes que tem a função de controlar conexão, transação e sincronização dos objetos e facilitar a recuperação dos dados do banco. O aspecto abstrato `TransactionControlAspect` tem a função de garantir que uma mesma conexão será usada em toda a transação, fazer atualizações dos dados por definitivo no banco ao término da transação, via *commit* e desfazê-las, caso alguma atualização não puder ser feita, via *rollback*. O aspecto `SynchronizationControlAspect` tem a função de sincronizar os objetos das classes persistentes, ou seja classes que implementam a interface `Persistent`, na memória com a sua representação no banco de dados. Esse processo é feito por meio de conjuntos de junção abstratos que definem os pontos da aplicação alvo que devem ser instrumentados com código para, respectivamente, inserir uma nova linha em uma determinada tabela do banco de dados e para atualização de uma linha de uma tabela. A classe `PersistentRetrieval` tem a função de facilitar a recuperação de dados. Essa classe contém o método estático `getObjects` que recebe dois parâmetros, um String SQL para consulta e um String referente ao nome da classe responsável por encapsular os dados vindos do banco. Esse método automaticamente faz a consulta ao banco e retorna uma lista com todos os registros mapeados em objetos da classe passada como parâmetro.

A ferramenta de geração é responsável por gerar os aspectos concretos que especializam os aspectos abstratos do GAP. O aspecto concreto gerado para o controle de conexão e transação, implementa os conjuntos de junção abstratos que definem os métodos transacionais do sistema, ou seja, métodos que necessitam de controle de transação, e definem os métodos que obtêm a conexão com o banco. Para cada classe persistente do sistema, o gerador de aspectos cria um aspecto que estende `SynchronizationControlAspect` e que define seus conjuntos de junção abstratos e implementa os métodos `create`, `update`

e `delete` que tem a função de criar, atualizar e remover registros no banco. O gerador de aspectos cria ainda o aspecto `RetrievalAspect` que tem função de inserir construtores nas classes persistentes, que serão usadas pela classe `PersistentRetrieval`.

A Figura 5.1 apresenta a arquitetura do GAP juntamente com a ferramenta, destacando o seus *hot spots* [46], ou seja, pontos de especialização do GAP e o seu relacionamento com o usuário.

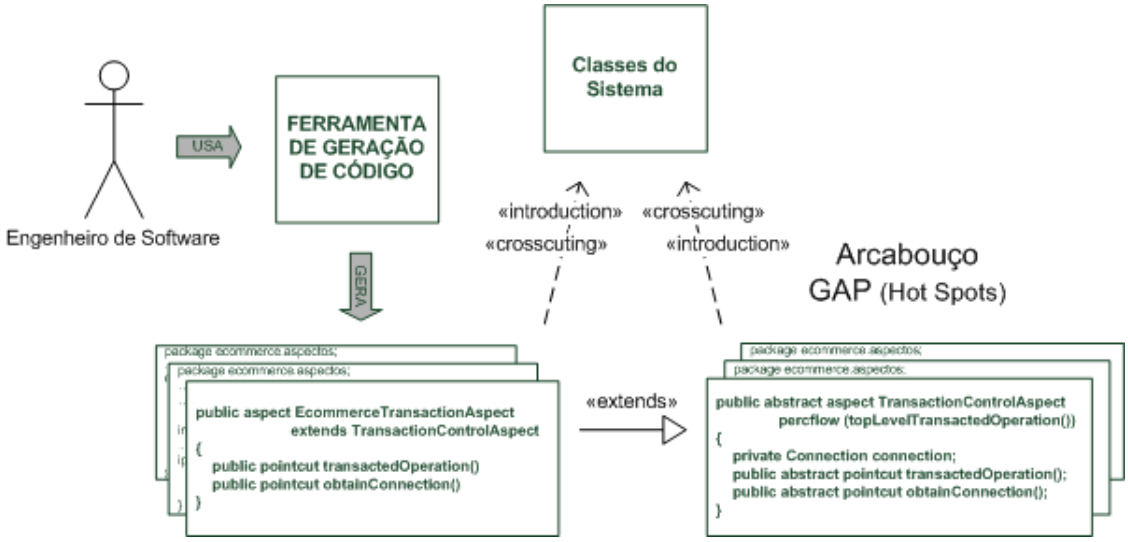

Figura 5.1: Arquitetura do GAP

O engenheiro de software usa a ferramenta de geração para importar o modelo de dados do banco, obtendo informações como tabelas, atributos, chaves primárias e estrangeiras. Logo em seguida, configura o modelo importado, mapeando classes e membros do sistema com as tabelas e atributos vindos do banco. Depois o engenheiro usa a ferramenta para definir o ponto do sistema onde deve-se estabelecer conexão com o banco de dados e os métodos transacionais do sistema. Feitas essas configurações, a ferramenta cria os aspectos para se ligarem aos *hot spots* pertencentes ao GAP. Esses aspectos se interagem com as classes funcionais do sistema provendo a funcionalidade de persistência.

As seções seguintes descrevem detalhamente cada aspecto abstrato pertencente ao GAP e como esses aspectos implementam o requisito de persistência.

5.2 Aspecto para Controle de Conexão e Transação

Controle de conexão e transação são sub-requisitos de persistência que estão dispersos e entrelaçados pelo código de um sistema cujo dispositivo de persistência é um banco de dados relacional. Para controlar a conexão com o banco, o desenvolvedor deve estabelecer a conexão, garantir que essa conexão será usada em toda a transação e fechar essa conexão ao término da transação. Para o controle de transação, o desenvolvedor deve se preocupar com atualização dos dados por definitivo no banco, via *commit* e desfazer essas atualizações, caso alguma atualização não puder ser feita, via *rollback*. Tal requisito, quando implementado se encontra disperso e entrelaçado por todo o código. Conseqüentemente, existe a necessidade de modularização desses sub-requisitos e de um módulo de implementação que forneça ao desenvolvedor recursos para esse controle. Assim, o desenvolvedor somente precisaria de informar em qual ponto do sistema é necessário o controle de conexão e transação.

No GAP, um aspecto abstrato para controle de conexão e transação interfere na execução de um sistema orientado por objetos de forma a garantir que uma conexão será usada por toda a transação e que ao término da transação essa conexão será fechada. Além disso, este aspecto é responsável por definir os métodos da aplicação alvo que são transacionais, isto é, métodos cuja execução deve preservar as propriedades ACID [20] (atomicidade, consistência, independência e durabilidade) ao atualizar o estado de objetos persistentes. Basicamente, antes de iniciar a atualização do estado, tais métodos devem se comunicar com o banco de dados para informar o início da transação. Caso a atualização seja efetuada com sucesso, eles devem comunicar tal fato ao gerenciador de banco de dados (*commit*). Caso contrário, deve-se comunicar o insucesso da atualização (*rollback*). Além disso, todas as atualizações devem ser realmente persistidas no banco quando um método *top-level*, ou seja, o primeiro método transacional no fluxo de controle da operação que está sendo executada, for finalizado com sucesso. Se alguma atualização falhar ou se algum método de negócio lançar alguma exceção, as atualizações necessárias até o momento serão desfeitas, preservando as propriedades ACID das informações.

A solução para implementação de controle de conexão e transação foi baseada em uma solução descrita por Ramnivas Laddad [30]. Os requisitos descritos acima foram implementados da seguinte forma:

- para prover uma implementação reusável, foi criado um aspecto abstrato que implementa grande parte da lógica de controle de conexão e transação. Para um sistema utilizar o controle de conexão e transação, basta implementar um aspecto que especializa esse aspecto abstrato;
- para permitir que sub-aspectos especializados definam os métodos que necessitam de controle de conexão e transação, foram incluídos conjuntos de junção abstratos,

que definem métodos para estabelecer conexão com o banco de dados e os métodos transacionais do sistema;

- para capturar métodos *top-level*, foi definido um conjunto de junção que usa o recurso `cflowbelow` de AspectJ. Esse recurso tem a função de capturar todos pontos de junção no fluxo de controle de algum método, excluindo os pontos de junção para esse método. Se esse recurso for negado, usando o operador `!`, todos os pontos de junção no fluxo de controle de algum método serão desconsiderados, e serão válidos somente os pontos de junção desse método. Dessa forma, o método *top-level* será capturado;

- para verificar o sucesso, ou não da execução de todos os métodos na transação foi criado uma regra de junção `around` para o método *top-level* com um bloco `try/catch` que irá tratar exceções levantadas por esse método e por qualquer método no seu fluxo de controle;

- para garantir o uso do mesmo objeto de conexão para todas as atualizações em uma transação, foi definido um ponto de junção que cria a conexão no fluxo de controle da transação. O método que obtém uma conexão é capturado e se for a primeira vez que é solicitada a obtenção da conexão, a conexão é criada e armazenada no aspecto. Para todas as outras requisições de conexão, a regra de junção simplesmente retorna a instância do objeto armazenada no aspecto.

- Para acomodar os possíveis métodos que criam uma conexão, foi criado um ponto junção abstrato, que deve ser concretizado pelos aspectos gerados pela ferramenta.

Apresentam-se nas Listagens 5.3 e 5.4, o aspecto abstrato que é usado para capturar os pontos de junção essenciais ao estabelecimento de conexões e os métodos que necessitam de controle de transação.

O aspecto é associado com cada transação *top-level* no fluxo de controle por meio do construto `percflow` (linha 2). Esta associação força a criação de uma nova instância do sub-aspecto para cada invocação de um método especificado pelo conjunto de junção `topLevelTransactedOperation()` (linha 7). Isso é fundamental para o funcionamento do controle de transação e conexão, uma vez que a instância do objeto `connection` não pode ser única para todo o sistema. Caso isso ocorra, transações podem se conflitar, ocasionando inconsistência no banco.

Os aspectos concretos devem implementar os conjuntos de junção abstratos `transactedOperation()` e `obtainConnection()` (linhas 5 e 6), sendo que `transacted-Operation()` define os métodos transacionais do sistema e `obtainConnection()` define os métodos que obtêm a conexão. O conjunto de junção `topLevelTransactedOperation()`

```
1  public abstract aspect TransactionControlAspect
2      percflow (topLevelTransactedOperation()){
3
4      private Connection connection;
5      public abstract pointcut transactedOperation();
6      public abstract pointcut obtainConnection();
7      protected pointcut topLevelTransactedOperation():
8          transactedOperation() && !cflowbelow(transactedOperation());
9
10     Object around() throws RuntimeException:
11                 topLevelTransactedOperation(){
12         Object result;
13         try{
14             result = proceed();
15             if (connection != null){connection.commit();}
16         } catch (Exception e){
17             if (connection != null){connection.rollback();}
18             throw new RuntimeException(e);
19         } finally{
20             if (connection != null){connection.close();}
21         }
22         return result;
23     }
```

Listagem 5.3: TransactionControlAspect.aj - Parte 1

é responsável por definir o método *top-level* no fluxo de controle de uma transação usando o conjunto de junção abstrato `transactedOperation()`. Assim que um ponto de junção do conjunto de junção `topLevelTransactedOperation()` for capturado, uma nova instância do aspecto concreto é criada. A regra de junção `around` para o conjunto de junção `topLevelTransactedOperation()` executa os métodos capturados dentro de um bloco `try/catch`. No bloco `try` (linha 13) essa regra de junção simplesmente procede com a execução por meio do `proceed()` (linha 14). Se algum método lançar uma exceção, o bloco `catch` é executado e todas as atualizações feitas até o momento são desfeitas por meio do `rollback()`. Os comandos (linhas 15, 17 e 20) garantem que a conexão com o dispositivo de persistência não é criada para requisitos funcionais que não necessitam de atualizações. No bloco `finally` (linha 19), as conexões são fechadas. A regra de junção `around` para o conjunto de junção `obtainConnection()` (linha 22) garante que uma mesma conexão será usada para todas as atualizações e que essas atualizações são persistidas quando o método `commit()` for invocado. A regra de junção verifica se a instância `connection` é nula; se

```
24      Connection around() throws SQLException
25          : obtainConnection() && cflow(transactedOperation()){
26          if (connection == null){
27              connection = proceed();
28              connection.setAutoCommit(false);
29          }
30          return connection;
31      }
32      private static aspect SoftenSQLException{
33          declare soft : java.sql.SQLException
34          : (call(void Connection.rollback())
35          || call(void Connection.close()))
36          && within(TransactionControlAspect);
37      }
38      pointcut illegalConnectionManagement()
39          : (call(void Connection.close())
40          || call(void Connection.commit())
41          || call(void Connection.rollback())
42          || call(void Connection.setAutoCommit(boolean)))
43          && !within(TransactionControlAspect);
44
45      declare error: illegalConnectionManagement():
46          "Transaction and connection control must be make" +
47          "only by TransactionControlAspect.";
48  }
```

Listagem 5.4: TransactionControlAspect.aj - Parte 2

for, significa que é necessário abrir uma nova conexão; senão a conexão já existente é retornada para o solicitante. O enfraquecimento da exceção `java.sql.SQLException` (linha 32) evita a necessidade de `try/catch` para a execução dos métodos `rollback` e `close`. Por último, nas linhas 38 a 47, é garantido que o gerenciamento da transação em todo o sistema será feito somente pelo aspecto `TransactionControlAspect`, evitando conflitos no gerenciamento de transação.

A Figura 5.2 apresenta o aspecto `TransactionControlAspect` com seus dois conjuntos de junção abstratos `transactedOperation()` e `obtainConnection()` e o conjunto de junção concreto `topLevelTransactedOperation()`. Anteriormente, `TransactionControlAspect` é associado com o conjunto de junção `topLevelTransactedOperation()` usando a especificação de associação `percflow`. Mostra-se ainda na figura um aspecto concreto, chamado `EcommerceTransactionControlAspect`, que implementa os conjuntos

de junção mencionados. O código deste aspecto será gerado pela ferramenta descrita no Capítulo 6.

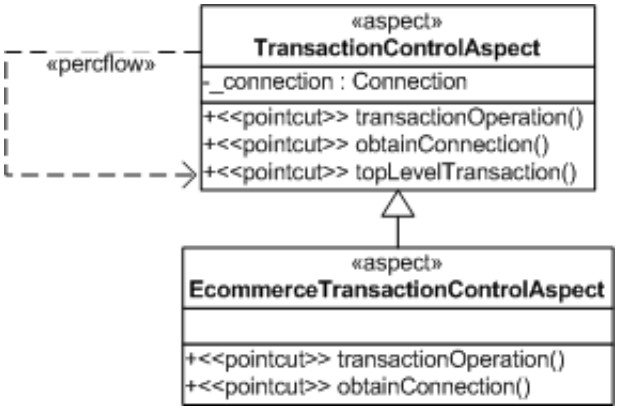

Figura 5.2: Controle de Conexão e Transação para um Sistema de Comércio Eletrônico

O aspecto abstrato descrito modulariza o código referente a controle de conexão e transação. Além disso, esse aspecto é um módulo de implementação que fornece ao desenvolvedor recursos para implementação desse controle. Conseqüentemente, o desenvolvedor passa a não se preocupar com a implementação desses sub-requisitos, podendo se concentrar na implementação de requisitos funcionais. Com efeito, aumenta-se a produtividade do desenvolvedor e diminuem-se os erros na lógica dos requisitos funcionais.

5.3 Aspecto para Sincronização de Objetos

O controle de sincronização de objetos é um sub-requisito de persistência que está disperso e entrelaçado pelo código de um sistema cujo dispositivo de persistência é um banco de dados relacional. Para sincronizar os objetos da memória com o dispositivo de persistência, o desenvolvedor deve criar métodos como `create`, `update` e `delete` para classes persistentes do sistema. A implementação e invocação desses métodos aparecem dispersas e entrelaçadas por todo o código. Assim, existe a necessidade de modularização desse sub-requisito e de um módulo de implementação que forneça ao desenvolvedor recursos para essa sincronização.

A geração de aspectos para sincronização entre objetos na memória e sua representação no banco de dados consiste em definir primeiramente quais as classes da aplicação são classes de persistência. Por exemplo, para um sistema de comércio eletrônico possivelmente as classes de persistência são `Cliente`, `Pedido`, `ItemPedido`, `Produto`, etc. Todas as classes

persistentes devem implementar uma interface chamada `Persistent`. O gerador de aspectos disponibilizará uma interface gráfica para que sejam definidas as classes persistentes da aplicação, além de fazer um mapeamento de uma classe persistente com sua tabela correspondente no modelo de dados.

O aspecto abstrato nas Listagens 5.5 e 5.6 é usado para capturar conjuntos de junção essenciais à sincronização de objetos:

```
1  public abstract aspect SynchronizationControlAspect
2  {
3      private Set dirtyEntities = new HashSet();
4
5      public abstract pointcut create(Persistent persistent);
6      public abstract pointcut update(Persistent persistent);
7      public abstract pointcut executeUpdate();
8      public pointcut detectDeletedObject(Persistent p): this(p) &&
9              (execution(public * Persistent.get*(..)) ||
10             execution(public * Persistent.set*(..)));
11
12     after(Persistent persistent)
13         throws RuntimeException: create(persistent){
14         try{
15             persistent.create();
16         }catch(Exception e){
17             throw new RuntimeException(e);
18         }
19     }
20     after(Persistent persistent)
21         throws RuntimeException: update(persistent){
22         try{
23             dirtyEntities.add(persistent);
24         }catch(Exception e){
25             throw new RuntimeException(e);
26         }
27     }
```

Listagem 5.5: SynchronizationControlAspect.aj - Parte 1

Os conjuntos de junção abstratos `create` e `update` (linhas 5 e 6) definem os pontos da aplicação alvo que devem ser instrumentados com código para, respectivamente, inserir uma nova linha em uma determinada tabela do banco de dados e para atualização de uma linha de uma tabela. No primeiro caso, tais pontos normalmente correspondem a instan-

```
28      after() throws RuntimeException: executeUpdate() {
29          Iterator iterator = dirtyEntities.iterator();
30          while(iterator.hasNext()) {
31              Persistent p = (Persistent) iterator.next();
32              try {
33                  p.update();
34              }catch(Exception e){ throw new RuntimeException(e);}
35              finally{iterator.remove();}
36          }
37      }
38      public boolean Persistent.isDeleted = false;
39      public boolean Persistent.isDeleted(){return this.isDeleted;}
40
41      before(Persistent persistent)
42          throws RuntimeException: detectDeletedObject(persistent){
43          if(persistent.isDeleted()){
44              throw new RuntimeException
45              ("It is impossible to have access to a deleted object.");
46          }
47      }
48  }
```

Listagem 5.6: SynchronizationControlAspect.aj - Parte 2

ciações, via operador `new`, de objetos de classes persistentes. O segundo caso corresponde a chamadas de métodos de nome `set*` de classes persistentes.

Ao conjunto de junção `create` associa-se uma regra do tipo `after` (linha 12), a qual simplesmente invoca um método de nome `create` tendo como alvo o objeto que foi instanciado. A razão para se usar `after` consiste em persistir no banco de dados uma instância de um objeto persistente após essa instância ter sido criada. A implementação do método `create` é gerada automaticamente pelo sistema e é adicionada à classe do respectivo objeto persistente usando-se os recursos de declaração inter-tipos de AspectJ. Esta implementação contém o código responsável por emitir um comando SQL para inserção de linhas em uma tabela de um banco de dados relacional.

Ao conjunto de junção `update` associa-se uma regra do tipo `after` (linha 19), a qual simplesmente adiciona o objeto a um conjunto de objetos que ainda não foram sincronizados com o banco de dados. Este conjunto é chamado de `dirtyEntities`. Após a execução de métodos transacionais (regra `executeUpdate`) (linha 28) este conjunto é percorrido invocando-se o método `update` de cada um de seus elementos. A implementação deste

método é gerado automaticamente pelo sistema e é adicionado à classe do respectivo objeto persistente. Esta implementação contém o código responsável por emitir um comando SQL para atualização de uma linha em uma tabela de um banco de dados relacional.

A remoção de objetos em um sistema é uma funcionalidade de grande importância que pode ser modularizada e automatizada usando aspectos. No entanto, devido ao processo automático de coleta de lixo, não existe a noção da remoção de um objeto em Java. Conseqüentemente, não existe um ponto de referência disponível para que algum aspecto responsável pela remoção possa remover a representação do objeto no banco de dados. Portanto, a função de remoção não fica transparente (*obliviousness*) para o programador, ou seja, é necessário que o programador defina explicitamente o momento que o objeto deve ser removido.

Para que a função de remoção possa ser modularizada e automatizada em um aspecto, é necessário criar um ponto de referência para a remoção no sistema. No GAP, esse ponto de referência é fornecido pelo método `delete()` que é inserido nas classes persistentes usando os recursos de transversalidade estática de AspectJ. Esse método é invocado por instâncias de classes persistentes. Além disso, o aspecto `SynchronizationControlAspect` é responsável por inserir o membro `isDeleted` (linha 38) e o método `isDelete` (linha 39) na classe persistente. Esse método é responsável pela remoção da representação de um objeto persistente no banco, garantindo que um objeto já removido não vai ser utilizado, mesmo que o coletor de lixo ainda não o tenha removido da memória.

O conjunto de junção `detectedDeletedObjects` (linha 8) identifica pontos de acesso a membros e métodos de objetos persistentes. A regra de junção `before` (linha 39) para esse conjunto, é responsável por garantir que um objeto já removido não vai ser acessado.

Para cada classe persistente da aplicação, o gerador de aspectos gera um aspecto que estende `SynchronizationControlAspect` e que define seus conjuntos de junção abstratos. Além disso, os aspectos gerados implementam os métodos `create`, `update` e `delete`. Caso o GAP não incluísse um gerador de aspectos, estes sub-aspectos deveriam ser implementados de forma manual, em um processo tedioso, que consumiria um tempo considerável, e sujeito a erros.

A Figura 5.3 apresenta o aspecto `SynchronizationControlAspect` com seus cinco conjuntos de junção abstratos `create()`, `update()`, `executeUpdate()` e `detectDeletedObject()`. O sub-aspecto `ClientSynchronizationControlAspect`, que é criado pela ferramenta de geração descrita no Capítulo 6, contém quatro conjuntos de junção concretos que fornecem a definição para os conjuntos de junção do aspecto base.

O aspecto abstrato descrito modulariza o código referente a sincronização de objetos. Além disso, esse aspecto consiste de um módulo de implementação que fornece ao desenvolvedor recursos para implementação da sincronização, pela introdução de métodos como

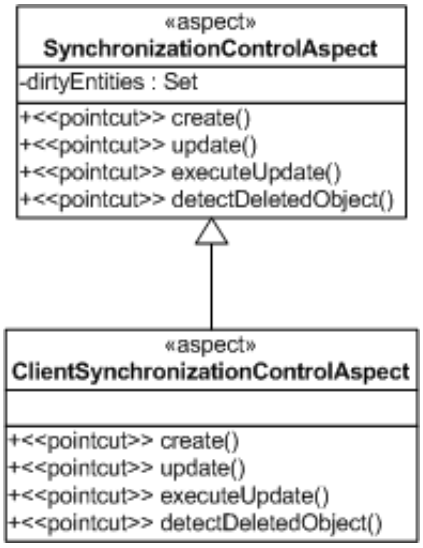

Figura 5.3: Sincronização de Objetos para a Entidade Client

`create`, `update` e `delete`, nas classes persistentes do sistema. Conseqüentemente, o desenvolvedor passa a não se preocupar com a implementação desse sub-requisito, podendo se concentrar na implementação dos módulos de requisitos funcionais.

5.4 Aspecto para Recuperação de Dados

Aspectos também podem ser importantes na modularização do código responsável pela recuperação dos dados em um banco de dados relacional. Um dos requisitos mais utilizados em um sistema que utiliza banco de dados é a recuperação dos dados a partir de linguagens de pesquisa como SQL. Por meio da definição de condições de seleção, um registro de uma tabela no banco de dados pode ser recuperado e mapeado em um objeto.

O GAP via ferramenta de geração de aspectos e de *frozen spots*, disponibiliza recursos para que a recuperação dos dados possa ser feita.

O GAP possui uma classe chamada de `PersistentRetrieval` descrita na Listagem 5.7. Esta classe contém o método estático `getObjects` responsável pela recuperação dos dados. Esse método recebe dois parâmetros, um String SQL para consulta e um String referente ao nome da classe responsável por encapsular os dados vindos do banco. Esse método automaticamente faz a consulta ao banco e retorna uma lista com todos os registros mapeados em objetos da classe passada como parâmetro. A invocação do método estático

`getCollection` (linha 9) tem a função de mapear cada registro do banco em um objeto da classe.

```
1  public class PersistentRetrieval {
2      public static ArrayList getObjects(String sql, String classe)..{
3          Connection connection = MyConnectionFactory.getConnection();
4          PreparedStatement stmt = null; ResultSet rs = null;
5          ArrayList arrayList = null;
6          try{
7              stmt = connection.prepareStatement(sql);
8              rs = stmt.executeQuery();
9              arrayList = getCollection(rs, classe);
10             rs.close();
11         } catch (SQLException e){throw(e)}finally{connection.close();}
12         return arrayList;
13     }
14     public static ArrayList getCollection(ResultSet rs, String c)..{
15         Object obj = null; ArrayList arrayObjects = new ArrayList();
16         try{
17             Constructor constructor =
18                 Class.forName(c).getConstructor(
19                         new Class []{ResultSet.class});
20             while (rs.next()) {
21                 obj = constructor.newInstance(new Object []{rs});
22                 arrayObjects.add(obj);
23             }
24         } catch(Exception e){throw(e);}
25         return arrayObjects;
26     }
27 }
```

Listagem 5.7: PersistentRetrieval.java

O método `getConnection` foi construído usando os recursos de reflexão computacional de Java. Esse método obtém uma classe a partir do String passado como parâmetro no método `forName` (linha 18). Após a obtenção da classe, o método construtor é obtido usando `getConstructor`. A construtora da classe é invocada pelo método `newInstance` da classe `Constructor` (linha 21), o qual retorna o objeto construído. Esse objeto é adicionado à lista de objetos retornados.

Nesse contexto, a ferramenta de geração de aspectos é responsável pela criação de um aspecto chamado `RetrievalAspect`, cujo o código é mostrado na Listagem 5.8. Esse

aspecto tem a função de inserir a construtora que recebe como parâmetro o `ResultSet` dentro de uma classe persistente (linha 4). Por meio do `ResultSet`, é possível mapear um registro do banco em um objeto persistente. A criação desse aspecto e dessa construtora se torna viável a partir de configurações iniciais feitas no GAP, onde cada atributo da tabela no banco de dados é mapeado para um atributo de uma classe. O Capítulo 6 descreve os detalhes dessa configuração.

```
1  public aspect RetrievalAspect {
2      ...
3      public void Cliente.Cliente(ResultSet rs) throws Exception{
4      {
5          this.id(rs.getString("ID_CLIENTE"))
6          this.cpf(rs.getString("CPF_CLIENTE"));
7          this.nome(rs.getString("NOME_CLIENTE"));
8          this.email(rs.getString("EMAIL_CLIENTE"));
9          this.senha(rs.getString("SENHA_CLIENTE"));
10         this.desEnd(rs.getString("DES_END_CLIENTE"));
11     }
12     ...
13 }
```

Listagem 5.8: RetrievalAspect.aj

Com a classe `PersistentRetrieval` e o aspecto `RetrievalAspect` é possível implementar de forma simples métodos para recuperação de dados. A implementação desses métodos torna-se simples para o desenvolvedor, pelo fato de ele não ter que manipular diretamente classes responsáveis pela recuperação de dados, como `ResultSet`. Um exemplo de implementação do método `findByCity`, referente a classe `Cliente` é apresentado na Listagem 5.9.

Outro recurso importante que o GAP disponibiliza é a recuperação de dados, cuja representação em objetos é feita usando hierarquia de classes. Em [2], são propostas quatro técnicas básicas para se mapear hierarquia de classes em tabelas de um banco de dados relacional:

- mapear todas as classes da hierarquia para uma única tabela;
- mapear cada classe concreta para a sua própria tabela;
- mapear cada classe para a sua própria tabela;
- mapear as classes da hierarquia dentro de uma estrutura genérica de tabelas.

```
1  public static ArrayList findByCity() throws Exception{
2      ArrayList listaClientes = null;
3      try {
4          listaClientes = PersistentRetrieval.
5              getObjects("select * from cliente"+
6                         "where ds_end_cliente like '%Belo Horizonte%'",
7                         "ecommerce.business.cliente.Cliente");
8      }catch (Exception e){throw(e);}
9      return listaClientes;
10 }
```

Listagem 5.9: Cliente - findAll

A técnica usada pelo GAP é o mapeamento de cada classe para a sua própria tabela. Essa técnica foi escolhida pelo fato de as tabelas do banco representarem de forma real os objetos da aplicação, onde cada tabela só tem colunas para propriedades de uma classe. Além disso, se torna fácil alterar superclasses e adicionar subclasses e o espaço ocupado é diretamente proporcional ao número de objetos armazenados. Mapear todas as classes em uma única tabela pode ocasionar desperdício de espaço e tornar difícil indicar o tipo do objeto, além da tabela ficar com muitas colunas se a hierarquia de classes for extensa. Já mapear as classes dentro de uma estrutura genérica de tabelas pode se gerar um modelo de dados confuso e de difícil entendimento, além de tornar difícil a obtenção dos dados.

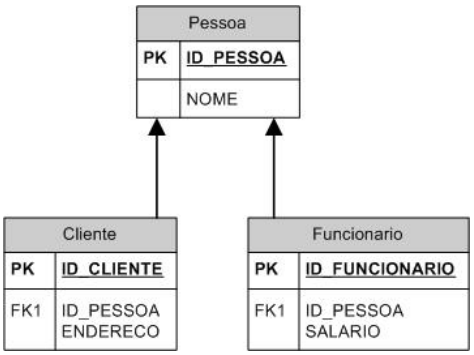

Figura 5.4: Modelo de Dados

Para mostrar como o GAP lida com herança, um exemplo com as classes `Cliente`, `Funcionario` e `Pessoa` é descrito a seguir. Suponha que `Funcionario` e `Cliente` estendam `Pessoa`, conforme mostrado na Figura 5.4. Para que o gerador possa introduzir por meio do aspecto `RetrievalAspect` as construtoras nas classes, deve-se realizar o mapeamento entre

cada tabela do banco e a classe. No Capítulo 6 mostra-se a interface gráfica do gerador responsável por realizar esta tarefa. Além disso, é necessário definir se uma classe herda da outra. No exemplo da Figura 5.4, as classes `Cliente` e `Funcionario` são definidas como classes filhas, portanto o código da construtora introduzido na classe conterá o comando `super`, que tem a função de executar a construtora da classe pai. Na Listagem 5.10 são apresentadas as construtoras das classes da hierarquia.

```
1  public Cliente(ResultSet rs) throws Exception
2  {
3      super(rs);
4      this.setId(rs.getString("ID_CLIENTE"));
5      this.setEndereco(rs.getString("ENDERECO"));
6  }
7  public Funcionario(ResultSet rs) throws Exception
8  {
9      super(rs);
10     this.setId(rs.getString("ID_FUNCIONARIO"));
11     this.setSalario(rs.getString("SALARIO"));
12 }
13 public Pessoa(ResultSet rs) throws Exception
14 {
15     this.setId(rs.getString("ID_PESSOA"));
16     this.setNome(rs.getString("NOME"));
17 }
```

Listagem 5.10: Construtoras Cliente, Funcionario e Pessoa

Os atributos dos objetos das classes `Cliente` e `Funcionario` são inicializados na suas respectivas construtoras, com exceção do atributo `nome` que é carregado na construtora da superclasse `Pessoa`.

5.5 Comentários Finais

Um arcabouço orientado por aspectos para modularização de persistência auxilia o engenheiro de software na construção de sistemas. O arcabouço proposto neste capítulo fornece mecanismos para controlar conexão com bancos de dados, controlar transações, persistir dados voláteis e facilitar a recuperação de dados. Além disso, embute um gerador de aspectos, o qual é responsável por criar de forma automática todos os aspectos concretos que especializam os aspectos abstratos do arcabouço.

Analisando as características do GAP pode se concluir que é um arcabouço orientado por aspectos, transversal, do tipo caixa branca, que possue *hot spots* e *frozen spots*. O GAP é orientado por aspectos, pelo fato de usar estruturas de aspectos em sua implementação e é transversal por possuir mecanismos de composição abstratos e variabilidades correspondentes a um único interesse transversal (persistência). Esse arcabouço é do tipo caixa branca, por possuir características como herança, amarração dinâmica e sobrescrever métodos como técnica de extensão. Os *hot spots* do GAP se encontram nos conjuntos de junção abstratos pertencentes aos aspectos abstratos do arcabouço. Os *frozen spots* se encontram dentro de partes concretas dos aspectos abstratos do arcabouço, como por exemplo as regras de junção.

É importante ressaltar que, em geral, usuários de arcabouços são os responsáveis pela ligação entre o sistema que está sendo desenvolvido e o arcabouço que está sendo utilizado. Ou seja, é função do usuário do arcabouço escrever o código que liga o sistema aos *hot spots* disponibilizados pelo arcabouço. Para libertar o usuário desta tarefa, ao GAP foi adicionado um facilitador para esse processo de ligação. Por meio de uma ferramenta de geração, o usuário consegue gerar o código para prover tal ligação. Basta o usuário configurar a ferramenta para que a mesma gere aspectos que ligam o sistema aos *hot spots* do GAP. No Capítulo 6 descreve-se este processo de automação de implementação de persistência por meio da ferramenta de geração.

Capítulo 6

Automação do Processo de Implementação de Persistência

6.1 Visão Geral

A ferramenta proposta para geração de aspectos é responsável pela criação de aspectos concretos, os quais especializam os aspectos abstratos do GAP que controlam conexão, transação e sincronização de objetos. A ferramenta de geração de aspectos também é responsável pela criação do aspecto concreto que cuida da recuperação dos dados e por criar classes e interfaces que cuidam do acesso ao banco de dados para inserir, atualizar ou remover algum registro. A Figura 6.1 descreve as entradas dessa ferramenta, o seu funcionamento a partir de um modelo de dados extraído de um banco de dados relacional e os aspectos gerados.

A ferramenta lê arquivos XML de configuração, preparados pelo usuário. Para a geração dos aspectos, o engenheiro de software, por meio da ferramenta, realiza as seguintes etapas:

- importação do modelo de dados do banco, para obter informações sobre tabelas, chaves primárias, chaves estrangeiras e atributos;
- configuração do modelo importado;
- configuração de métodos que estabelecem a conexão com o banco e definição dos métodos transacionais do sistema;
- geração dos aspectos, classes e interfaces.

A primeira etapa consiste na importação do modelo. Para a importação, o engenheiro de software deve informar dados para a conexão com o banco, como URL, *driver*, usuário e senha. Esses são armazenados no arquivo denominado `database.xml`. Uma vez informado esses dados, a ferramenta acessa o banco de dados e extrai os metadados como tabelas e atributos e armazena-os no arquivo `model.xml`. A configuração do modelo consiste em mapear classes e membros do sistema com as tabelas e atributos vindos do banco. Logo em seguida, é feita a definição dos pontos do programa onde deve-se estabelecer conexão com o banco de dados e definição dos métodos transacionais do sistema. Essas informações são armazenadas no arquivo denominado `connectiontransactioncontrol.xml`. Uma vez feita a configuração dos arquivos XML, o engenheiro utiliza a ferramenta de geração para ler as informações contidas nestes arquivos e gerar os aspectos.

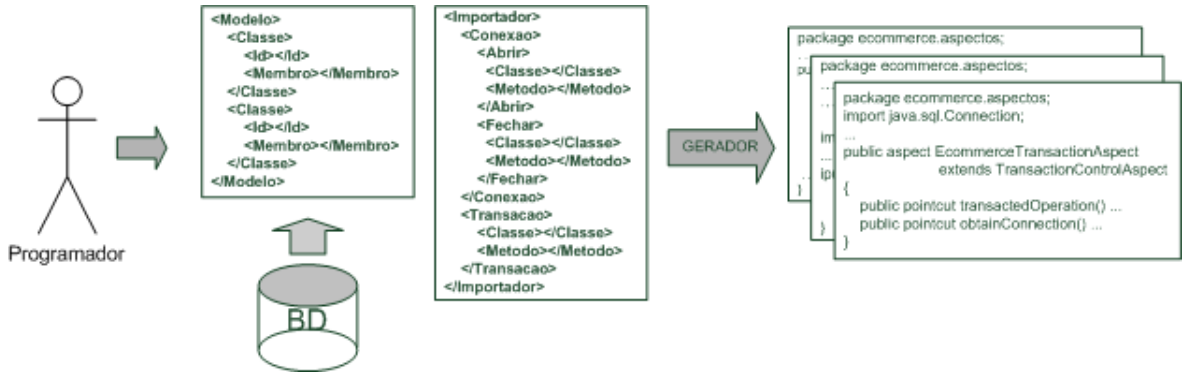

Figura 6.1: Etapas para Geração dos Aspectos, Classes e Interfaces

6.2 Configuração dos Arquivos XML

A geração do código somente é possível a partir da importação e configuração do modelo, definição dos pontos de controle de conexão e definição dos métodos transacionais. Toda essa configuração se faz necessária, pois os aspectos gerados contêm informações originadas do banco, como tabelas, campos, chaves primárias, chaves estrangeiras e informações providas pelo programador, como pontos para estabelecer conexões e os métodos transacionais.

A importação do modelo consiste em extrair do banco os metadados que compõem o modelo. Para que essa importação seja realizada, algumas informações para estabelecer a conexão com o banco de dados devem ser fornecidas. Essas informações são armazenadas no arquivo `database.xml`. A Figura 6.2 apresenta um exemplo do arquivo `database.xml`.

A configuração do modelo consiste em atribuir aos atributos vazios dos nodos do arquivo

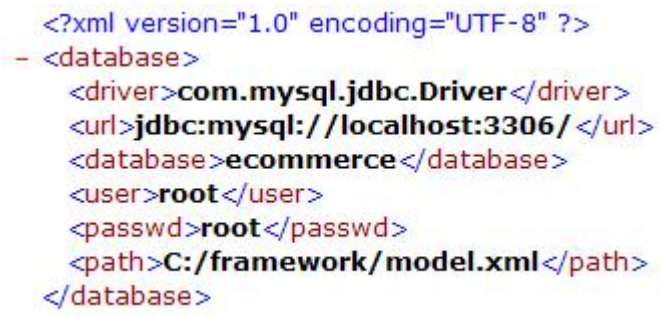

Figura 6.2: Arquivo database.xml

XML gerado na importação as informações necessárias para geração de código. Existem vários nodos do arquivo a ser configurado, dos quais se destacam:

1. **Modelo**: corresponde ao modelo do sistema em questão.
2. **Classe**: corresponde às classes do modelo que associam-se as tabelas do banco de dados.
3. **Id**: corresponde a um membro identificador de uma classe, o qual representa o campo chave primária de uma tabela no banco.
4. **Membro**: corresponde a um membro simples de uma classe.
5. **Campo**: corresponde a um campo de uma tabela no banco de dados.

Estes nodos se relacionam de acordo com o diagrama de classes da Figura 6.3. O modelo possui uma lista de classes; cada classe, por sua vez, possui um identificador e uma lista de membros. Um identificador possui um ou mais campos e um membro possui um único campo.

Para exemplificar, suponha um modelo de entidade-relacionamento que possua a entidade `Cliente`. O arquivo de configuração de modelo para essa entidade é apresentado na Figura 6.4.

O arquivo XML para controle de conexão e transação possui nodos que encapsulam o nome das classes e dos métodos para estabelecer conexões e nome dos métodos transacionais. A Figura 6.5 mostra um exemplo desse arquivo. Essa configuração se faz necessária, pois a ferramenta de geração criará o aspecto que define os pontos do programa que necessitam de controle de conexão e transação. Uma vez definidos quais são os métodos para esse controle, a ferramenta gera o aspecto correspondente. O aspecto gerado estende o aspecto `TransactionControlAspect` do GAP, que cuida do controle de conexão e transação.

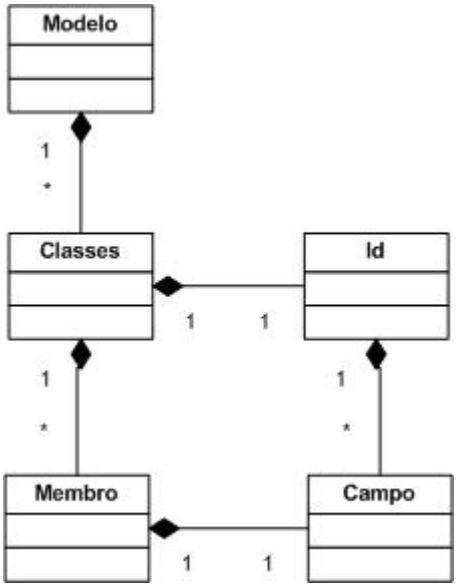

Figura 6.3: Diagrama de Classes do Modelo

```
<?xml version="1.0" encoding="UTF-8" ?>
- <Model name="Ecommerce">
  - <ListClass>
    - <Class aspectPackage="" entityPackage="" name="" table="cliente">
      - <Id>
          <Field attribute="ID_CLIENTE" name="" type="String" />
        </Id>
      - <ListMember>
          <Field attribute="CP_CLIENTE" name="" type="String" />
          <Field attribute="NM_CLIENTE" name="" type="String" />
          <Field attribute="EM_CLIENTE" name="" type="String" />
          <Field attribute="SE_CLIENTE" name="" type="String" />
          <Field attribute="DS_END_CLIENTE" name="" type="String" />
        </ListMember>
      </Class>
      ...
```

Figura 6.4: Arquivo model.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
- <Control name="Ecommerce">
  - <ListConnection>
    - <Connection>
        <Class>MyConnectionFactory</Class>
        <Method>getConnection</Method>
      </Connection>
    </ListConnection>
  - <ListTransaction>
    - <Transaction>
        <Class>ecommerce.controlador.ControladorCadastro</Class>
        <Method>cadastrarCliente</Method>
      </Transaction>
    - <Transaction>
        <Class>ecommerce.controlador.ControladorCadastro</Class>
        <Method>atualizarCliente</Method>
      </Transaction>
    - <Transaction>
        <Class>ecommerce.controlador.ControladorCadastro</Class>
        <Method>removerCliente</Method>
      </Transaction>
    </ListTransaction>
    <AspectPackage>ecommerce.aspectos</AspectPackage>
  </Control>
```

Figura 6.5: Arquivo connectiontransactioncontrol.xml

6.3 Geração de Código

O gerador é responsável por criar três aspectos: o aspecto responsável pelo controle de conexão e transação, chamado `NOMEMODELOTransactionControlAspect.aj`, o aspecto para sincronização de objetos, que é único para cada entidade do modelo, chamado de `NOMEENTIDADESynchronizationControlAspect.aj` e, por último, um aspecto responsável pela recuperação dos dados, chamado de `NOMEMODELORetrievalAspect.aj`. Além disso, o gerador é responsável por criar as classes persistentes que representam as entidades do banco de dados, denominadas `NOMEENTIDADE.java`, e as classes para acesso ao banco de dados, denominadas `NOMEENTIDADEDaoImpl.java`, as quais implementam as interfaces `INOMEENTIDADEDao.java` também geradas. O prefixo `NOMEMODELO` é substituído pelo nome do modelo definido no arquivo de configuração de modelo, e `NOMEENTIDADE` se refere a cada entidade do sistema, as quais também são definidas no arquivo de configuração.

`NOMEMODELOTransactionControlAspect.aj` estende o aspecto abstrato `TransactionControlAspect` do GAP e, como já foi dito anteriormente, tem a função de definir os pontos (métodos) que necessitam do controle de conexão e transação. `TransactionControlAspect` possui dois conjuntos de junção abstratos (`transactedOperation()` e `obtainConnection()`), que são especializados pelo aspecto gerado. Na Listagem 6.1 é

apresentado um exemplo do aspecto concreto para controle de transação.

```
1  ...
2  public aspect NOMEMODELOTransactionAspect
3                      extends TransactionControlAspect{
4      public pointcut transactedOperation() :
5                  execution(METODO1(..))
6                        || execution(METODO2(..))
7                        || execution(METODO3);
8                                      ...
9      public pointcut obtainConnection() :
10                                     call (METODO1);
11                                     ...
12 }
```

Listagem 6.1: NOMEMODELOTransactionAspect.aj

`NOMEENTIDADESynchronizationControlAspect.aj` estende o aspecto abstrato `SynchronizationControlAspect` do GAP e especializa o conjunto de junção que tem a função de sincronizar os objetos e sua representação no banco de dados. Na Listagem 6.2, apresenta-se esse aspecto concreto.

`NOMEMODELORetrieveAspect.aj` é um aspecto concreto que insere nas classes do sistema que representam entidades, construtoras que recebem como parâmetro um `ResultSet`. Estas construtoras são invocadas por métodos da classe `PersistentRetrieval` usando reflexão computacional. Na Listagem 6.3, apresenta-se um exemplo desse aspecto.

`NOMEENTIDADE.java` são as classes persistentes do sistema que representam as tabelas do banco de dados. Essas classes contêm membros que representam os atributos de cada tabela. Esses membros são inicializados e acessados através de métodos (`set`) e (`get`) que também são gerados pela ferramenta. Na Listagem 6.4, apresenta-se um exemplo de uma classe persistente.

`NOMEENTIDADEDaoImpl.java` são as classes do sistema que implementam o acesso ao banco para fazer atualizações dos registros. São nessas classes que se encontram métodos para inserção, atualização e remoção de registros. Na Listagem 6.5, apresenta-se um exemplo de classe de acesso ao banco de dados para entidades. Essa classe implementa a interface `INOMEENTIDADEImpl` que também é gerada pela ferramenta.

```
1  public aspect NOMEENTIDADESynchronizationControlAspect
2                    extends SynchronizationControlAspect {
3            declare parents: NOMEENTIDADE implements Persistent;
4      public pointcut create(Persistent obj):
5          execution(public NOMEENTIDADE.new(..)) && target(obj)
6          && !cflow(execution (public static * NOMEENTIDADE.find*(..)));
7      public pointcut update(Persistent obj):
8         (execution(public void NOMEENTIDADE.set*(..)) && target(obj)
9         && !cflow(execution (public static * NOMEENTIDADE.find*(..))));
10     public pointcut executeUpdate():
11                NOMEMODELOTransactionAspect.transactedOperation();
12     public void NOMEENTIDADE.create() throws Exception {
13         INOMEENTIDADEDAO dao = (..) DAOFactory.getDAO(..);
14         try {  dao.create(this); } catch (Exception e) {..}
15     }
16     public void NOMEENTIDADE.update() throws Exception {
17         INOMEENTIDADEDAO dao = (..) DAOFactory.getDAO(..);
18         try {dao.update(this); } catch (Exception e) {..}
19     }
20     public void NOMEENTIDADE.delete() throws Exception {
21          INOMEENTIDADEDAO dao = (..) DAOFactory.getDAO(..);
22          try {dao.delete(this);} catch (Exception e) {..}
23    }
24 }
```

Listagem 6.2: NOMEENTIDADESynchronizationControlAspect.aj

6.4 Comentários Finais

A ferramenta de geração auxilia o engenheiro de software na ligação do seu sistema com o GAP. A ferramenta proposta gera automaticamente os aspectos para controle de conexão e transação e sincronização de objetos. Gera ainda os métodos para inserção, atualização, deleção e recuperação de registros no banco de dados. O seu uso provê um ganho de produtividade, uma vez que a especialização dos aspectos é de sua responsabilidade e não do desenvolvedor. Além disso, a ferramenta, facilita a manutenção do modelo entidade-relacionamento e reduz pontencialmente o numero de erros de programação por parte do desenvolvedor.

Para gerar tais aspectos exige-se apenas que o usuário do GAP informe parâmetros como os seguintes: pontos do sistema onde deve-se estabelecer a conexão, métodos que atualizam o estado de objetos persistentes e o mapeamento de tais objetos para tabelas de

```
1 public aspect NOMEMODELORetrievalAspect {
2     public NOMEENTIDADE.new(ResultSet rs) throws Exception
3     {
4         if(rs != null){
5             this.setMEMBRO1(rs.getTIPOCOLUNA1("COLUNA1"));
6             this.setMEMBRO2(rs.getTIPOCOLUNA2("COLUNA2"));
7         }
8     }...
9 }
```

Listagem 6.3: NOMEMODELORetrievalAspect.aj

```
1  public class NOMEENTIDADE {
2      private TIPO1 membro1;
3      private TIPO2 membro2;
4
5      public NOMENTIDADE(){}
6      public NOMENTIDADE(TIPO1 membro1, TIPO2 membro2){
7          this.membro1 = membro1;
8          this.membro2 = membro2;
9      }
10     public void setMEMBRO1(TIPO1 membro1){this.membro1 = membro1;}
11     public TIPOMEMBRO1 getMEMBRO1(){ return this.membro1; }
12     public void setMEMBRO2(TIPO2 membro2){this.membro2 = membro2;}
13     public TIPOMEMBRO2 getMEMBRO2(){ return this.membro2; }
14 }
```

Listagem 6.4: NOMEENTIDADE.java

um banco de dados relacional. Esses parâmetros são todos informados por meio de uma interface gráfica, dispensando assim o usuário de dominar uma linguagem orientada por aspectos. Na versão atual do sistema, assume-se que o meio de armazenamento persistente seja um banco de dados relacional. No entanto, nada impede que o GAP seja utilizado com outros meios de armazenamento. Por fim, o sistema não requer que a aplicação alvo obedeça a uma arquitetura particular (por exemplo, que seja um sistema *Web*, um sistema três camadas, etc).

A ferramenta consiste de uma pequena aplicação *Web* construída em Java que contém cerca de 550 linhas de código. As principais funções da ferramenta são importar o modelo de dados a partir do banco, extrair as informações dos arquivos XML configurado pelo usuário e gerar código. O processo de importação do modelo de dados foi feito usando a

interface `DatabaseMetaData` da API de Java. Essa classe disponibiliza métodos necessários para obter metadados como tabelas, atributos e tipo de atributos. Já o processo de leitura das informações contidas no XML foi feito usando classes do pacote `org.w3c.dom` também da API de Java. Essas classes disponibilizam métodos para obter as informações contidas em elementos, nodos e atributos do XML.

O código gerado pela ferramenta é de qualidade pelo fato de ser simples, legível e modularizado. O principal ponto negativo do código gerado é que o mesmo introduz no sistema que está sendo construído, a dificuldade com o raciocínio modular. Raciocínio modular significa que podemos pensar a respeito de um módulo, independentemente do contexto em que esse módulo está inserido e que podemos deduzir o seu comportamento a partir de seus componentes. Os sistemas que utilizam o GAP juntamente com a ferramenta violam estes princípios pelo fato de possuir estruturas de aspectos em sua implementação.

```
1  public class NOMEENTIDADEDAOImpl
2                              implements INOMEENTIDADEDAO {
3       public void create(NOMEENTIDADE pNOMEENTIDADE)
4                              throws SQLException {
5           Connection connection =  MyConnectionFactory.getConnection();
6           PreparedStatement stmt = null;
7           String sql = "INSERT INTO NOMETABELA (COLUNA1, ...";
8           stmt = connection.prepareStatement(sql);
9           stmt.executeUpdate();
10          stmt.close();
11      }
12      public void update(NOMEENTIDADE pNOMEENTIDADE)
13                             throws SQLException {
14          Connection connection = MyConnectionFactory.getConnection();
15          PreparedStatement stmt = null;
16          String sql = "UPDATE NOMETABELA SET ...";
17          stmt = connection.prepareStatement(sql);
18          stmt.executeUpdate();
19          stmt.close();
20      }
21      public void delete(NOMEENTIDADE pNOMEENTIDADE)
22                              throws Exception {
23          Connection connection = MyConnectionFactory.getConnection();
24          PreparedStatement stmt = null;
25          String sql = "DELETE FROM NOMETABELA ...";
26          stmt = connection.prepareStatement(sql);
27          stmt.executeUpdate();
28          stmt.close();
29      }
30      ...
31 }
```

Listagem 6.5: NOMEENTIDADEDAOImpl.java

Capítulo 7

Estudo de Caso: Sistema de Comércio Eletrônico

Este capítulo descreve um estudo de caso baseado em um sistema de comércio eletrônico que emprega o arcabouço e a ferramenta de geração de código.

7.1 Descrição do Sistema de Comércio Eletrônico

O sistema de comércio eletrônico contém funções como cadastramento e controle de produtos, cadastramento de clientes e geração de pedidos. Os requisitos funcionais identificados foram implementados usando orientação por objetos (Java) e os transversais, como controle de conexão, controle de transação e sincronização dos objetos foram implementados usando o GAP. Analisando esse tipo de sistema, verifica-se que as entidades básicas são Cliente, Produto, Pedido e Item de Pedido. A Figura 7.1 apresenta o modelo de dados para tais entidades.

O modelo de dados apresenta as entidades do sistema e o relacionamento entre essas entidades. A entidade Cliente possui CPF, nome, email, senha, endereço e uma lista de pedidos. Cada Pedido possui o valor e uma lista de itens de pedido. Por sua vez, cada Item de Pedido possui um produto e a quantidade desse produto. Por último, Produto possui código, nome, descrição e preço.

Uma vez definidas as entidades do sistema e os relacionamentos, é possível automatizar o processo de implementação de persistência.

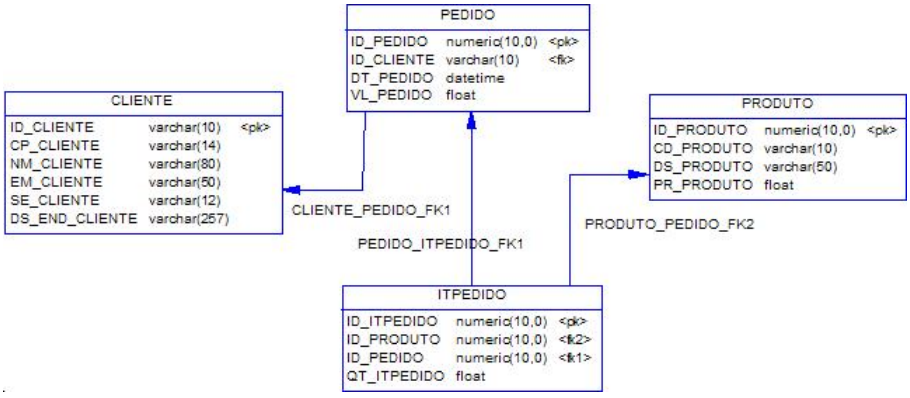

Figura 7.1: Modelo de Dados do Sistema de Comércio Eletrônico

7.2 Automação do Processo de Implementação de Persistência

A partir do modelo de dados apresentado na seção 7.1 foi criado um banco de dados no SGBD MySQL 4.1 [39]. Para utilização da ferramenta de geração, o engenheiro de software passa pela etapa de importação automática do modelo e armazenamento em um arquivo XML, configuração desse modelo importado, configuração dos pontos para estabelecer a conexão com o banco, definição dos métodos transacionais do sistema e, por último, a geração de código.

A primeira etapa a ser feita consiste em informar dados como URL do banco, local de importação, nome do banco, usuário, senha e *driver* de conexão, para que a importação do modelo do banco possa ser feita e armazenada em um arquivo XML. A Figura 7.2 mostra a interface gráfica para importação do modelo de dados.

Concluída a entrada de dados, o engenheiro solicita a importação do modelo. A ferramenta gera o modelo e o armazena em arquivo denominado `model.xml`. Um exemplo deste arquivo para o Sistema de Comércio Eletrônico é mostrado na Figura 7.3.

A segunda etapa consiste em configurar o modelo e definir os pontos do sistema que precisam de sincronização dos objetos. Para esse sistema, foi definido que as classes `Cliente`, `Produto`, `Pedido` e `ItemPedido` são classes persistentes, ou seja, classes que implementam a interface `Persistent` do arcabouço. Portanto, é feito o mapeamento entre as classes persistentes e as entidades do banco de dados. Além disso, é feito também um mapeamento entre os atributos das classes e os campos das tabela no banco de dados. A Figura 7.4 ilustra esse processo.

Figura 7.2: Geração do Modelo

```
<?xml version="1.0" encoding="UTF-8" ?>
- <Model name="Ecommerce">
  - <ListClass>
    - <Class aspectPackage="" entityPackage="" name="" table="cliente">
      - <Id>
          <Field attribute="ID_CLIENTE" name="" type="String" />
        </Id>
      - <ListMember>
          <Field attribute="CP_CLIENTE" name="" type="String" />
          <Field attribute="NM_CLIENTE" name="" type="String" />
          <Field attribute="EM_CLIENTE" name="" type="String" />
          <Field attribute="SE_CLIENTE" name="" type="String" />
          <Field attribute="DS_END_CLIENTE" name="" type="String" />
        </ListMember>
      </Class>
      ...
```

Figura 7.3: Arquivo model.xml

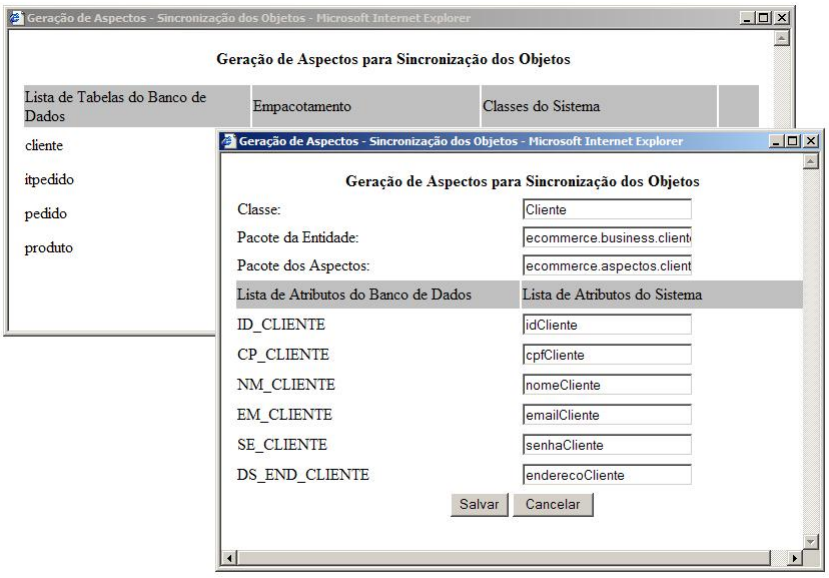

Figura 7.4: Sincronização de Objetos

A Figura 7.5 apresenta o arquivo `model.xml` após as configurações das entidades. Observe que os atributos **aspectPackage**, **entityPackage** e **name** do nodo **Class** foram preenchidos, juntamente com os atributos do nodo **Field** pertencente aos nodos pais **Id** e **ListMember**.

A terceira etapa consiste em definir o ponto do sistema que estabelece a conexão com o dispositivo de persistência e os métodos que precisam de controle de transação. Para o Sistema de Comércio Eletrônico estes pontos são definidos como mostra a Figura 7.6.

O método `getConnection()` da classe `MyConnectionFactory`, escrito pelo usuário, é usado para estabelecer a conexão com o banco e os métodos `cadastrarCliente`, `atualizarCliente` e `removerCliente` da classe `ControladorCadastro` são métodos transacionais.

Após todas essas configurações, a ferramenta está pronta para a geração de aspectos. Quando a geração for solicitada (pressionando o botão Gerar Código na Figura 7.7), seis aspectos, oitos classes e quatro interfaces são geradas. Sendo que quatro aspectos destinam-se a sincronização de objetos de cada entidade, um aspecto é responsável pelo controle de conexão e transação e o aspecto restante é responsável pela recuperação dos dados. As oito classes se dividem em quatro classes que representam as entidades do sistema e quatro classes responsáveis pelo acesso ao banco de dados. As quatro interfaces também são responsáveis pelo acesso ao banco. Para melhor entendimento, serão mostrados exemplos

```
<?xml version="1.0" encoding="UTF-8" ?>
- <Model name="Ecommerce">
  - <ListClass>
    - <Class aspectPackage="ecommerce.aspectos.cliente"
        entityPackage="ecommerce.business.cliente" name="Cliente" table="cliente">
      - <Id>
          <Field attribute="ID_CLIENTE" name="idCliente" type="String" />
        </Id>
      - <ListMember>
          <Field attribute="CP_CLIENTE" name="cpfCliente" type="String" />
          <Field attribute="NM_CLIENTE" name="nomeCliente" type="String" />
          <Field attribute="EM_CLIENTE" name="emailCliente" type="String" />
          <Field attribute="SE_CLIENTE" name="senhaCliente" type="String" />
          <Field attribute="DS_END_CLIENTE" name="enderecoCliente" type="String" />
        </ListMember>
      </Class>
    + <Class aspectPackage="ecommerce.aspectos.itpedido"
        entityPackage="ecommerce.business.itpedido" name="ItemPedido" table="itpedido">
    + <Class aspectPackage="ecommerce.aspectos.pedido"
        entityPackage="ecommerce.business.pedido" name="Pedido" table="pedido">
    + <Class aspectPackage="ecommerce.aspectos.produto"
        entityPackage="ecommerce.business.produto" name="Produto" table="produto">
    </ListClass>
  </Model>
```

Figura 7.5: Model XML Configurado

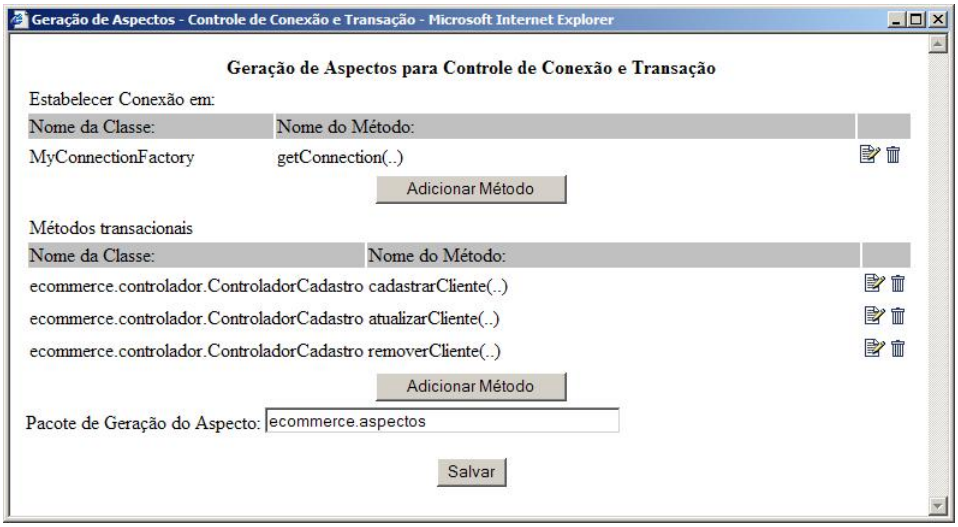

Figura 7.6: Controle de Conexão e Transação

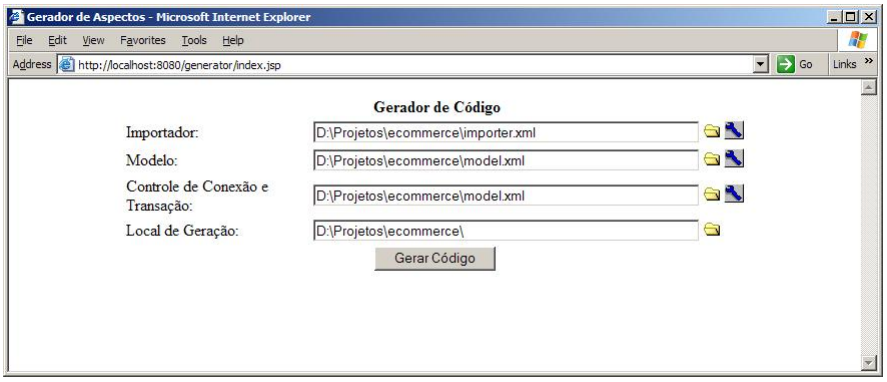

Figura 7.7: Geração de Código

dos aspectos gerados.

Para a entidade Cliente gerada pela ferramenta como mostra a Listagem 7.1, o aspecto abstrato `SynchronizationControlAspect` foi especializado pelo aspecto `ClienteSynchronizationControlAspect` definido na Listagem 7.2.

Como descrito na seção 5.3, toda classe persistente implementa a interface `Persistent`. Quando um objeto que implementa a interface `Persistent` for criado e essa criação ocorrer dentro de um método transacional, esse objeto é persistido no banco automaticamente após o término da execução desse método transacional. Essa sincronização é feita de forma automática porque o aspecto abstrato `SynchronizationControlAspect` pertencente ao arcabouço invoca em sua regra de junção `after` o método `create` que foi introduzido pelo aspecto concreto `ClienteSynchronizationControl` apresentado no código acima. Além disso, quando um objeto que implementa a interface `Persistent` for atualizado por meio da chamada a algum método `set`, esse objeto é adicionado na lista de objetos que ao término da execução do método transacional serão atualizados pela invocação do método `update` pelo aspecto abstrato `SynchronizationControlAspect`. Na Listagem 7.3 mostra-se a classe `ClienteDAOImpl` gerada pela ferramenta com a implementação dos métodos `create`, `update` e `delete`.

O aspecto responsável pelo controle de conexão e transação é apresentado na Listagem 7.4. Os conjuntos de junções `transactedOperation()` e `obtainConnection()` foram especializados pelos métodos definidos pelo engenheiro na etapa de configuração da ferramenta.

O aspecto responsável pela recuperação dos dados é mostrado na Listagem 7.5. Esse aspecto introduz nas classes persistentes as construtoras que tem o `ResultSet` como parâmetro. Essas construtoras são invocadas pelo método estático `getObjects` da classe `PersistentRetrieval` responsável pela recuperação dos dados.

```
1  public class Cliente {
2      private String login;
3      private String nome;
4      private String email;
5      private String senha;
6
7      public Cliente (){}
8      public Cliente(String login, String nome,
9                     String email, String senha){
10         this.login = login;
11         this.nome = nome;
12         this.email = email;
13         this.senha = senha;
14     }
15     public void setLogin(String login){ this.login = login; }
16     public String getLogin(){ return this.login; }
17     public void setNome(String nome){ this.nome = nome; }
18     public String getNome(){ return this.nome; }
19     public void setEmail(String email){ this.email = email; }
20     public String getEmail(){ return this.email; }
21     public void setSenha(String senha){ this.senha = senha; }
22     public String getSenha(){ return this.senha; }
23 }
```

Listagem 7.1: Cliente.java

Por último, o diagrama de classes do sistema de comércio eletrônico após a geração dos aspectos é apresentado na Figura 7.8.

Para a modelagem de sistemas, há estudos de adaptações da UML [44, 45] para o projeto de sistemas orientados por aspectos [8, 9, 50]. Destes, o estudo de padrões de composição [6, 7] pode ser destacado como uma metodologia de projeto para a separação de requisitos transversais. Um aspecto pode ser representado no diagrama de classes da UML de maneira semelhante a uma classe. Utiliza-se o designador <<**aspect**>> para denotar um aspecto. Além disso, no relacionamento de entidades, utiliza-se o designador <<**crosscutting**>> para indicar que um aspecto é transversal a uma classe, ou seja, esse aspecto entrecorta a classe, e o designador <<**introduction**>> para indicar que o aspecto introduz algo na classe.

O aspecto `EcommerceTransactionControlAspect` entrecorta métodos dos Controladores para inserir o controle de transação. O aspecto `SynchronizationControlAspect`

```
1  public aspect ClienteSynchronizationControlAspect
2          extends SynchronizationControlAspect {
3      declare parents: Cliente implements Persistent;
4      public pointcut create(Persistent obj):
5              execution(Cliente.new(..)) && target(obj)
6              && !cflow(execution (public static * find*(..)));
7      public pointcut update(Persistent obj):
8                     (execution(* Cliente.set*(..)) && target(obj)
9                     && !cflow(execution (public static * find*(..))));
10     public pointcut executeUpdate():
11             EcommerceTransactionAspect.transactedOperation();
12     public pointcut detectDeletedObject(Persistent p): this(p) &&
13             (execution(public * Persistent.get*(..)) ||
14                    execution(public * Persistent.set*(..)));
15     public void Cliente.create() throws Exception {
16         IClienteDAO dao = (IClienteDAO) DAOFactory.getDAO(..);
17         try {  dao.insert(this); } catch (Exception e) {..}
18     }
19     public void Cliente.update() throws Exception {
20         IClienteDAO dao = (IClienteDAO) DAOFactory.getDAO(..);
21         try {dao.update(this); } catch (Exception e) {..}
22     }
23     public void Cliente.delete() throws Exception {
24          IClienteDAO dao = (IClienteDAO) DAOFactory.getDAO(..);
25          try {dao.delete(this);} catch (Exception e) {..}
26    }
27 }
```

Listagem 7.2: ClienteSynchronizationControlAspect.aj

entrecorta os métodos `new` e `set` das classes persistentes para inserir o controle de transação. Além disso, esse aspecto é especializado pelos aspectos `ClienteSynchronization-ControlAspect`, `ProdutoSynchronizationControlAspect`, `PedidoSynchronization-ControlAspect` e `ItemPedidoSynchronizationControlAspect`, os quais têm a função de introduzir o código referente a criação, atualização e remoção nas classes persistentes.

```
1  public class ClienteDAOImpl implements IClienteDAO {
2      public void create(Cliente pCliente) throws SQLException {
3          Connection connection =  MyConnectionFactory.getConnection();
4          PreparedStatement stmt = null;
5          String sql = "INSERT INTO CLIENTE (ID_CLIENTE ...";
6          stmt = connection.prepareStatement(sql);
7          stmt.executeUpdate();
8          stmt.close();
9      }
10     public void update(Cliente pCliente) throws SQLException {
11         Connection connection = MyConnectionFactory.getConnection();
12         PreparedStatement stmt = null;
13         String sql = "UPDATE CLIENTE SET ...";
14         stmt = connection.prepareStatement(sql);
15         stmt.executeUpdate();
16         stmt.close();
17     }
18     public void delete(Cliente pCliente) throws Exception {
19         Connection connection = MyConnectionFactory.getConnection();
20         PreparedStatement stmt = null;
21         String sql = "DELETE FROM CLIENTE ...";
22         stmt = connection.prepareStatement(sql);
23         stmt.executeUpdate();
24         stmt.close();
25     }
26 }
```

Listagem 7.3: ClienteDAOImpl.java

7.3 Conclusões

Usando a ferramenta de geração de código para o Sistema de Comérico Eletrônico com quatro entidades (Cliente, Pedido, ItemPedido e Produto), foram gerados seis aspectos, quatro classes e quatro interfaces. Sendo que quatro aspectos destinam-se a sincronização de objetos de cada entidade, um aspecto é responsável pelo controle de conexão e transação e o aspecto restante responsável pela recuperação dos dados. As quatro classes e as quatro interfaces são responsáveis pelo acesso ao banco. Essa geração automática dos aspectos possibilitou a instanciação automática do GAP, ou seja, a ligação do Sistema Comércio Eletrônico com os *hot spots* disponibilizados pelo arcabouço.

Foi feita uma análise comparativa a respeito do número de classes que o desenvolvedor

```
1  public aspect EcommerceTransactionAspect
2                  extends TransactionControlAspect{
3      public pointcut transactedOperation() :
4          execution(public static void
5              ControladorCadastro.CadastrarCliente(..))
6          || execution(public static void
7                  ControladorCadastro.AtualizarCliente(..))
8          || execution(public static void
9                  ControladorCadastro.RemoverCliente(..));
10     public pointcut obtainConnection() :
11         call (Connection
12             MyConnectionFactory.getConnection(..));
13 }
```

Listagem 7.4: EcommerceTransactionAspect.aj

```
1  public aspect EcommerceRetrievalAspect {
2      public Cliente.new(ResultSet rs) throws Exception
3      {
4          if(rs != null){
5              this.setLogin(rs.getString("id_login"));
6              this.setNome(rs.getString("nom_usuar"));
7              this.setEmail(rs.getString("des_email"));
8              this.setSenha(rs.getString("id_senha"));
9          }
10     }...
```

Listagem 7.5: EcommerceRetrievalAspect.aj

deixou de criar e número de linhas de código transversal retirado das classes do sistema. Com o uso do GAP e da ferramenta de geração, o desenvolvedor deixou de criar quatro classes e quatro interfaces referentes a cada entidade persistente do sistema. Essas classes e interfaces têm a função de criar, atualizar e remover registros do banco. Foram retiradas em média sete linhas de código transversal de métodos transacionais. Sendo que dessas sete linhas, cinco eram destinadas ao requisito de controle de conexão e transação, ou seja, estabelecer conexão com o banco, configurar o *autocommit* do banco para falso, executar o *commit*, executar o *rollback* caso alguma exceção fosse gerada e liberar a conexão com o banco. As outras duas linhas destinavam-se a sincronização de objetos. Outro ponto importante a ser considerado foi a redução do número de linhas dos métodos responsáveis recuperação de dados. Em média dez linhas foram retiradas desses métodos, uma vez que

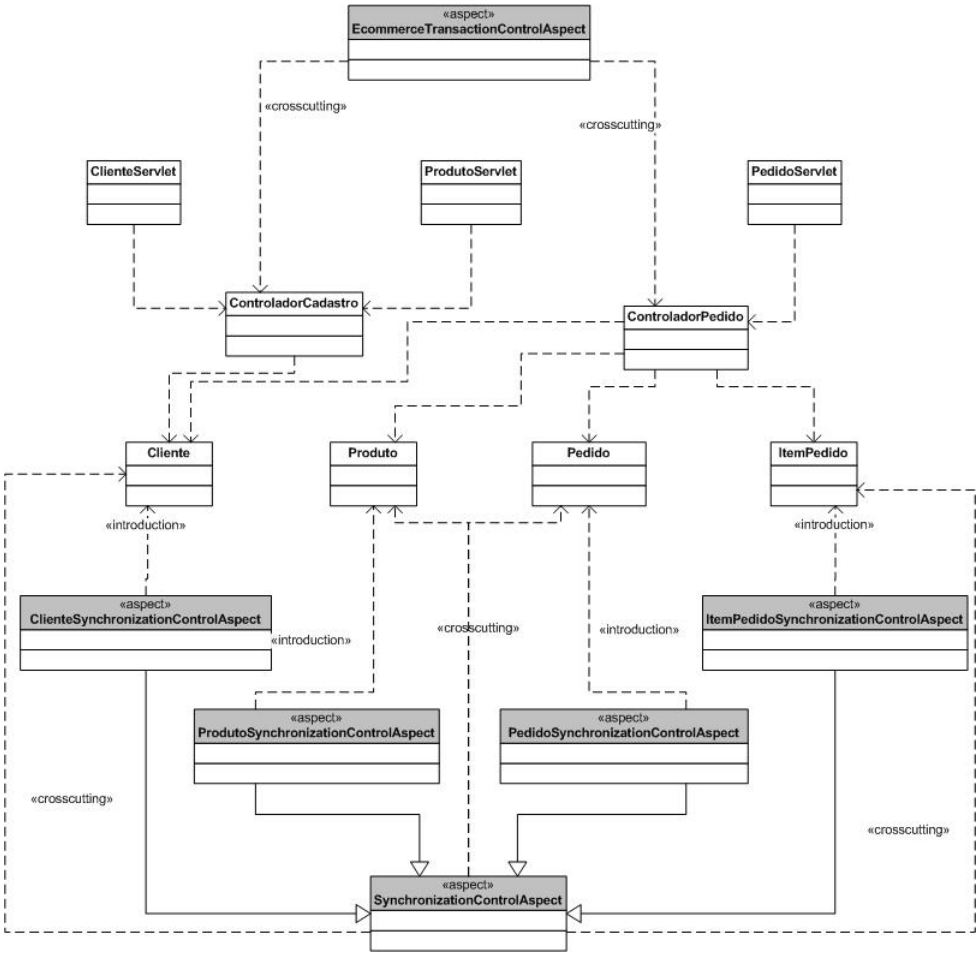

Figura 7.8: Diagrama de Classes

o desenvolvedor não precisa mais manipular acesso ao banco e nem manipular classes que cuidam da recuperação de dados como `ResultSet` em Java.

O estudo de caso realizado mostrou que o GAP possui algumas restrições de uso. As seguintes restrições foram encontradas:

1. a criação de um registro no banco está restrita ao preenchimento de todos os campos obrigatórios da entidade em questão, incluindo chaves estrangeiras. Ou seja, não existe uma forma de se criar registros no banco sem explicitamente definir o relacionamento entre esses registros. Suponha que um objeto `Pedido` possua uma lista de objetos `ItemPedido`. Neste caso, é necessário primeiro persistir o objeto `Pedido` no banco, obter a sua chave primária e logo em seguida persistir o objeto `ItemPedido`,

usando como chave estrangeira, a chave primária do objeto `Pedido`;

2. a ferramenta de geração não fornece mecanismos para manter possíveis alterações feitas pelo usuário nos módulos de implementação criados automaticamente. Por exemplo, se um usuário adicionar algum método em um arquivo gerado pela ferramenta, esse método será perdido quando uma nova geração for feita;

3. o processo de recuperação de dados usando o método `getObjects` da classe `PersistentRetrieval` é limitado no que diz respeito ao retorno de registros de uma tabela que estão relacionados com registros de outras tabelas. Por exemplo, não é possivel retornar um objeto da classe `Pedido` juntamente com a sua lista de objetos da classe `ItemPedido` em uma única consulta. É necessário primeiramente retornar o objeto da classe `Pedido`, fazer uma nova consulta usando sua chave primária, para então retornar a lista de objetos da classe `ItemPedido`;

4. a importação do modelo de dados na fase de configuração da ferramenta, se limita ao banco de dados MySQL;

5. o código de classes e métodos gerados pela ferramenta, também se limitam ao banco MySQL. A forma de se criar um registro no MySQL é diferente da forma de se criar um registro em outros bancos de dados como Oracle [40], por exemplo.

Existem soluções que poderiam ser usadas para minimizar, ou até mesmo, remover estas limitações. A seguir são apresentados algumas soluções:

- **restrição 1**: disponibilizar via aspectos, a construtora para uma classe persistente capaz de receber como parametro uma lista de objetos na qual a classe persistente se relaciona. Por exemplo, a construtora da classe `Pedido` recebe como parâmetro uma lista de objetos da classe `ItemPedido`. Dessa forma, o arcabouço persistiria automaticamente essa lista de objetos;

- **restrição 2**: definir pontos de alteração nos arquivos gerados. Esses pontos seriam delimitados por um comentário especial ou por uma anotação, permitindo que a ferramenta identifique o código a ser mantido.

Não existe uma solução simples para a restrição 3. Para a restrição 4 e 5, basta modificar o gerador de código para permitir que outros banco de dados sejam utilizados. As restrições encontradas são consideradas restrições de implementação, ou seja, a versão atual do arcabouço não contempla a solução para esses problemas. Tais restrições poderão ser minimizadas, ou até mesmo eliminadas, em trabalhos futuros.

No entanto, o estudo de caso realizado mostrou que o GAP é de fácil utilização, pois são necessárias somente algumas configurações iniciais. Além disso, o sistema atende a diversos sistemas, já que permite a configuração de qualquer classe e método que necessite usar o serviço de persistência. A ferramenta facilita ainda manutenção no modelo de entidade-relacionamento, uma vez que qualquer modificação no modelo requer somente uma nova importação e configuração, usando a ferramenta, do ponto específico alterado.

Capítulo 8

Avaliação do Arcabouço

Este capítulo avalia o arcabouço levando em consideração os fatores relativos à produtividade, modularização, automatização, facilidade de uso e extensibilidade.

8.1 Produtividade

O arcabouço GAP apresentado nesse trabalho melhora a produtividade do desenvolvimento de sistema que usa banco de dados relacional como dispositivo de persistência. As melhorias no desenvolvimento se devem ao fato de o arcabouço possuir uma ferramenta de geração de aspectos. A ferramenta aumenta a produtividade no desenvolvimento, automatizando as tarefas de criação de aspectos que são tediosas, repetitivas e sujeita a erros. Isto reduz o número de erros de programação e também ajuda a melhorar a qualidade do software desenvolvido. A ferramenta é responsável pela geração de todo o código referente ao requisito de persistência.

Para mostrar os benefícios do uso do GAP, observe o exemplo na Listagem 8.1 da classe `Cliente` sem o uso do arcabouço. A classe `Cliente` possui código referente a persistência nas linhas 18 a 21, o que prejudica sua modularização. Além disso, essa classe possui métodos referentes a recuperação de dados nas linhas de 24 a 26. A implementação de tais métodos torna-se trabalhosa quando feita da forma convencional, ou seja, quando é necessário manipular diretamente classes e métodos responsáveis por essa recuperação, como por exemplo a classe `ResultSet`. Na Listagem 8.2, apresenta-se a classe `Cliente` usando recursos de persistência do GAP e o método **findAll** implementado usando recursos da classe **PersistentRetrieval** também pertencente ao GAP.

Os métodos **create**, **update** e **delete** foram removidos do código da classe e adici-

```
1  public class Cliente implements Persistent{
2
3      private String nome;
4      private String cpf;
5      private String email;
6      public Cliente(..){..}
7
8      /* contrutora responsavel pela recuperacao de dados */
9      public Cliente(ResultSet rs){..}
10
11     public String getNome(){..}
12     public void setNome(String nome){..}
13     public String getCpf(){..}
14     public void setCpf(String cpf){..}
15     public String getEmail(){..}
16     public void setEmail(String email){..}
17
18     /*metodos de criacao, atualizacao e remocao*/
19     public void create(..){..}
20     public void update(..){..}
21     public void delete(..){..}
22
23     /*metodos para recuperacao dos dados*/
24     public Cliente findByPrimaryKey(..){..}
25     public Cliente findByCpf(..){..}
26     public ArrayList findAll(..){..}
27 }
```

Listagem 8.1: Cliente.java - Sem o uso do GAP

onados nos aspectos gerados. Nesta versão, os métodos para recuperação dos dados são construídos usando o método `getObjects` da classes `PersistentRetrieval`, por exemplo, o método **findAll** nas linhas 17 a 23. Basta que o desenvolvedor defina a *query* SQL e a classe de retorno, para que o método faça automaticamente a recuperação dos dados.

Outro exemplo de dispersão e entrelaçamento de código encontra-se na classe `ControladorCliente`, Listagem 8.3, onde o método `cadastrarCliente` contém código referente ao controle de transação nas linhas 4 e 8. Com uso do GAP, as classes responsáveis pelo controle de conexão e transação também são beneficiadas. O código referente a tais requisitos é modularizado e encapsulado nos aspectos do GAP. Na Listagem 8.4 apresenta-se a classe `ControladorCliente` após o uso do GAP.

```
1  public class Cliente{
2
3      private String nome;
4      private String cpf;
5      private String email;
6      public Cliente(..){..}
7
8      public String getNome(){..}
9      public void setNome(String nome){..}
10     public String getCpf(){..}
11     public void setCpf(String cpf){..}
12     public String getEmail(){..}
13     public void setEmail(String email){..}
14
15     /*metodos para recuperacao dos dados*/
16      ...
17      public static ArrayList findAll() throws Exception{
18         ArrayList listaClientes =
19                       PersistentRetrieval.getObjects(
20                            "select * from cliente",
21                            "ecommerce.business.cliente.Cliente");
22         return listaClientes;
23     }
24 }
```

Listagem 8.2: Cliente.java - Com o uso do GAP

Sem o uso do GAP, o método `create` da classe `ClienteDAO` deveria ser criado pelo desenvolvedor. O desenvolvedor é responsável por escrever o código que controla a transação e por definir e executar o String SQL correspondente à inserção dos dados no banco. Com o uso do GAP, o método `create` é criado pela ferramenta de geração e a invocação desse método é feita pelo aspecto de sincronização de objetos.

A fim de se obter resultados mais objetivos no que diz respeito a produtividade, foi implementada uma versão orientada por objetos do Sistema de Comércio Eletrônico (descrito no Capítulo 7), além da versão original orientada por aspectos usando o GAP. Uma forma de se calcular o ganho de produtividade de diferentes versões de um sistema é avaliar o número de linhas de códigos utilizados para a construção dessas versões. A seguir, é apresentado a fórmula que calcula a redução do número de linhas de código nas classes de

```
1  public class ControladorCliente{
2      public static Cliente cadastrarCliente(..) throws Exception{
3          try{
4              TransactionControl.beginTransaction(..);
5              Cliente cliente = clienteDAO.create(..);
6              return cliente;
7          }finally{
8              TransactionControl.endTransaction();
9          }
10     }
11 }
12 public class ClienteDAO implements IClienteDAO{
13     public Cliente create(String nome, String login,
14                 String email, String senha) throws SQLException{
15         Connection con = TransactionControl.getConnection();
16         Statement stm;
17         try {
18             stm = getStatement(con);
19             stm.executeUpdate("INSERT INTO CLIENTE (ID_CLIENTE...");
20             con.commit();
21             Cliente cliente = new Cliente(nome, login,
22                                           email, senha)
23             return cliente;
24         } catch (SQLException e){ throw(e)}
25     }
26 }
```

Listagem 8.3: ControladorCliente.java - Sem o uso do GAP

um sistema:

$$\textbf{Redução} = \frac{\textbf{Número de linhas OO} - \textbf{(Número de linhas OA)}}{\textbf{Número de linhas de código OO}}$$

Para se calcular o número de linhas de código encontrados em ambas as versões do sistema, utilizou-se um *plug-in* de métricas para o Eclipse [1]. O cálculo não contempla linhas de código de arquivos JSP da camada de apresentação e nem linhas de código de aspectos, classes e interfaces geradas pela ferramenta. A versão orientada por objetos possui 938 linhas. A versão orientada por aspectos utilizando o GAP possui 398 linhas de código escritas manualmente. Portanto, obteve-se a seguinte redução no tamanho de

```
1 public class ControladorCliente{
2     ...
3     public static Cliente cadastrarCliente(..) throws Exception{
4         return new Cliente(..);
5     }
6     ...
7 }
```

Listagem 8.4: ControladorCliente.java - Com o uso do GAP

código escrito manualmente:

$$\textbf{Redução do Número de Linhas de Código} = \frac{\textbf{938 - 398}}{\textbf{938}} = \textbf{0.57}$$

Analisando o resultado obtido, percebe-se que utilizando o GAP e a ferramenta de geração houve uma redução de 57% no número de linhas de código escritas manualmente na versão que utiliza o GAP. Esse resultado foi obtido pelo fato de a ferramenta de geração criar automaticamente todos os aspectos, classes e interfaces que implementam o requisito de persistência.

8.2 Modularização e Automatização

Outra métrica importante a ser avaliada é o nível de modularização e automatização provido pelo GAP em comparação com os arcabouços orientados por aspectos apresentados no Capítulo 4. Além disso, é importante também avaliar o nível de modularização das classes na versão orientada por aspectos do Sistema de Comércio Eletrônico em comparação com a versão orientada por objetos. Quanto maior o nível de modularização, maior será a facilidade de manutenção e maior será a qualidade do software desenvolvido.

Soares, Borba e Laureano [49] apresentam um arcabouço em que todas as especializações de aspectos abstratos e a implementação dos métodos para sincronização dos objetos ficam a cargo do desenvolvedor. O arcabouço oferece somente a modularização do interesse de persistência, deixando de lado qualquer tipo de automatização.

O padrão de projeto camada de persistência orientado por aspectos proposto por Camargo, Ramos, Penteado e Masiero não apresenta automatização em nenhum dos subpadrões (Camada Persistente, CRUD, Gerenciador de Tabelas, Métodos de Mapeamento de Atributos e Gerenciador de Conexão) [14]. Todas as especializações de aspectos ficam a cargo do desenvolvedor. A única automatização encontrada foi na solução original do

padrão de projeto camada de persistência proposta por Yoder e outros [54]. Nessa solução, no entanto os métodos para sincronização de objetos e recuperação de dados foram implementados usando orientação por objetos.

Rashid e Chitchyan [43] apresentam um arcabouço em que a especialização dos aspectos também fica a cargo do desenvolvedor. No entanto, a recuperação dos dados foi automatizada usando classes e métodos, responsáveis por executar *queries* SQL especificadas pelo desenvolvedor e retornar objetos representando registros do banco.

De uma forma geral, todos os trabalhos mencionados conseguem modularizar o requisito de persistência em um sistema. O código dos exemplos de sistemas mostrado nos trabalhos, tornou-se mais claro e funcional após o emprego dos aspectos. No entanto, todos apresentam deficiências na automatização. Em nenhum dos trabalhos é proposto algo parecido com a ferramenta do GAP. Soares, Borba e Laureano [49] mencionam a necessidade de uma ferramenta para auxiliar o desenvolvedor na criação de aspectos e a sugerem como trabalho futuro. Assim, a principal vantagem do arcabouço descrito nessa dissertação é incluir uma ferramenta com suporte a geração automática de aspectos responsáveis por concretizar conjuntos de junção abstratos.

Para avaliar o nível de modularização das classes de ambas as versões do Sistema de Comércio Eletrônico foram escolhidas as principais classes que utilizam recursos de persistência. As classes escolhidas foram, `Cliente`, `ItemPedido`, `Pedido`, `Produto`, `ControladorCadastro` e `ControladorPedido`. A Tabela 8.2 apresenta os resultados obtidos. O símbolo LC representa o número de linhas de código nas classes.

Classe	**LC versão OO**	**LC versão OA**	**Ganho**
Cliente	83	50	33
ItemPedido	30	30	0
Pedido	45	39	6
Produto	46	34	12
ControladorCadastro	80	75	5
ControladorPedido	68	63	5

Tabela 8.1: Nível de Modularização das Classes do Sistema

Analisando os resultados obtidos, percebe-se que o requisito de persistência foi modularizado nos aspectos gerados pela ferramenta. Métodos das entidades persistentes **Cliente**, **Produto** e **Pedido** (como `create`, `update` e `delete`) foram retirados e modularizados nos aspectos. Comandos para controle de transação embutidos nos métodos das classes `ControladorCadastro` e `ControladorPedido` também foram removidos. Isso possibilitou uma maior modularização do requisito de persistência e uma redução no número de linhas

do sistema.

8.3 Facilidade de Uso e Extensibilidade

A ferramenta de suporte a geração de aspectos que compõe o GAP fornece um conjunto de páginas Web que permite facilmente ao desenvolvedor informar os parâmetros necessários para geração dos aspectos, como pontos do sistema onde deve-se estabelecer conexões, métodos transacionais, além de fazer o mapeamento de objetos persistentes para tabelas de um banco de dados relacional. Uma vez realizadas tais configurações, a ferramenta é capaz de gerar todo o código referente ao requisito de persistência, retirando do desenvolvedor a tarefa tediosa e repetitiva de criação dos mesmos.

O GAP fornece recursos para que sistemas possam ser modificados no que diz respeito ao requisito de persistência. Um sistema implementado usando o GAP permite facilidade na alteração do modelo de dados, uma vez que qualquer alteração feita nesse modelo, basta gerar novamente o código para refletir tal alteração nos módulos de implementação do sistema. Assim, é possível alterar tipos de atributos de tabelas do banco, adicionar ou remover atributos em uma tabela e adicionar ou remover tabelas.

No entanto, a ferramenta de geração não fornece mecanismos para manter possíveis alterações feitas pelo usuário nos módulos de implementação criados automaticamente. Por exemplo, se um usuário adicionar algum método em um arquivo gerado pela ferramenta, esse método será perdido quando uma nova geração for feita. A solução para esse problema é relativamente simples. A idéia é definir pontos de alteração nos arquivos gerados. Esses pontos seriam delimitados por um comentário especial ou por uma anotação, permitindo que a ferramenta identifique o código a ser mantido.

O projeto inicial da ferramenta de geração somente permite o uso do banco de dados relacional MySQL como dispositivo de persistência, ou seja, a persistência não é extensível a outros bancos, a arquivos TXT, XML, etc. Além disso, a ferramenta se restringe a uma única linguagem de programação. Basicamente, para que se possa utilizar outro dispositivo de persistência deve-se alterar internamente a ferramenta de geração.

A idéia futuramente é criar arquivos de modelo que permitam generalizar o processo de geração, fazendo com que modificações no código do gerador não sejam necessárias. Tais arquivos funcionariam como *templates* para o gerador. Caso exista a necessidade de se alterar o dispositivo de persistência, bastaria alterar o *template* inserindo o código correspondente ao novo dispositivo.

Uma solução semelhante às *templates* é encontrada em MetaJ [16]. MetaJ é um ambiente para meta-programação construído como uma extensão da linguagem Java que

define quatro abstrações que capturam recursos essenciais de uma ferramenta de meta-programação: referências-p, iteradores, *templates* e *plug-ins*. Com a utilização dessas abstrações, programas Java podem trabalhar facilmente com trechos de programas de qualquer linguagem. A idéia seria usar MetaJ para a geração do código responsável pela persistência a partir de meta-programas.

8.4 Comentários Finais

Um arcabouço orientado por aspectos para modularização de persistência certamente auxilia o engenheiro de software na construção de sistemas. O arcabouço avaliado neste capítulo fornece mecanismos para controlar conexões com o banco de dados, controlar transações, persistir dados voláteis e talvez a mais importante contribuição, inclui uma ferramenta de geração de aspectos, a qual é responsável por criar de forma automática todos os aspectos concretos que especializam os aspectos abstratos do GAP.

Com uso do GAP e da ferramenta de geração na construção de um sistema, o engenheiro de software não precisará se preocupar com a implementação do requisito de persistência. O GAP implementa tal requisito e o modulariza via aspectos. O uso da ferramenta de geração propicia ganho de produtividade, uma vez que a especialização dos aspectos fica a cargo da ferramenta e não do programador. A ferramenta fornece uma maior modularização das classes do sistema, uma vez que o código transversal de persistência é encapsulado nos aspectos. A ferramenta fornece ainda facilidade na alteração do modelos de dados. Qualquer alteração feita no modelo requer apenas que se gere código novamente.

As principais restrições do GAP e da ferramenta foram levantadas na Seção 7.3 do Capítulo 7. Dentre as restrições destacam-se problemas no relacionamento entre entidades, problemas na atualização de atributos de um objeto persistente na mesma transação que esse objeto foi criado, problemas em manter métodos criados pelo usuário dentro de módulos de implementação gerados pela ferramenta e problemas com a limitação de linguagem e dispositivo de persistência, que a princípio somente permite o banco de dados MySQL.

Capítulo 9

Conclusões e Trabalhos Futuros

A implementação de requisitos transversais em linguagens orientadas por objetos leva a duas condições adversas conhecidas como dispersão e entrelaçamento de código. Programação orientada por aspectos é um novo paradigma de programação que tem por objetivo modularizar requisitos de um sistema que por sua vez não podem ser modularizados por meio de programação orientada por objetos. A POA vem para separar os requisitos transversais do sistema dos requisitos não transversais, reduzindo, ou até mesmo eliminando, a dispersão e o entrelaçamento de código.

O requisito de persistência possui um comportamento transversal e pode ser dividido em sub-requisitos como, controle de conexões, controle de transações, controle de sincronização de objetos e recuperação de dados. Esses sub-requisitos quando encontrados em módulos de um sistema, apresentam problemas de dispersão e entrelaçamento de código. Assim, não surpreende que a implementação de persistência seja um exemplo clássico de emprego de orientação por aspectos. No entanto, os arcabouços já propostos para implementação de persistência por meio de aspectos demandam um esforço considerável de programação, exigindo a definição de diversos aspectos e interfaces. A implementação de tais aspectos é normalmente tediosa e, portanto, sujeita a erros. Na prática, tais problemas podem dificultar a adoção desses arcabouços por parte de desenvolvedores de sistemas. Daí a importância de se construir um arcabouço para implementação de persistência que inclua uma ferramenta de geração de código que auxilie usuários a criar os aspectos de persistência específicos de uma determinada aplicação.

O objetivo de se construir um arcabouço para implementação do interesse transversal de persistência, englobando controle de conexão, controle de transação e sincronização de objetos foi alcançado nesta dissertação. Além disso, foi comprovado com o arcabouço que técnicas de programação orientada por aspectos conseguem resolver os problemas de entrelaçamento e dispersão do código ocasionados pelo requisito de persistência.

O GAP para implementação de persistência possui as seguintes características:

- fornece mecanismos para controle da conexões com bancos de dados;
- fornece mecanismos para controle de transações, garantindo a integridade dos dados;
- fornece mecanismos para sincronização entre dados voláteis e dados persistentes;
- inclui um gerador de aspectos, o qual é responsável por criar de forma automática todos os aspectos responsáveis pelo controle de conexão e transação com o banco e por persistir objetos de uma aplicação alvo em um meio de armazenamento não-volátil;
- foi construído para uso em sistemas gerais desenvolvidos em Java;
- foi implementado em AspectJ.

O estudo de caso realizado mostrou que o GAP é de fácil utilização, pois são necessárias somente algumas configurações iniciais. A ferramenta de suporte a geração de aspectos que compõe o GAP fornece um conjunto de páginas Web que permite facilmente ao desenvolvedor informar os parâmetros necessários para geração dos aspectos, como pontos do sistema onde deve-se estabelecer conexões, métodos transacionais, além de fazer o mapeamento de objetos persistentes para tabelas de um banco de dados relacional.

O GAP fornece recursos para que sistemas possam ser modificados no que diz respeito ao requisito de persistência. Um sistema implementado usando o GAP permite facilidade na alteração do modelo de dados, uma vez que qualquer alteração feita nesse modelo, basta gerar novamente o código para refletir tal alteração nos módulos de implementação do sistema. Além disso, o GAP atende a diversos sistemas, já que permite a configuração de qualquer classe e método que fazem uso do serviço de persistência. O estudo de caso realizado mostrou também que o GAP tem algumas restrições de uso. As seguintes restrições foram encontradas:

- a criação de um registro no banco está restrita ao preenchimento de todos os campos obrigatórios da entidade em questão, incluindo chaves estrangeiras. Ou seja, não existe uma forma de se criar registros no banco sem explicitamente definir o relacionamento entre esses registros;
- a ferramenta de geração não fornece mecanismos para manter possíveis alterações feitas pelo usuário nos módulos de implementação criados automaticamente;
- o processo de recuperação de dados usando o método `getObjects` da classe `PersistentRetrieval` é limitado no que diz respeito ao retorno de registros de uma tabela que estão relacionados com registros de outras tabelas;

- a importação do modelo de dados na fase de configuração da ferramenta, se limita ao banco de dados MySQL;
- o código de classes e métodos gerados pela ferramenta, também se limitam ao banco MySQL.

9.1 Principais Contribuições

As principais contribuições alcançadas com esta dissertação foram as seguintes:

- apresentação e avaliação de um conjunto de arcabouços já propostos na literatura para a implementação de persistência;
- projeto e implementação do GAP, um arcabouço orientado por aspectos para modularização de persistência [15]. Esse produto embute uma ferramenta que permite:
  - geração automática dos aspectos para controle de conexão e transação e sincronização de objetos;
  - especialização automática dos aspectos abstratos que compõem o GAP;
  - geração automática dos métodos para inserção, atualização, deleção e recuperação de registros no banco de dados.
- demonstração da utilização do GAP e da ferramenta em um sistema real de comércio eletrônico;
- avaliação e comparação do sistema construído usando o GAP com uma versão do mesmo sistema orientado por objetos sem uso de qualquer arcabouço.

9.2 Trabalhos Futuros

A seguir são apresentados uma lista de possíveis trabalhos futuros:

- eliminar as restrições encontradas no estudo de caso do Capítulo 7;
- adicionar novos requisitos transversais ao GAP, como controle de autenticação, controle de autorização, registro de operações e segurança. A implementação de novos aspectos com intuito de modularizar tais requisitos, pode auxiliar o engenheiro de software na construção de sistemas;

- tornar a ferramenta de geração mais genérica, ou seja, uma ferramenta que permita gerar código para qualquer dispositivo de persistência. A idéia é criar arquivos de modelo que permitam generalizar o processo de geração, não sendo necessárias modificações no código do gerador. Tais arquivos funcionariam como *templates* de geração de código;

- criar versões desse arcabouço para outras linguagens de programação orientadas por aspectos. Linguagens de programação como, HyperJ e AspectC++ poderiam ser usadas com o intuito de avaliar o pontencial de modularização das mesmas.

Bibliografia

[1] Metrics 1.3.6. *http://metrics.sourceforge.net/*.

[2] Scott W. Amber. *Agile Database Techniques*. Wiley, 2003.

[3] AspectC++. *http://www.aspectc.org/*.

[4] Aspectj.org. *http://www.aspectj.org/*.

[5] Colin Atkinson and Thomas Kuhne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20(1):81–89, 2003.

[6] Johan Brichau, Maurice Glandrup, Siobhan Clarke, and Lodewijk Bergmans. Advanced separation of concerns. In *Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP) Workshop Reader*. Springer-Verlag, 2001.

[7] Siobhan Clarke and Robert J. Walker. Composition patterns: An approach to designing reusable aspects. In *International Conference on Software Engineering*, pages 5–14, 2001.

[8] Siobhán Clarke. Extending standard uml with model composition semantics. *Sci. Comput. Program.*, 44(1):71–100, 2002.

[9] Siobhán Clarke and Robert J. Walker. Towards a standard design language for aosd. In *AOSD '02: Proceedings of the 1st international conference on Aspect-Oriented Software Development*, pages 113–119, New York, NY, USA, 2002. ACM Press.

[10] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–98, New York, NY, USA, 2001. ACM Press.

[11] Tal Cohen and Joseph (Yossi) Gil. AspectJ2EE = AOP + J2EE. In *Proceedings of the 18th European Conference on Object-Oriented Programming*. Springer-Verlag, 2004.

[12] Constantinos A. Constantinides, Atef Bader, Tzilla H. Elrad, P. Netinant, and Mohamed E. Fayad. Designing an aspect-oriented framework in an object-oriented environment. *ACM Comput. Surv.*, 32(1es):41, 2000.

[13] Valter V. de Camargo and Paulo C. Masiero. Frameworks Orientados a Aspectos. In *XIX Simpósio Brasileiro de Engenharia de Software*, 2005.

[14] Valter V. de Camargo, Ricardo A. Ramos, Rosângela Penteado, and Paulo C. Masiero. Projeto Baseado em Aspectos do Padrão Camada de Persistência. In *XVII Simpósio Brasileiro de Engenharia de Software*, 2003.

[15] César Francisco de M. Couto, Marco Túlio O. Valente, and Roberto da S. Bigonha. Um Arcabouço Orientado por Aspectos para Implementação Automatizada de Persistência. In *II Workshop Brasileiro de Desenvolvimento de Software Orientado por Aspectos*, 2005.

[16] A. A. de Oliveira, T. H. Braga, M. Maia, and R. da Silva Bigonha. Metaj: An extensible environment for metaprogramming in java. 10(7):872–891, 2004.

[17] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[18] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns:Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[19] Eclipse.org. *http://www.eclipse.org*.

[20] Ramez Elmasri and Shamkant B. Navathe. *Sistemas de Banco de Dados - Fundamentos e Aplicações*. LTC, 2002.

[21] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[23] Joseph D. Gradecki and Nicholas Lesieck. *Mastering AspectJ: Aspect-Oriented Programming in Java*. Maning, 2003.

[24] Hibernate. *http://www.hibernate.com/*.

[25] Ralph E. Johnson. Components, frameworks, patterns. In *SSR '97: Proceedings of the 1997 Symposium on Software Reusability*, pages 10–17, New York, NY, USA, 1997. ACM Press.

[26] Ralph E. Jonhson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, 1997.

[27] Gregor Kiczales and Erik Hilsdale. Aspect-Oriented Programming. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, page 313. ACM Press, 2001.

[28] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, page 220ff. Springer-Verlag, 2001.

[29] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, page 220ff. Springer-Verlag, 1997.

[30] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Maning, 2003.

[31] Donal Lafferty and Vinny Cahill. Language-independent Aspect-Oriented Programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12. ACM Press, 2003.

[32] Log4J. *http://www.log4j.org*.

[33] Robert C. Martin. *Dependency Inversion Principle*. http://www.objectmentor.com/resources, 1996.

[34] Vlada Matena and Beth Stearms. *Applying Enterprise JavaBeans*. Addison Wesley, 2001.

[35] Michael Mattsson, Jan Bosch, and Mohamed E. Fayad. Framework integration problems, causes, solutions. *Commun. ACM*, 42(10):80–87, 1999.

[36] Sean McDirmid and Wilson C. Hsieh. Aspect-oriented programming with jiazzi. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 70–79, New York, NY, USA, 2003. ACM Press.

[37] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[38] Glenford J. Myers. *Reliable Software through Composite Design*. Petrocelli/Charter, 1975.

[39] MySQL. *http://www.mysql.com/*.

[40] Oracle. *http://www.oracle.com/database/*.

[41] Harold Ossher and Peri Tarr. Hiper/J: Multi-Dimensional Separation of Concerns for Java. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 734–737. ACM Press, 2000.

[42] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[43] Awais Rashid and Ruzanna Chitchyan. Persistence as an Aspect. In *Proceedings of the 2nd Aspect-Oriented Software Development*, pages 120–129. Springer-Verlag, 2003.

[44] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman, 1999.

[45] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language User Guide*. Addison-Wesley Longman, 1999.

[46] H. A. Schmid. Systematic framework design by generalization. *Communications of the ACM*, 40(10):48–51, 1997.

[47] Bill Shannon, Mark Hapner, Vlada Matena, James Davidson, and Larry Cable. *Java 2 Enterprise Edition: Platform and Component Specifications*. Addison Wesley, 2000.

[48] Sérgio Soares and Paulo Borba. Distribution and persistence as aspects. *Software: Practice and Experience*, 2006. A ser publicado.

[49] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 174–190. ACM Press, 2002.

[50] Dominik Stein, Stefan Hanenberg, and Rainer Unland. A uml-based aspect-oriented design notation for aspectj. In *AOSD '02: Proceedings of the 1st international conference on Aspect-Oriented Software Development*, pages 106–112, New York, NY, USA, 2002. ACM Press.

[51] Fabio Tirelo, Mariza A. da S. Bigonha Roberto da S. Bigonha, and Marco T. de O. Valente. Desenvolvimento de Software Orientado por Aspectos. In *XIII Jornada de Atualização em Informática*. Sociedade Brasileira de Computação, 2004.

[52] Bart Vanhaute, Bart De Win, and Bart De Decker. Building Frameworks in AspectJ. Position paper for the ECOOP2001 ASOC Workshop, June 2001. http://trese.cs.utwente.nl/Workshops/ecoop01asoc/newpage11.htm.

[53] Arndt von Staa. *Programação Modular*. Editora Campus, 2000.

[54] Yoder J. W., Johnson R. E., and Wilson Q. D. Connecting Business Objects to Relational Databases. In *Conference on the Pattern Languages of Programs*, 1998.

[55] Mitchell Wand. Understanding aspects: extended abstract. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 299–300, New York, NY, USA, 2003. ACM Press.