Projeto Baseado em Aspectos do Padrão Camada de Persistência

Valter Vieira de Camargo [1] - valter@icmc.usp.br / *Ricardo Argenton Ramos* [2,3] - rar@dc.ufscar.br /*Rosângela Penteado* [2] – rosangel@dc.ufscar.br / *Paulo Cesar Masiero* [1]– masiero@icmc.usp.br

[1] Universidade de São Paulo – USP
ICMC – Instituto de Ciências Matemáticas e de Computação
Cep 13.560-970 – São Carlos - SP

[2] Universidade Federal de São Carlos – UFSCar DC – Departamento de Computação
Cep 13.565-905 – São Carlos – SP

[3] Faculdade Gennari & Peartree - FGP
Rodovia Comandante João Ribeiro de Barros – Pederneiras - SP

Resumo

Padrões de projeto são criados e documentados para apoiar o reuso e a evolução de software, porém pesquisas mais recentes mostram que embora o projeto desses padrões seja reusável, o mesmo não ocorre com sua implementação. Alguns autores têm mostrado que as técnicas de implementação orientadas a objetos não são adequadas para a implementação de padrões de projeto por não possuírem mecanismos adequados para tratamento de interesses ortogonais ao interesses principais dos padrões. Porém, técnicas de programação orientadas a aspectos fornecem construções de linguagem adequadas ao encapsulamento desse tipo de comportamento, fazendo com que os benefícios de reusabilidade e evolução também ocorram com o código. Este artigo apresenta uma proposta de projeto baseado em aspectos para o padrão de projeto Camada de Persistência e também algumas diretrizes que auxiliam na obtenção de um projeto baseado em aspectos para um sistema orientado a objetos que tenha sido desenvolvido reusando um determinado padrão de projeto.

Palavras Chave: Padrões de Projeto, Aspectos, Camada de Persistência, Persistência, Modularidade.

Abstract

Design Patterns are considered well established mechanisms for software reuse and evolution. However, more recent researches show that although the design of these patterns is reusable, their implementation is not. Some authors have showed that object oriented implementation techniques are not suitable to implement design patterns because they do not have mechanisms to handle orthogonal concerns to the pattern´s main interest. Aspect oriented programming techniques provide language constructors more suitable to encapsulate this type of behavior, thus supporting reusability of the pattern's code. This paper presents a proposal of aspect-based design for the Persistence Layer design pattern and some guidelines that help creating an aspect-based design from an object-oriented system developed reusing a specific design pattern.

Key Words: Design Patterns, Aspects, Persistence Layer, Persistence, Modularity.

1. Introdução

Padrões de projeto auxiliam na solução de problemas recorrentes encontrados no desenvolvimento de software por meio da reutilização do projeto de soluções computacionais já testadas e aprovadas. Porém, enquanto o projeto dessas soluções pode ser reutilizado, o mesmo não ocorre com sua implementação [9] [11].

O padrão de projeto Camada de Persistência (*Persistence Layer*) diminui as dificuldades de implementação quando se utiliza uma linguagem de programação orientada a objetos e um banco de dados relacional. Trabalhos anteriores [3] mostram melhoras de manutenibilidade quando se utiliza esse padrão, porém a reusabilidade e a evolução da aplicação ficam comprometidas pela dependência existente entre a camada de aplicação e a de persistência. Essa dependência foi percebida por diversos autores em trabalhos anteriores que utilizam este padrão[1][2][3].

A utilização de padrões em ambientes de desenvolvimento reais necessita de técnicas de implementação apropriadas, caso contrário, a aplicação se torna dependente dos padrões, diminuindo as chances de reuso de código tanto da parte funcional da aplicação quanto da parte que implementa o padrão [11]. Essa dependência é causada pela utilização de técnicas de implementação orientadas a objetos como, por exemplo, relacionamentos de herança e associações. Essas técnicas causam o entrelaçamento do código funcional com o código de persistência e o espalhamento do código de persistência pelos módulos funcionais da aplicação.

O objetivo deste artigo é mostrar como o padrão Camada de Persistência pode ser implementado separando-se os aspectos funcionais dos aspectos relativos ao padrão, facilitando dessa forma o entendimento do padrão e melhorando sua reusabilidade. Para isso, compara-se uma implementação baseada em aspectos com uma orientada a objetos e apresentam-se diretrizes para esse tipo de implementação.

Na Seção 2 e 3 encontram-se alguns trabalhos relacionados e a descrição do padrão de projeto Camada de Persistência, respectivamente. A Seção 4 apresenta o projeto baseado em aspectos desse padrão e a Seção 5 as diretrizes que foram elaboradas a partir desse estudo. Na Seção 6, o processo para a obtenção do projeto baseado em aspectos do padrão Camada de Persistência é exemplificado e a Seção 7 discute sobre a composição da funcionalidade completa. As considerações finais são encontradas na Seção 8.

2. Padrões de Projeto e Separação de Interesses

Czarnecki e Eisenecker [4] comentam que linguagens de programação geralmente fornecem construtores para organizar um sistema em unidades modulares que representam os interesses funcionais da aplicação. Essas unidades são expressas como objetos, módulos e procedimentos. Mas também há interesses em um sistema que abrangem mais de um componente funcional, tais como: sincronização, interação de componentes, persistência e controle de segurança. Esses interesses são geralmente expressos por fragmentos de código espalhados pelos componentes funcionais. Eles são chamados de aspectos e alguns são dependentes de um domínio específico, enquanto outros são mais gerais [9].

Segundo Elrad e outros [5] [6] [7], a programação orientada a aspectos consiste na separação dos interesses de um sistema em unidades modulares e posterior composição (*weaving*) desses módulos em um sistema completo. Esses interesses podem variar de noções de alto nível, como segurança e qualidade de serviço, a noções de baixo nível, como sincronização e manipulação de *buffers* de memória. Eles podem ser tanto funcionais, como

características ou regras de negócio, quanto não-funcionais, como gerenciamento de transação e persistência.

Há duas abordagens lingüísticas mais difundidas para a implementação de um sistema orientado a aspectos: Hyper/J e Aspect/J. Segundo Elrad e outros [7], Hyper/J apóia a identificação e modularização de quaisquer tipos de interesses em Java, inclusive os de entrecorte (*crosscutting concerns*). Aspect/J é uma extensão orientada a aspectos de propósito geral da linguagem Java, em que a principal unidade modular é o *Aspect* [9]. Um *Aspect* pode possuir atributos e métodos e participar de uma hierarquia de aspectos por meio da definição de aspectos especializados. Aspect/J possibilita nomear um conjunto de pontos de junção e associar uma determinada implementação a eles, que pode ser executada antes, após ou apropriar-se do fluxo de execução dos eventos relacionados a esses pontos.

Segundo Noda e Kishi [11], as técnicas atuais de programação não são adequadas para a utilização dos padrões de projeto, pois tornam a aplicação dependente deles, o que diminui as chances de reuso da parte funcional da aplicação. Eles realizaram a implementação de duas versões de um mesmo sistema utilizando o conceito da separação avançada de interesses (*Advanced Separation of Concern - ASOC*) em Aspet/J e Hyper/J, utilizando o padrão de projeto *Observer* [8]. Nas duas versões do sistema os autores separam o código da aplicação do código do padrão, os quais foram tratados como interesses separados. O interesse que trata da aplicação contém apenas suas funções específicas, as quais são independentes do padrão, aumentando as chances de reuso desse interesse. O interesse que trata do padrão descreve a estrutura e o comportamento abstratos do padrão e é independente da aplicação.

Assim como Noda e Kishi[11], Hannemann e Kiczales [9] comentam que a técnica de implementação orientada a objetos não é adequada à implementação de padrões de projeto. Para testar essa hipótese, realizaram um experimento com programação orientada a aspectos no qual os vinte e três padrões de projeto propostos por Gama e outros [8] foram implementados. Os padrões foram implementados em Java e em Aspect/J e comparações mostraram que a implementação orientada a aspectos apresentou benefícios em dezessete dos vinte e três padrões. Os benefícios obtidos foram: melhor localidade de código, maior reusabilidade, facilidade de composição e de (desc)conectividade entre classes do padrão e da aplicação. Segundo os autores, alguns padrões possuem estrutura que entrecorta o relacionamento entre os papéis impostos pelo padrão e as classes de aplicação que assumem esses papéis. Os três tipos de estruturas identificadas foram: estrutura de papéis definidos, estrutura de papéis sobrepostos e estrutura mista. No primeiro tipo de estrutura o papel é bem definido, isto é, as classes participantes não têm funcionalidade além daquela fornecida pelo padrão. Nesses casos a implementação com Aspect/J não apresentou benefícios. No segundo tipo os papéis são sobrepostos, isto é, eles são atribuídos a classes que têm funcionalidade além daquela fornecida pelo padrão. Esse tipo de estrutura foi a que obteve maiores benefícios já que a programação orientada a aspectos possui construtores específicos para tratar relacionamentos de entrecorte. O terceiro tipo de estrutura possui tanto papéis definidos quanto sobrepostos e também obteve benefícios, porém menores do que o segundo tipo.

As melhorias obtidas com o uso do Aspect/J são inicialmente devidas à inversão das dependências. O código do padrão depende dos participantes e não ao contrário, como ocorre na implementação orientada a objetos. Isso possui um impacto direto na localidade de código, pois todas as dependências entre os padrões e a aplicação são localizadas no código do padrão.

Rashid e Chitchyan [15] propõem um projeto inicial de framework de persistência baseado em aspectos, com o objetivo de introduzir o interesse de persistência em uma aplicação nas fases finais do desenvolvimento. Os autores generalizaram as partes da persistência que são independentes da aplicação por meio da criação de pontos de junção

abstratos que devem ser concretizados por aspectos especializados, do mesmo modo que foi realizado neste trabalho. Os autores também dividiram o interesse de persistência em Conexão, Armazenamento e Atualização, Remoção, Transações e Acesso a meta-dados, mas não consideraram essas partes como sub-interesses de persistência nem basearam-se em algum padrão de projeto que trata de persistência. A conexão com o banco de dados foi implementada de forma bastante similar à proposta neste trabalho, porém, a abordagem dos autores difere em relação ao momento que os objetos são persistidos. Quando um objeto é instanciado, automaticamente ele é persistido no banco de dados e, quando é retirado da memória, ele também é removido do banco.

3. O Padrão de Projeto Camada de Persistência

Yoder e outros propuseram o padrão de projeto Camada de Persistência para minimizar a incompatibilidade existente entre o paradigma orientado a objetos e a persistência de dados em um banco relacional [14]. Esse padrão consiste em uma camada de persistência que cuida da interface entre objetos de aplicação e tabelas de banco de dados relacional. Os autores definem um conjunto de sub-padrões que podem ser utilizados para a implementação dessa camada de persistência:

- Camada Persistente (*Persistent Layer*): camada para salvamento e recuperação de objetos em um banco de dados relacional.
- *CRUD* (Criação (**C**reate), Leitura (**R**ead), Atualização (**U**pdate) e Remoção (**D**elete)): operações mínimas necessárias para a persistência de objetos, que são: criação, leitura, atualização e eliminação.
- Descrição de Código SQL (*SQL Code Description*): mecanismo de geração de cláusulas SQL para cada objeto persistente da aplicação. Essas cláusulas são utilizadas na implementação do padrão CRUD.
- Gerenciador de Tabelas (*Table Manager*): permite o mapeamento de um objeto para a sua respectiva tabela (ou tabelas) no banco de dados.
- Métodos de Mapeamento de Atributos (*Atributte Mapping Methods*): efetua o mapeamento entre os atributos de um objeto e as respectivas colunas da tabela (ou tabelas) do banco de dados.
- Conversão de Tipos (*Type Conversion*): trabalha em conjunto com o padrão Métodos de Mapeamento de Atributos convertendo os dados do banco de dados para o tipo apropriado de objetos e vice-versa.
- Gerenciador de Mudanças (*Change Manager*): mantém o controle de quais objetos sofreram modificação para que o sistema possa determinar quais objetos devem ser persistidos ou não. Auxilia o sistema a garantir a integridade dos dados.
- Gerenciador de Identificadores Únicos (*OID Manager*): gera automaticamente um identificador único para os objetos criados recentemente.
- Gerenciador de Conexão (*Connection Manager*): efetua a conexão do sistema com a base de dados e garante a sua manutenção.
- Gerenciador de Transação (*Transaction Manager*): mecanismo que permite o controle de transações com o banco de dados, com a implementação de mecanismos como *Commit* e *Rollback*, que melhora o controle e a segurança dos dados.

Três formas de implementação desse padrão são propostas por Yoder e outros. A terceira forma é a que foi utilizada neste trabalho por ser considerada a mais reusável e fácil de manter [3][1]. Ela utiliza uma classe de intermediação denominada `TableManager`, que é a implementação do padrão Gerenciador de Tabelas. Essa classe possui funcionalidade bem

definida, pois trata apenas de código de persistência, podendo ser facilmente reutilizada em outros contextos. Segundo Cagnin [3], essa forma de implementação foi a que mais facilmente permitiu reutilizar essa classe, pois não há preocupação com a persistência de qualquer tipo de objeto no banco de dados, necessitando apenas conhecer os parâmetros dos métodos das classes. Essa autora comparou o tempo necessário para a manutenção dos três modos de implementação desse padrão e os resultados mostraram que o sistema implementado utilizando a terceira forma é o mais manutenível. Embora a classe `TableManager` seja reusável, o mesmo não ocorre com as classes da aplicação, devido à dependência do padrão. Essa dependência ocorre porque essas classes invocam métodos existentes nas classes que implementam o padrão.

No contexto deste trabalho, o padrão Camada de Persistência representa o interesse de persistência de uma aplicação e os sub-padrões que o compõem, são seus sub-interesses.

3.1 Entrelaçamento e Espalhamento de Código

Como parte do objetivo proposto, foram desenvolvidas duas implementações de um sistema de oficina eletrônica, uma orientada a objetos usando puramente a linguagem Java, e outra usando a versão 1.04 do Aspect/J. Essa oficina efetua consertos em produtos eletrônicos como televisores, vídeo-cassetes e forno de microondas e seus técnicos recebem comissão referente aos consertos efetuados. Ambos os sistemas foram implementados para ambiente Web utilizando o banco de dados relacional *SyBase* e HTML para a implementação das interfaces. A comunicação entre o banco de dados e as interfaces foi implementada por meio de *servlets*, uma biblioteca da linguagem Java ideal para sistemas baseados na Web com arquitetura cliente-leve (*thin-client*).

A Figura 1 apresenta o modelo de classes para a implementação orientada a objetos do sistema de oficina eletrônica. As classes persistentes de aplicação devem implementar a interface `PersistentObject` e relacionar-se por associação com a classe `TableManager`. Embora não apresentadas no modelo, as classes dos *servlets* também relacionam-se com a classe `ConnectionManager`, que é responsável pela conexão com o banco de dados. A figura mostra que a utilização desse padrão divide o sistema em duas camadas lógicas, a camada de persistência, com três classes referentes ao padrão, e a camada de aplicação, referente às classes que implementam a funcionalidade do sistema.

A Figura 2 mostra um trecho de código da classe `Cliente` para a implementação orientada a objetos do sistema. A parte rotulada com a letra (a) mostra os atributos referentes ao padrão, denominados de atributos de persistência, que devem ser inseridos em todas as classes persistentes de aplicação. O quadro rotulado com a letra (b) mostra em negrito alguns métodos também referentes ao padrão, denominados de métodos de persistência, e que também devem existir em todas as classes de aplicação persistentes.

Tanto os atributos quanto os métodos de persistência encontram-se emaranhados com o código funcional e espalhados por todas as classes persistentes da aplicação. Além disso, dentro desses métodos encontram-se referências à classe `TableManager`, o que aumenta a dependência com o padrão. Tanto o entrelaçamento e o espalhamento do código relativo ao interesse de persistência quanto a dependência causada pelos mesmos, dificultam a evolução e a reutilização dessa classe em outros contextos [11]. Essas observações motivaram o estudo e a implementação do padrão Camada de Persistência utilizando o conceito de separação de interesses.

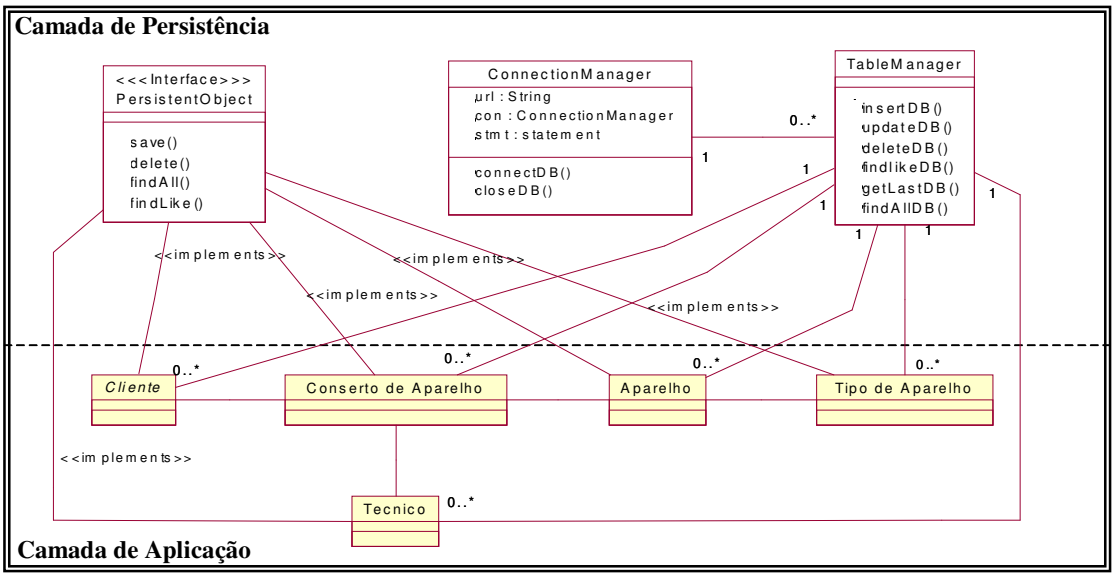

Figura 1 – O Padrão de Projeto Camada de Persistência

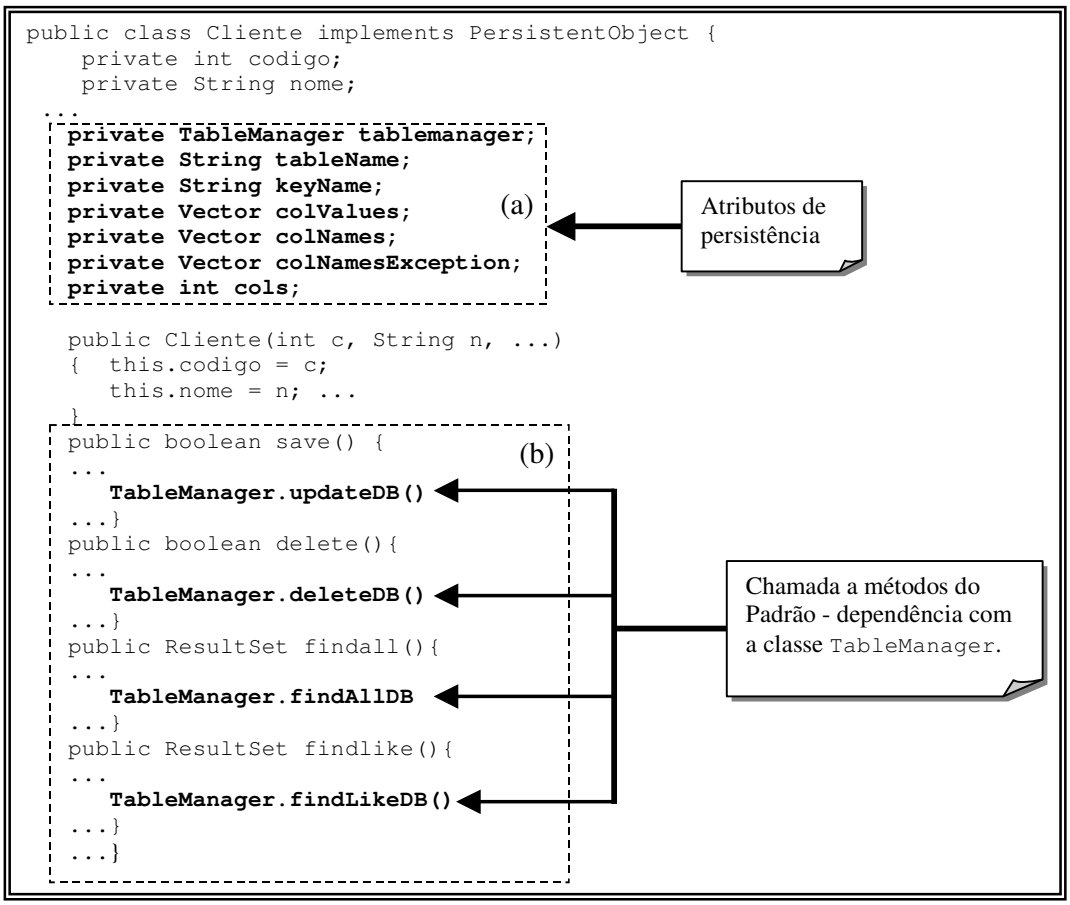

Figura 2 – Classe Cliente para Implementação Orientada a Objetos

Na implementação que utiliza aspectos, os atributos e métodos de persistência, bem como as dependências com o interesse de persistência mostradas na Figura 2 desaparecem completamente, permanecendo apenas os atributos e métodos referentes ao código funcional.

4. Projeto Baseado em Aspectos do Padrão Camada de Persistência

No estudo de caso realizado, identificou-se por meio do documento que descreve o padrão [14] que ele é composto por sub-padrões, considerados como sub-interesses dentro do interesse maior de persistência. Por isso, foi conveniente identificar quais atributos e métodos de persistência, espalhados pelo sistema, são provenientes de quais sub-padrões. Assim, realizou-se um mapeamento desses sub-interesses para determinados aspectos isto é, os aspectos foram criados utilizando fragmentos de vários sub-padrões ou um sub-padrão por completo, fazendo com que o mapeamento entre eles não seja sempre 1-1. Essa identificação de granulosidade de interesses, já mencionada por Soares [11], facilita a implementação modular desses sub-padrões em aspectos específicos de persistência.

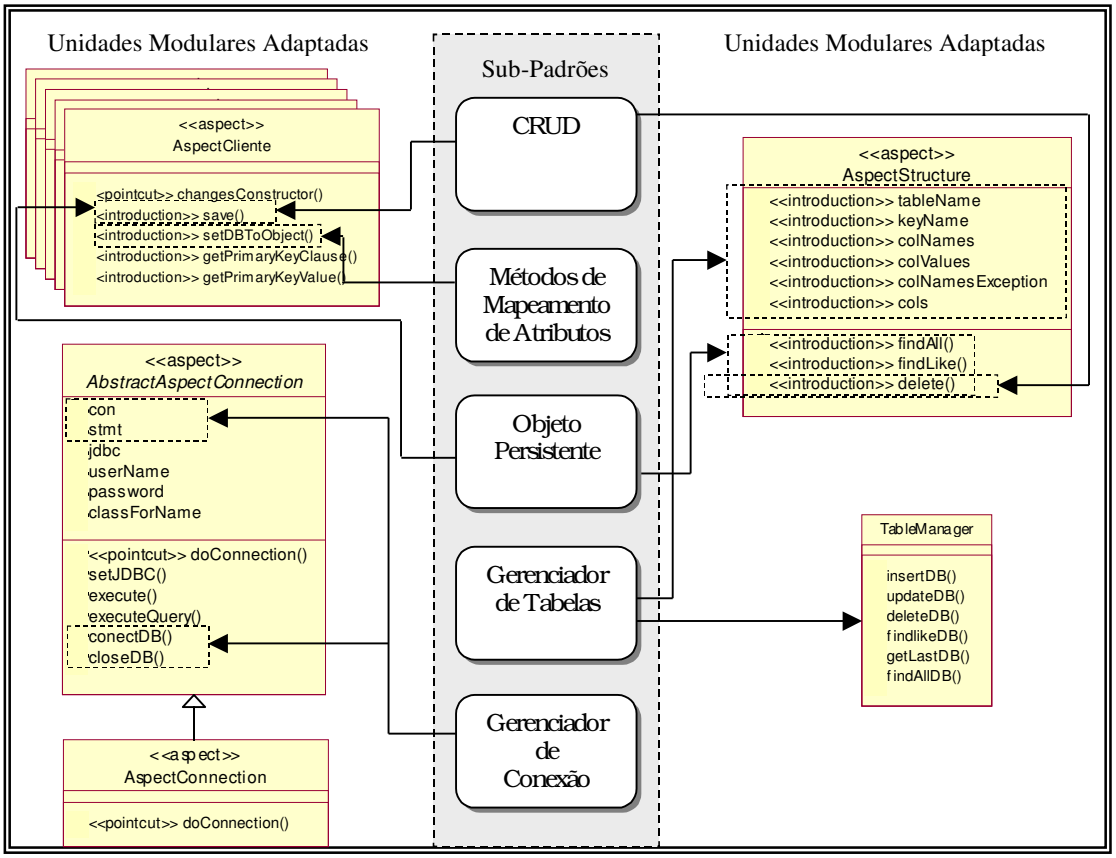

Figura 3 – Mapeamento Padrões x Aspectos

As duas implementações realizadas neste trabalho utilizaram os seguintes cinco sub-padrões (sub-interesses) dos dez que compõem o padrão Camada de Persistência: Objeto Persistente, CRUD, Gerenciador de Tabelas, Métodos de Mapeamento de Atributos e Gerenciador de Conexão.

Para melhor representar a semântica dos aspectos, os autores deste trabalho estenderam a UML por meio de alguns poucos estereótipos que podem se encontrados nas Figuras 3 e 4. Esses estereótipos são: `<<aspect>>`, `<<introduction>>`, `<<crosscutting>>` e `<<pointcut>>`.

A Figura 3 mostra que os cinco sub-padrões aplicados ao sistema deram origem a nove unidades modulares (classes ou aspectos) sendo: três aspectos genéricos: `AspectStructure`, `AbstractAspectConnection` e `AspectConnection`; a classe `TableManager`; e cinco aspectos específicos de cada classe de aplicação: `AspectCliente`(o único completamente visível na figura), `AspectAparelho`, `AspectConserto`, `AspectTecnico` e `AspectTipoAparelho`. As setas indicam o que foi fornecido pelo sub-padrão para a modularização em aspectos. Por exemplo, os sub-padrões CRUD, Métodos de Mapeamento de Atributos e Objeto Persistente colaboraram para o desenvolvimento do `AspectCliente`. Um aspecto como esse possui características específicas de cada classe de aplicação persistente e deve ser criado um aspecto como esse para cada classe persistente de aplicação.

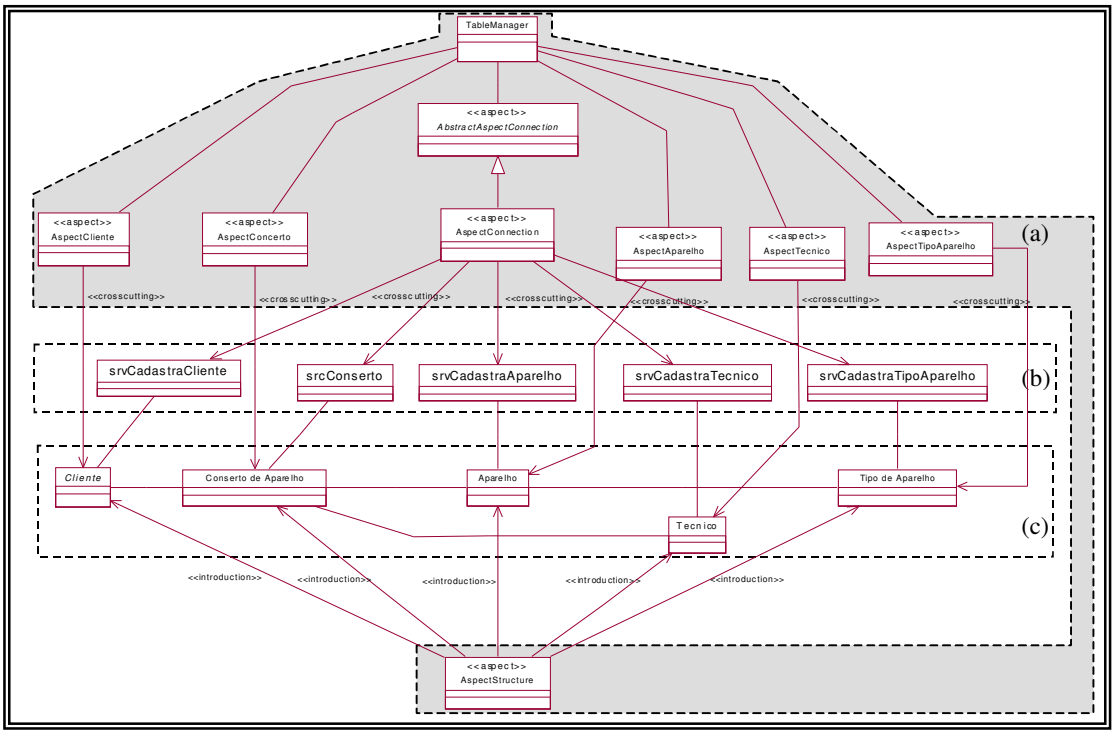

Figura 4 – Modelo de Projeto Baseado em Aspectos

O modelo de classe do sistema de oficina eletrônica utilizando aspectos é apresentado na Figura 4. A parte cinza destacada com a letra (a) é relativa ao padrão de projeto Camada de Persistência, modularizado com a utilização de oito aspectos e uma classe. Os aspectos específicos de cada classe entrecortam chamadas aos seus construtores por meio do ponto de corte `changesConstructor()`. O aspecto `AspectConnection`, que especializa o `AbstractAspectConnection`, entrecorta por meio de seu ponto de corte `doConnection()`, o método `init()` de todos os *servlets* para efetuar a conexão com o banco de dados. O aspecto `AspectStructure` introduz atributos e métodos genéricos em todas as classes de aplicação persistentes e a classe `TableManager` implementa os métodos de persistência que são utilizados por todos os aspectos específicos e também pelo aspecto da conexão.

A parte destacada com a letra (b) na Figura 4 mostra as classes referentes aos *servlets* utilizados como *middleware* entre as interfaces HTML e as classes de aplicação. Eles se relacionam com as classes de aplicação e com o aspecto `AspectConnection`. Vale ressaltar

que os métodos `init()` dos *servlets* existentes na implementação com aspectos não invocam o método `connectDB()` da classe `TableManager`, como acontece na implementação orientada a objetos. Na implementação com aspectos houve inversão de dependência, pois o `AspectConnection` é que entrecorta chamadas ao método `init()` e realiza a conexão com banco de dados.

Ainda na Figura 4, a parte destacada com a letra (c) apresenta as classes de aplicação persistentes, que na implementação com aspectos estão desprovidas de todos os atributos e métodos de persistência do padrão Camada de Persistência.

5. Diretrizes para Projeto Baseado em Aspectos

Esta seção apresenta diretrizes que podem auxiliar no processo de projeto baseado em aspectos, de um sistema que foi desenvolvido com o paradigma orientado a objetos e que utiliza um determinado padrão de projeto. Essas diretrizes foram criadas com base: na experiência obtida com a realização do estudo descrito neste artigo; nos trabalhos de Hannemann e Kiczales [9] e Noda e Kishi [11] e na experiência dos autores com a aplicação do padrão de projeto Camada de Persistência em trabalhos anteriores [1][2][3].

A utilização do padrão de projeto Camada de Persistência [14] com o paradigma orientado a objetos faz com que cada classe persistente de aplicação possua um mapeamento 1-1 com suas devidas tabelas do banco de dados. Assim, os atributos e métodos de persistência dessas classes referenciam dados de suas tabelas específicas, tornando-as dependentes desse mapeamento. Porém, também há métodos de persistência genéricos que não possuem essa dependência. Tendo em vista isso, as diretrizes, apresentadas a seguir, visam separar o comportamento genérico, o qual é agrupado em um único aspecto, do comportamento específico, o qual é agrupado em aspectos específicos de cada classe de aplicação. O processo consiste em identificar interesses de entrecorte, isto é, pontos onde o código do padrão está entrelaçado com o código que implementa a funcionalidade da classe. Nesses pontos, que podem ser atributos, métodos ou trechos de código dentro de métodos, deve haver uma análise a fim de identificar as dependências da aplicação com o padrão e vice-versa. Essas dependências irão guiar a criação de aspectos específicos e genéricos que deverão agrupar o comportamento de entrecorte do padrão, resultando em uma inversão de dependência, assim como descrito por Hannemann e Kiczales[9].

Como apresentado pelo Quadro 1, o passo número 1 das diretrizes consiste em obter o código fonte do sistema e o que mais estiver disponível sobre ele. Também é relevante obter algum documento que descreva detalhadamente o padrão. O passo 2 consiste em identificar os padrões que foram aplicados no sistema analisando apenas a documentação obtida no passo 1. O objetivo desse passo é que o responsável pelo processo fique ciente dos padrões que foram aplicados e de seus comportamentos de entrecorte. Esse tipo de comportamento geralmente pode ser constatado na seção Solução do documento que descreve o padrão, embora também haja documentos que possuem outras seções que mostram tal comportamento, como por exemplo a seção Exemplo de Implementação[14].

Quadro 1 - Diretrizes para o Projeto Baseado em Aspectos de Padrões de Projeto

1. Obter os artefatos disponíveis sobre o sistema e também sobre o padrão de projeto utilizado;
2. Tomar consciência dos padrões implementados e de sua granulosidade;
3. Para cada padrão;
   - 3.1. Para cada classe do sistema;
     - **3.1.1. Se essa classe assume inteiramente o papel definido pelo padrão;**
       - 3.1.1.1. Verificar a possibilidade de generalizá-la em um aspecto abstrato;
       - 3.1.1.2. Realizar a inversão de dependência;
     - **3.1.2. Para cada atributo específico do padrão;**
       - 3.1.2.1. Retirar esse atributo dessa classe e criar um aspecto (*) responsável por retornar esse atributo a essa classe de forma estática, e o valor desse atributo de forma dinâmica;
     - 3.1.3. Para cada método;
       - **3.1.3.1. Se é completamente específico do padrão;**
         - 3.1.3.1.1. Retirar esse método dessa classe e criar um aspecto genérico (*) responsável por retornar esse método a essa classe de forma estática;
       - **3.1.3.2. Se possui dependência do padrão e também da aplicação;**
         - 3.1.3.2.1. Avaliar a possibilidade de generalizar apenas a parte genérica em um aspecto;
         - 3.1.3.2.2. Retirar esse método dessa classe e criar um aspecto específico dessa classe (*) que retorne esse método a essa classe em tempo de composição;
       - **3.1.3.3. Se há trechos de código específicos do padrão;**
         - 3.1.3.3.1. Avaliar a possibilidade de generalizar a parte específica do padrão em um aspecto genérico;
4. Realizar a composição dos aspectos para obtenção do sistema completo.

Legenda

(*) = (ou usar um já existente)

O passo 3 é o processo de projeto propriamente dito, que é direcionado aos padrões identificados nos passos 1 e 2. Para cada padrão aplicado, buscam-se no código fonte, pontos de entrelaçamento e espalhamento com o código funcional da aplicação, isto é, procura-se identificar classes, atributos, métodos inteiros e/ou trechos de código dentro de métodos, que implementam funcionalidade relativa ao padrão. Quando esses pontos são identificados, procura-se modularizá-los em aspectos genéricos ou específicos, separando-os do código funcional da aplicação.

Os passos 3.1.1, 3.1.2, 3.1.3.1, 3.1.3.2 e 3.1.3.3 estão em negrito, pois representam algumas situações típicas quando se utiliza o padrão Camada de Persistência. Essas situações guiaram todo o processo de projeto e serão apresentadas em maiores detalhes na próxima seção.

6. Processo de Projeto Baseado em Aspectos

Como comentado anteriormente, as diretrizes apresentadas no Quadro 1 são baseadas em algumas situações típicas do padrão Camada de Persistência. Essas situações podem ser divididas em três grupos:

1. Classe com papel definido;
2. Atributos e Métodos específicos do Padrão;
3. Métodos específicos do padrão com dependência da aplicação;

Cada uma dessas situações é apresentada em maiores detalhes nas próximas sub-seções.

6.1 Classe com Papel Definido

Essa situação ocorre quando uma classe representa integralmente o papel definido pelo padrão. Na implementação orientada a objetos realizada neste trabalho, o sub-padrão Gerenciador de Conexão é implementado por uma classe que o representa completamente, isto é, ele não possui atributos e métodos espalhados pelo sistema. Sendo assim, o processo para essa situação consiste em criar um aspecto que representa o sub-padrão e localizar no código funcional o local em que esse padrão era utilizado, que é o ponto de junção entre o padrão e o código funcional. Com base nesse ponto de junção, cria-se um ponto de corte no aspecto que representa esse padrão e também uma Sugestão, que determina o momento correto do entrecorte, a fim de manter a funcionalidade desejada.

```
public abstract aspect AbstractAspectConnection
{
       public static Connection con;
       public static Statement stmt;
      ...
      abstract pointcut doConnection();

      after() : doConnection()
      {...
          ConnectDB();
       }
      public    void             setJDBC(...)    {...}
      protected static void      ConnectDB ()    {...}
      private   static void      CloseDB ()      {...}
      public    static void      execute()       {...}
      public    static ResultSet executeQuery() {...}
}
```

Figura 5 – Aspecto de Conexão Abstrato

```
public aspect AspectConnection extends AbstractAspectConnection
{
   pointcut doConnection(): call (void init(..)); (a)

   before() : doConnection()
      {
         super.setJDBC("jdbc:odbc:Oficina", "DBA", "sql",...); (b)
      }
}
```

Ponto de junção fornecido

Parâmetros de configuração

Figura 6 – Aspecto de Conexão Concreto

Conforme apresentado na Figura 3, o sub-padrão Gerenciador de Conexão colaborou para a elaboração de dois aspectos, `AbstractAspectConnection` e `AspectConnection`. O primeiro, apresentado na Figura 5, é abstrato e genérico o bastante para poder ser reutilizado em outros contextos. Ele possui um ponto de corte abstrato denominado `doConnection()`, cujo ponto de junção deve ser fornecido pelo aspecto concreto que o especializar. O segundo, `AspectConnection`, apresentado pela Figura 6, especializa o primeiro definindo o ponto de junção concreto, que indica o método que deve ser entrecortado para realizar a conexão, e também parâmetros de configuração dessa conexão na Sugestão `before`. A parte (a) na Figura 6 mostra que o ponto de junção concreto são chamadas ao

método `init()` de algum *servlet*, pois é nesse local do código da implementação orientada a objetos que havia uma chamada ao método responsável pela conexão com o banco de dados. Além disso, na Sugestão `before` desse aspecto, o método `setJDBC()` deve ser invocado do aspecto abstrato passando alguns parâmetros de configuração necessários à conexão, conforme apresentado na parte destacada com a letra (b) dessa mesma figura.

Optou-se neste caso, por criar dois aspectos, um abstrato e um concreto, pois notou-se que a maior parte do código relativo ao padrão Gerenciador de Conexão era independente da aplicação e poderia ser reutilizado em outros contextos.

6.2 Atributos e Métodos Específicos do Padrão

Essa segunda situação colabora para a criação de dois tipos de aspectos: genéricos e específicos. Como todos os atributos e métodos de persistência são indesejados nas classes de aplicação, devem ser retirados dessas, mas colocados novamente em tempo de composição por meio do conceito de Introduções do Aspect/J. Sendo assim criou-se um aspecto genérico, denominado `AspectStructure` (ver Figura 3), que agrega todos os atributos e métodos específicos do padrão que se repetem em todas as classes de aplicação.

A Figura 7 mostra o aspecto `AspectStructure` que, por meio do conceito de Introduções, insere novamente os atributos de persistência `tableName`, `keyName`, `colNames`, `colValues`, `colNamesException` e `cols` em todas as classes de aplicação, pois são atributos provenientes do sub-padrão Gerenciador de Tabelas. O mesmo ocorre com os métodos `findAll()` e `findLike()`, que foram recebidos do sub-padrão Objeto Persistente e, `delete()` que foi recebido tanto do Objeto Persistente quanto do CRUD.

```
public aspect AspectStructure
{
       private String (Cliente || Aparelho || ... || Conserto).tableName;
       private String (Cliente || Aparelho || ... || Conserto).keyName;
       private Vector (Cliente || Aparelho || ... ).colNames = new Vector();
       private Vector (Cliente || ... || Conserto).colValues = new Vector();
       private Vector (Cliente || ... ).colNamesException = new Vector();
       private int    (Cliente || Aparelho || ... || Conserto).cols;
 ... métodos sets e gets
       public ResultSet (Cliente || Aparelho || ... || Conserto).findall ()
       {
          return AspectTableManager.findallDB(this.getTableName());
       }
       public ResultSet (Cliente || Aparelho ... || Conserto).findlike ()
       {...}
       public boolean (Cliente || Aparelho || ... || Conserto).delete ()
       {...} }
```

Figura 7 – Aspecto AspectStructure

Os aspectos específicos são criados quando o valor dos atributos de persistência são dependentes de cada classe da aplicação. Quando isso ocorre, deve-se criar um aspecto específico para cada classe de aplicação com um ponto de corte que retorne o valor desses atributos em um determinado momento, definido por uma Sugestão. O momento correto da Sugestão entrecortar o código deve ser o mesmo onde os atributos recebiam seus valores na implementação orientada a objetos.

O aspecto `AspectCliente`, apresentado na Figura 8, possui comportamento específico para a classe Cliente, pois deve definir valores de persistência para essa classe, tais como: nome da tabela, nome das colunas da tabela, nome da chave primária e número de colunas, como mostrado parcialmente na parte destacada com a letra (a). A parte destacada com a letra (b) mostra a atribuição dos valores para os atributos definidos anteriormente.

Como esse tipo de aspecto possui características específicas de cada classe de persistência, deve ser criado um aspecto como esse para cada uma delas. O ponto de corte `changesConstructor()` desse aspecto possui como tarefa interceptar chamadas ao construtor da classe Cliente e definir valores para os atributos de persistência dessa classe, como era feito no construtor da classe Cliente na implementação orientada a objetos.

```
public aspect AspectCliente
{
      pointcut changesConstructor(Cliente c): target (c) && execution
              (Cliente.new(int, String, String, int, String, String,…));
      after (Cliente c): changesConstructor(c)
       {   Integer id;
           c.setTableName("Cliente");
           c.setKeyName("codigo");
           c.setCols(10);                                   (a)
           c.setColNames("codigo");
           c.setColNames("nome");
          ...
           id = new Integer(c.getCodigo());
           c.setColValues(id);
           c.setColValues(c.getNome());                     (b)
           c.setColValues(c.getRua());
        …
    public boolean Cliente.save ()                {...}
    public Vector  Cliente.getPrimaryKeyClause()  {...}
    public Vector  Cliente.getPrimaryKeyValue()   {...}
}
```

Figura 8 – Aspecto AspectCliente

6.3 Métodos Específicos da Aplicação com Dependência do Padrão

A terceira situação ocorre quando são encontrados métodos específicos do padrão que possuem muita dependência das classes de aplicação. Nesse caso, deve-se avaliar a possibilidade de generalizar a parte relativa ao padrão colocando-a em um aspecto genérico. Se não houver possibilidade, deve-se retirar esse método da classe de aplicação em que ele se encontra e colocá-lo no aspecto específico dessa classe para que depois seja retornado a essa classe por meio de Introduções.

Como já comentado anteriormente, a Figura 7 mostra que o aspecto genérico `AspectStructure` introduz novamente os atributos e métodos, que são específicos do padrão, em todas as classes de aplicação. Porém, os métodos `findLike()` e `delete()` foram adaptados para que pudessem ser generalizados nesse aspecto. Essa adaptação consistiu na criação de dois métodos auxiliares em cada aspecto específico das classes de aplicação, denominados de `getPrimaryKeyClause()` e `getPrimaryKeyValue()`, que são responsáveis por retornar o atributo que representa a chave primária da tabela e o valor desse atributo, respectivamente. A Figura 9 mostra o corpo do método `delete()` utilizando os métodos auxiliares citados acima para que a generalização possa ser feita. Antes da generalização, esse método referenciava dados específicos de alguma tabela, tornando-o dependente das descrições do banco de dados.

O processo de adaptação realizado com os métodos `delete()` e `findlike()` não pode ser feito com o métodos `save()`, devido à sua alta dependência dos dados da aplicação. Esse método possui como tarefa invocar métodos de persistência da classe `TableManager` passando parâmetros com valores específicos de cada uma.

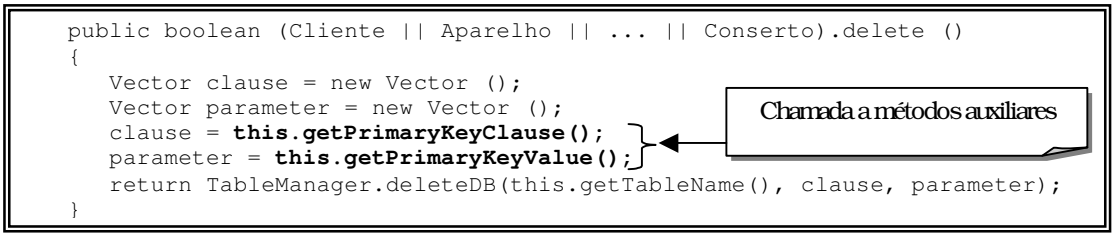

Figura 9 – Método delete( ) Generalizado

7. Composição da Funcionalidade Completa

O processo apresentado anteriormente separa o código que implementa o interesse do padrão de projeto, do código que implementa o interesse funcional da aplicação. Esse processo de separação se inicia identificando pontos de junção indesejáveis em que o interesse do padrão se junta com o interesse funcional. Esses pontos de junção entre os dois interesses, apesar de indesejáveis, são muito importantes, pois mesmo após a separação dos interesses, eles devem continuar existindo e permitindo o relacionamento das partes separadas, a fim de que a funcionalidade completa seja obtida. O que ocorre é que as técnicas de implementação orientadas a aspetos, como o Aspect/J, não eliminam esses pontos de junção, mas fornecem construções adequadas à modularização desses pontos, diferentemente das técnicas de implementação orientadas a objetos.

Os pontos de junção podem ser de dois tipos: estático e dinâmicos. Os estáticos, representados pelas Introduções, são em tempo de compilação e os dinâmicos, representados pelas Sugestões, ocorrem quando há chamadas a métodos, invocação de construtores ou lançamento de exceções. O importante é garantir que os pontos de junção entre os dois interesses separados continem existindo, porém, no aspecto.

Um exemplo de ponto de junção estático é mostrado na Figura 2. Os atributos de persistência destacados são pontos de relacionamento com o sub-padrão Gerenciador de Tabelas. Se esses atributos forem retirados dessa classe, ela não poderá mais ser persistida utilizando esse padrão. Assim, deve-se utilizar o conceito de Introduções em algum aspecto para que esses atributos retornem a essa classe em tempo de composição. A Figura 7 mostra o aspecto `AspectStructure`, que se utiliza desse conceito para retornar os atributos de persistência na classe Cliente no momento adequado. Dessa forma, o relacionamento entre os interesses continua existindo, mas sem entrelaçamento e espalhamento de códigos de diferentes interesses.

A Figura 2 também pode ser analisada para pontos de junção dinâmicos. Embora não apresentado nessa figura, no construtor dessa classe há alguns comandos que atribuem valores específicos para os atributos de persistência destacados com a letra (a). Como esses valores estão determinados no construtor, só terão realmente valores atribuídos quando um objeto dessa classe for instanciado. Esse momento da instanciação é um ponto de junção dinâmico que há entre os dois interesses. Dessa forma, algum aspecto deve realizar essa atribuição quando um objeto dessa classe for instanciado. Isso pode ser feito como mostra a Figura 8. O ponto de corte `ChangesConstrutor` determina que quando o construtor da classe Cliente for invocado, a sugestão `after()` atribui os valores de persistência aos seus devidos atributos, como mostra a parte destacada com a letra (a).

Com esses pontos de junção determinados e modularizados, a aplicação continua a ter a funcionalidade completa após a compilação, porém o código fonte das classes de aplicação contém apenas atributos e métodos relativos à sua funcionalidade, como mostra a Figura 10.

Na abordagem de Persistência com Aspectos apresentada por Rashid e Chitchyan[15], as classes de aplicação devem implementar uma interface, denominada `PersistentData`, que contém métodos que são utilizados para definir pontos de variação utilizados por outros aspectos. Isso implica que as classes de aplicação não estão totalmente desprovidas de métodos relacionados à persistência, já que toda classe deve implementar os métodos definidos em sua interface. Este é um ponto significativamente diferente do trabalho proposto aqui e pode ser o diferencial em relação à usabilidade.

```
public class Cliente {
    private int codigo;
    private String nome;
    private String rua;
    private int numero;
 ...

   public Cliente(int c, String n, ...) {
      this.codigo = c;
      this.nome = n; ...
   }
   public setCodigo(int c) {
       ...
   }
   métodos sets() e gets()
}
```

Figura 10 – Classe Cliente Após Modularização com Aspectos

8. Considerações Finais

A utilização do padrão de projeto Camada de Persistência implementado com técnicas de programação orientadas a objetos causa alguns problemas de entrelaçamento e espalhamento de códigos com diferentes propósitos. Este artigo apresentou uma possível solução para esse padrão utilizando o conceito da separação de interesses com Aspect/J e mostrou que esse conceito elimina a causa dos problemas de reusabilidade e evolução. Conforme apresentado na implementação com aspectos, as classes de aplicação ficam desprovidas de atributos e métodos de persistência eliminando o entrelaçamento e o espalhamento de código de persistência com código funcional. Isso faz com que as classes de aplicação sejam mais reusáveis, manuteníveis e com maior facilidade de extensão, já que deixaram de ser tão dependentes do padrão.

Assim como foi realizado em [15], buscou-se, sempre que possível, generalizar os aspectos para aumentar seus níveis de reuso, porém, alguns métodos e atributos referenciavam dados específicos do banco de dados e não poderiam ser generalizados. Por isso, houve necessidade de criar um aspecto específico para cada classe de aplicação persistente, causando um aumento significativo do número de unidades modulares para o sistema.

Diferentemente dos padrões propostos por Gama e outros [8], o padrão de projeto Camada de Persistência não atribui papéis às classes de aplicação. Porém, independentemente da existência de papéis, todos os seus sub-padrões possuem a estrutura de entrecorte citada por Hannemann e Kiczales[9]. Isso mostra que a existência desse tipo de estrutura não depende da existência de papéis, mas sim da própria natureza do padrão. A existência dessa estrutura de entrecorte no padrão justifica a utilização de técnicas de implementação apropriadas, como Aspect/J.

Com base no estudo realizado, os autores puderam generalizar um conjunto inicial de diretrizes que podem auxiliar engenheiros de software que tenham interesse em realizar projetos baseados em aspectos de padrões de projeto. As diretrizes apresentadas podem ser

refinadas e melhoradas à medida que a experiência de modularização baseada em aspectos de padrões de projeto aumente.

9. Referências Bibliográficas

[1] Camargo, V.V. Reengenharia Orientada a Objetos de Sistemas COBOL com a utilização de Padrões de Projeto e Servlets. Dissertação de Mestrado, Departamento de Computação da Universidade Federal de São Carlos, 2001.

[2] Camargo, V.V.; Prieto, A.G.; Penteado, R.A.D. Uma Experiência na Integração de UML, Padrão de Projeto e Servlet. In: Proceedings do CLEI 2000.

[3] Cagnin, M.I. Avaliação das Vantagens quanto à Facilidade de Manutenção e Expansão de Sistemas Legados Sujeitos à Engenharia Reversa e Segmentação. Dissertação de Mestrado, DC-UFSCar, 1999.

[4] Czarnecki, K., Eisenecker, U. W. Generative Programming. Addison Wesley, 2000.

[5] Elrad, T, et al. Discussing Aspects of AOP. Communications of the ACM, vol. 44, No. 19, pp. 33-38, outubro de 2001.

[6] Elrad, T, Filman R., Bader A. Aspect-Oriented Programming. Communications of the ACM, vol 44, No 10, pp. 29-32, 2001.

[7] Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H. Discussing Aspects of AOP. Communications of the ACM, october 2001/Vol. 44 No.10.

[8] Gama, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[9] Hannemann and Kiczales. Design Pattern Implementation in Java and AspectJ, Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 161-173, November 2002.

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G. Getting Started with AspectJ. Communications of the ACM, 44(10):59-65, October 2001.

[11] Noda, N. and Kishi, T. "Implementing Design Patterns Using Advanced Separation of Concerns", in Proceedings of OOPSLA 2001, Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa Bay, FL, October 2001.

[12] Soares, S., Laureano, E., Borba, P. Implementing Distribution and Persistence Aspects with AspectJ. In Proceedings of OOSPLA 2002, November 4-8, Seattle, Washington, USA.

[13] Soares, S., Borba, P. AspectJ - Programação orientada a aspectos em Java. Tutorial no SBLP 2002, 6º Simpósio Brasileiro de Linguagens de Programação. páginas 39-55. 5 a 7 de Junho 2002, PUC-Rio, Rio de Janeiro, Brasil.

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

[15] Rashid, A.; Chitchyan, R. Persistence as an Aspect. In Proceedins of 2nd International Conference on Aspect Oriented Software Development – AOSD 17 to 21 march of 2003. Boston – USA.