PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Programa de Pós-Graduação em Engenharia de Sistemas baseada em Tecnologias Java

USO DE PROGRAMAÇÃO ORIENTADA A ASPECTOS EM UMA APLICAÇÃO JAVA

Adão Francisco Correia Gonçalves

Poços de Caldas

2013

ADÃO FRANCISCO CORREIA GONÇALVES

USO DE PROGRAMAÇÃO ORIENTADA A ASPECTOS EM UMA APLICAÇÃO JAVA

Monografia apresentada ao curso de pós-graduação *lato sensu* em Engenharia de Sistemas Baseada em Tecnologia Java da Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de especialista em Engenharia de Sistemas Baseada em Tecnologia Java.

Poços de Caldas

2013

Adão Francisco Correia Gonçalves

USO DE PROGRAMAÇÃO ORIENTADA A ASPECTOS EM UMA APLICAÇÃO JAVA

Monografia apresentada ao curso de pós-graduação lato sensu em Engenharia de Sistemas Baseada em Tecnologia Java da Universidade Católica de Minas Gerais, como requisito parcial para obtenção do título de especialista em Engenharia de Sistemas Baseada em Tecnologia Java.

_____

Prof. Dr. Udo Fritzke Jr. (Orientador)

_____

Prof. Dr. Neil Paiva Tizzo

_____

Prof. M.Sc. Luiz Alberto Ferreira Gomes

Poços de Caldas, 05 de maio de 2013.

Dedico o este trabalho a minha esposa, pelo carinho e apoio.

AGRADECIMENTOS

A Deus, primeiramente.

A todos que contribuíram com este trabalho, de forma direta ou indireta, deixo meus sinceros agradecimentos, especialmente:

Ao professor e orientador Dr. Udo Fritzke Jr., pelas dicas e pelo acompanhamento na produção deste trabalho.

Ao tutor Tiago Antônio Rosa, pelos esclarecimentos durante todo o curso.

E aos colegas, pelas trocas de experiências via *Messenger*, *"noite adentro"*.

Ciência da Computação tem tanto a ver com o computador como a Astronomia com o telescópio, a Biologia com o microscópio, ou a Química com os tubos de ensaio. A Ciência não estuda ferramentas, mas o que fazemos e o que descobrimos com elas.

Edsger W. Dijkstra

RESUMO

Orientação a objeto é um conceito relativamente antigo, mas sua aplicação foi intensificada à medida que "o mundo" exigia cada vez mais softwares – e cada vez mais complexos – pela sua eficiencia em representar "coisas" inerentes ao software, através de objetos e suas funções. O paradigma de orientação a aspecto estende aquele conceito e o complementa com uma nova dimensão, o aspecto. Este trabalho apresenta um estudo sobre a Programação Orientada a Aspecto (POA), sua aplicabilidade na análise e desenvolvimento de sistemas Orientados a Objeto e como ela pode minimizar ou – preferencialmente – resolver o problema de entrelaçamento e espalhamento de código em um sistema. O estudo é direcionado a linguagem Java e, consequentemente, a linguagem *AspectJ*, por ser linguagem que possibilita a inserção de aspectos na linguagem Java. Tem-se como objeto de estudo a aplicação da orientação a aspecto em um projeto orientado a objeto, destinado a centros de formação de condutores – as auto-escolas – em fase de desenvolvimento.

Palavras-chave: Separação de Interesses. Interesses transversais. AspectJ. Programação Orientada a Aspecto.

ABSTRACT

Object orientation is a relatively old concept, but its application was intensified as "the world" demanded more and more software - and increasingly complex - for their efficient to represent "stuff" inherent to software, through objects and their functions. The aspect-oriented paradigm extends that concept and adds the aspect as a new dimension. This paper presents a study on Aspect Oriented Programming (AOP), its applicability in the analysis and development of Object Oriented systems and how it can minimize or - preferably - solve the problem of code entanglement and scattering in a system. The study is directed towards Java language and thus the language AspectJ, for being language that allows the insertion aspects in the Java language. As a case study, aims to study the application of aspect-orientation in an object-oriented design, to driver training centers - the driving schools - in development.

Key words: Separation of Concerns. Crosscutting concerns. AspectJ. Aspect-oriented Programming.

LISTA DE FIGURAS

LISTA DE SIGLAS

ACL - Access Control List

AOP - Aspect-Oriented Programming

AJDT - AspectJ Development Tools

CFC - Centro de Formação de Condutores

CNH - Carteira Nacional de Habilitação

CNPJ - Cadastro Nacional de Pessoa Jurídica

CPF - Cadastro de Pessoa Física

CONTRAN - Conselho Nacional de Trânsito

DAO - Data Access Object

DB - Database

IDE - Integrated Development Environment

M2E - Maven 2 Eclipse

OO – Orientação a Objetos

OOP - Object Oriented Programming

POO – Programação Orientada a Objetos

RF - Requisitos Funcionais

RNF - Requisitos Não Funcionais

RENACH - Registro Nacional de Carteira de Habilitação

SoC - Separation of Concerns

SUMÁRIO

1 INTRODUÇÃO

Historicamente a programação de computadores evolui à medida que problemas novos surgem e os paradigmas, então praticados, não os resolvem de maneira eficiente. Nesse contexto a Programação Orientada a Aspectos (POA) - *Aspect-Oriented Programming* (AOP) - foi criada para propor soluções às limitações encontradas no desenvolvimento de software, tais como o "Entrelaçamento de Código" e "Espalhamento de Código", que não são resolvidos apenas com o paradigma de Programação Orientada a Objetos (POO) - *Object-Oriented Programming* (OOP)[1] - ou Programação Estruturada. Neste trabalho pretende-se utilizar a AOP para implementar os requisitos não funcionais do sistema de forma mais eficiente.

Para facilitar a compreensão, manutenção e reuso de código, pesquisadores em engenharia de software têm investido na separação de interesses[2] através do uso de diferentes abstrações e modelos, realçando diferentes visões do sistema (SILVA; LEITE, 2005).

O princípio de Separação de Interesses (*Separation of Concerns* - SoC) baseia-se na divisão de um problema em partes menores e de menor complexidade, permitindo uma análise de um interesse por vez. (NELSON, 2012). Todavia, é necessário visualizar o sistema como um todo, com todos seus interesses, para melhor entendê-lo (SILVA; LEITE, 2005). Tão importante quanto separar os interesses é compô-los. Muitas vezes estes interesses são fortemente relacionados, entrelaçados e/ou sobrepostos, influenciando ou restringido uns aos outros, sendo chamados de interesses transversais[3] (SILVA; LEITE, 2005).

A AOP é um paradigma que estende a OOP (e outros paradigmas, como o estruturado, por exemplo) introduzindo novas abstrações. Estes novos elementos são destinados a suprir deficiências na capacidade de representação de alguns interesses (WINCK; GOETTEN, 2012). Portanto, é importante considerar os interesses transversais, proposta pela AOP, desde a fase de projeto.

O desenvolvimento estruturado e o orientado a objetos propõem separação do sistema em módulos de forma diferentes. No modelo estruturado a separação de interesse considera as funcionalidades do programa, onde cada função é implementada em um único módulo. Na

[1] Programação Orientada a Objetos (OOP) é uma "técnica de programação que focaliza os dados (= objetos) e interfaces com esses objetos". (HORSTMANN; CORNELL, 2010).
[2] Neste trabalho utilizaremos o termo interesse, do inglês *concern*, como sinônimo de característica a ser provida por um software.
[3] Do inglês *crosscutting concerns*. Neste trabalho usaremos o termo interesses transversais como tradução para *crosscutting concerns*.

orientação a objetos a separação é feita em termos de dados e depois em termos de funções que utilizam esses dados (NELSON, 2012).

A Figura 1 apresenta uma representação de separação de interesses.

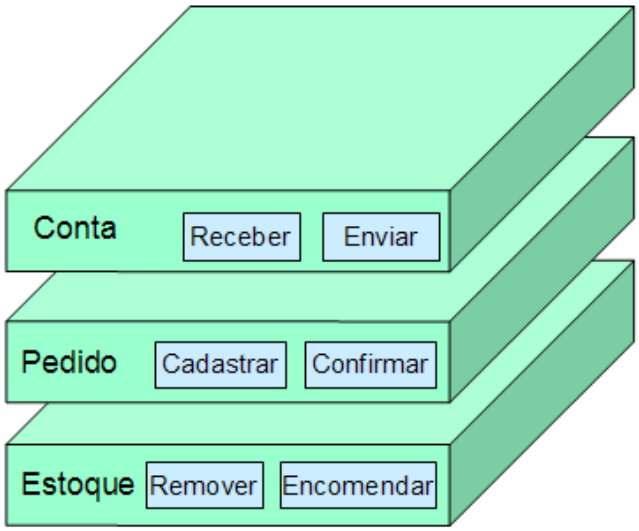

Figura 1: Separação de Interesses proposta pela OO
Fonte: NELSON, 2012.

Conforme Nelson (2012), a orientação a objetos melhorou as possibilidades de separação de interesses, mas ainda existem deficiências no atendimento de alguns.

Os diagramas, apresentados nas figuras 2 e 3 mostram uma representação gráfica do código do Sistema para Auto-Escolas. Cada coluna representa um módulo do sistema, sendo que o tamanho de cada coluna mostra o número proporcional de linhas de código daquele módulo (NELSON, 2012).

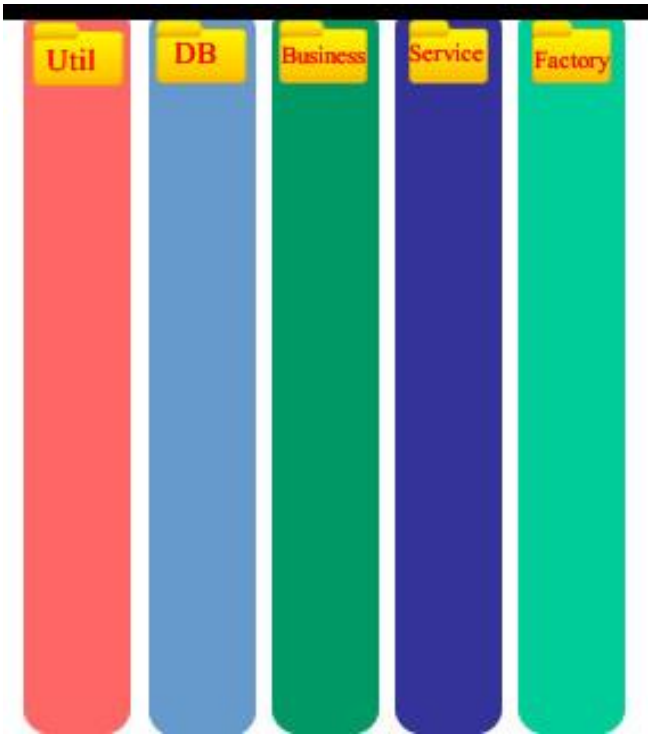

Figura 2: Distribuição responsabilidade em módulos
Fonte: criado pelo autor

Para exemplificar o conceito de separação de interesses em um sistema, estendamos ao nível da arquitetura. Considere, na Figura 2, que cada cor corresponda ao código desenvolvido (em classes, interfaces, etc.) em um determinado módulo. Representando, portanto, a responsabilidade atribuída a ele. Portando o código vermelho são de classes utilitárias, o azul claro código referente à manipulação do bando de dados, verde-escuro para regras de negócio, azul-escuro para implementação de serviços e verde-claro para módulo *factory*, que gerenciará os demais módulos. Não é interessante ter código vermelho no módulo azul-escuro, pois não é sua "responsabilidade".

O problema ocorre quando considerarmos a funcionalidade que guarda registros para auditoria, ou registrar histórico do comportamento do sistema (*logging*), ou ainda verificar permissão de acesso à funcionalidade. Então vê-se que os códigos responsáveis por essas funcionalidades estão espalhados por quase todos os módulos (NELSON, 2012). Problema exemplificado na figura 4, considerando as faixas amarelas da figura como o código responsável pelo registro de log do sistema, por exemplo.

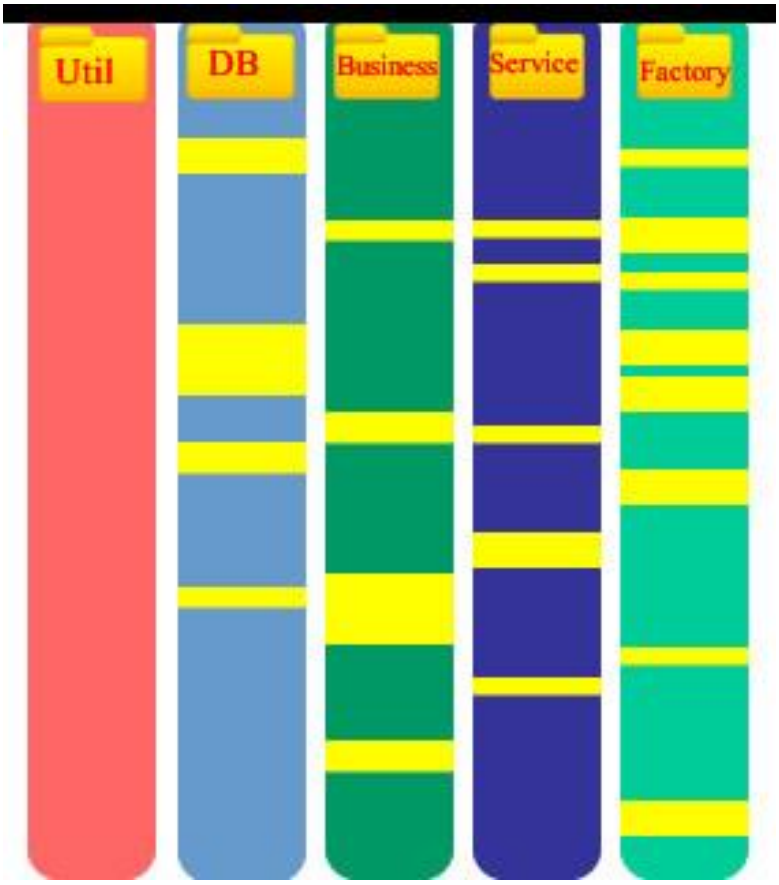

Figura 3: Implementação do Logging do sistema
Fonte: criado pelo autor

Na terminologia da AOP, diz-se que a função de registro de log do sistema é um interesse entrecortante (ou interesses transversais - crosscutting concerns), porque a sua implementação "corta" a estrutura de módulos do sistema. Praticamente todo programa orientado a objetos não-trivial contém interesses transversais. (NELSON, 2012).

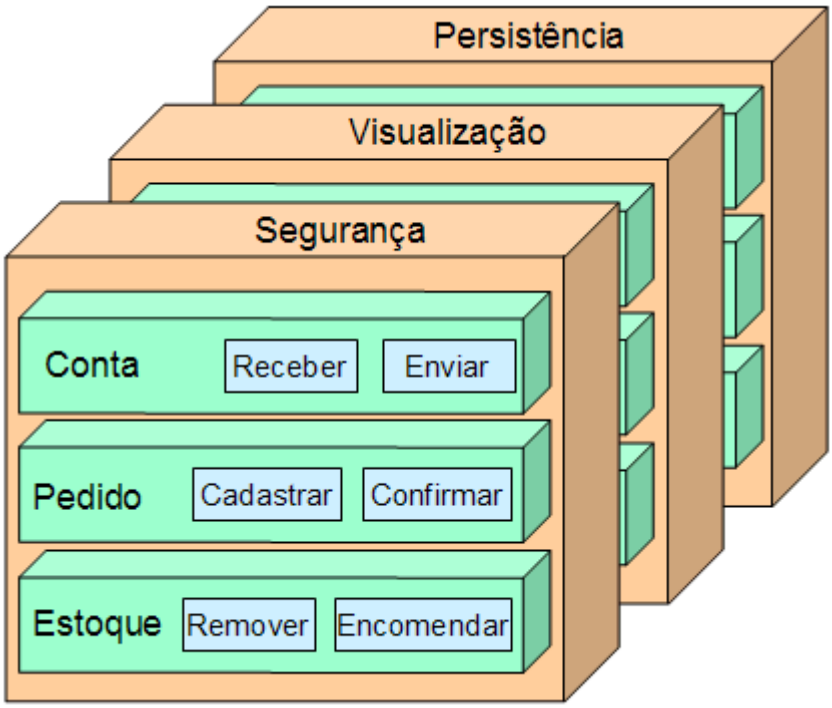

Figura 4: Separação de Interesses AOP
Fonte: NELSON, 2012.

A figura 4 representa o objetivo da AOP, conforme Nelson (2012), pois propõem encapsular interesses entrecortantes em aspectos separados do restante do código.

De forma abstrata, a AOP introduz uma terceira dimensão a OOP. Os conceitos de OO decompõem um sistema em objetos e métodos específicos, com a AOP a decomposição é realizada em objetos e métodos comuns a um interesse (NELSON, 2012).

2 PROGRAMAÇÃO ORIENTADA A ASPECTOS (AOP)

A programação orientada a aspectos foi criada em 1997, em uma divisão de pesquisa da Xerox, em Palo Alto, por Gregor Kiczales e sua equipe composta por: John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier e John Irwin (WINCK; GOETTEN, 2012). Eles tinham como objetivo:

> Construir uma abordagem que fosse um conjunto não necessariamente homogêneo, que permitisse à linguagem de programação, composta por linguagem central e várias linguagens específicas de domínio, expressar de forma ideal as características sistêmicas (também chamadas de ortogonais ou transversais) do comportamento do programa (WINCK; GOETTEN, 2012).

Esta abordagem recebe o nome de meta-programação (WINCK e GOETTEN, 2012), e o produto desta pesquisa de Kiczales e sua equipe, foi a AspectJ, a primeira linguagem de programação orientada a aspectos.

A Orientação a Aspectos propõe facilitar, principalmente, a implementação de requisitos não funcionais, através da separação e composição dos interesses transversais (SILVA; LEITE, 2005).

Na Figura 5 são demonstradas a separação e posterior composição dos interesses possibilitada pela AOP, "onde cada nuvem satisfaz uma característica sistêmica e os seus componentes, separados e bem definidos, podem ser reutilizados de modo mais eficiente" (ANDRADE; SILVA; ARAUJO, 2012).

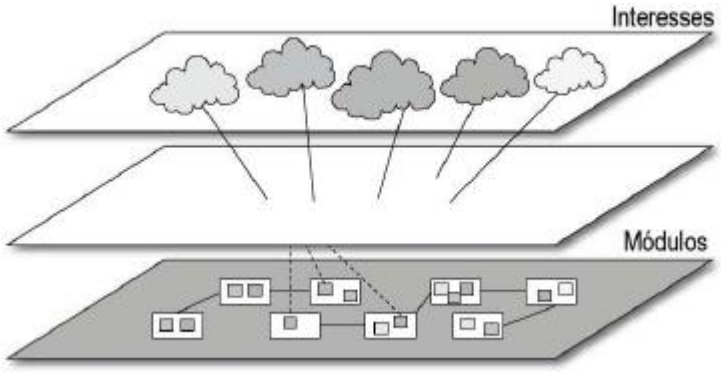

Figura 5: Separação de Interesses com AOP
Fonte: ANDRADE, SILVA E ARAUJO, 2012.

Na Figura 6 são apresentados os cinco elementos necessários para que um sistema utilize o paradigma de orientação a aspectos (ANDRADE; SILVA; ARAUJO, 2012). São eles: uma linguagem de componentes, uma ou mais linguagens de aspectos, programas escritos em linguagens de componentes, programas escritos em linguagens de aspectos e um combinador de aspectos.

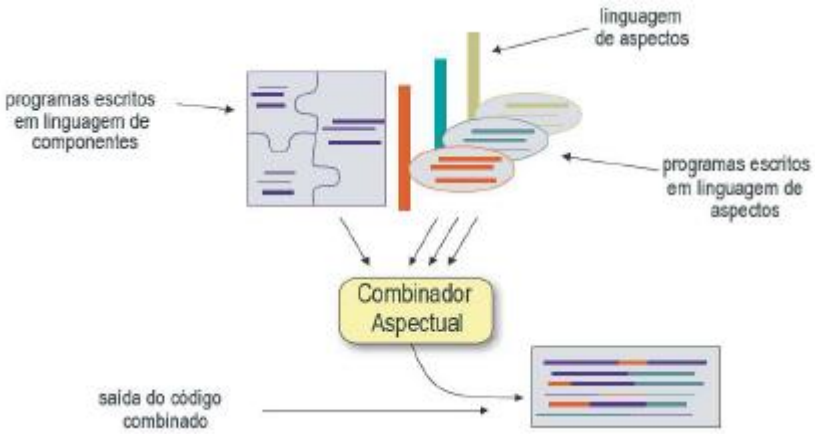

Figura 6: Composição de um sistema orientado a aspectos
Fonte: ANDRADE, SILVA E ARAUJO, 2012.

3 AOP EM JAVA - AspectJ

A linguag em AspectJ possibilita a separação dos interesses transversais através da inserção dos aspectos à OOP (SILVA; LEITE, 2005). A composição é realizada "através de um combinador denominado *weaver*, que é responsável por pré-processar o código orientado a objetos, inserindo ou modificando os objetos com o comportamento dos aspectos" (SILVA; LEITE, 2005).

Os interesses transversais são divididos em dois tipos (LADDAD, 2003): estáticos e dinâmicos. Os interesses transversais dinâmicos são maioria em um sistema, comparados com transversais estáticos, e são caracterizados por "combinar" um novo comportamento ao programa em sua execução. Já nos estáticos as alterações são realizadas na estrutura estática do programa (classes, interfaces, etc.) (LADDAD, 2003). Segundo Laddad (2003) a função mais comum dos interesses transversais estáticos é de apoiar os transversais dinâmicos, que é a maior parte dos interesses transversais em AspectJ.

3.1 Pontos de junção (join points)

Um *join point*, em AspectJ, é um ponto de identificação no sistema (LADDAD, 2003). São utilizados para alterar a estrutura dinâmica de um programa em tempo de execução.

3.2 Pontos de atuação (pointcuts)

Um *pointcut* indica um ou mais *join points* onde um determinado comportamento (*advice*) será inserido.

3.3 Adendo (advice)

Os Advices "descrevem o comportamento (trecho de código) a ser inserido nos *joinpoints*" (SILVA; LEITE, 2005). Podem ser executados antes (*before*), depois (*after*), ou antes e depois (*around*) de um *join point*.

3.4 Aspectos

Um Aspecto é para AspectJ o que uma classe é para Java, a unidade central da linguagem (LADDAD, 2003).

3.5 Exemplo prático

Código produzido no ambiente integrado de desenvolvimento (IDE – sigla de *Integrated Development Environment*) Eclipse, acrescido do plugin AJDT [4] que fornece suporte ao desenvolvimento com o AspectJ.

Classe Java "AspectJPrimus" executável e com o método "primeiroOla()", que imprime o texto "Executado " (Figura 7).

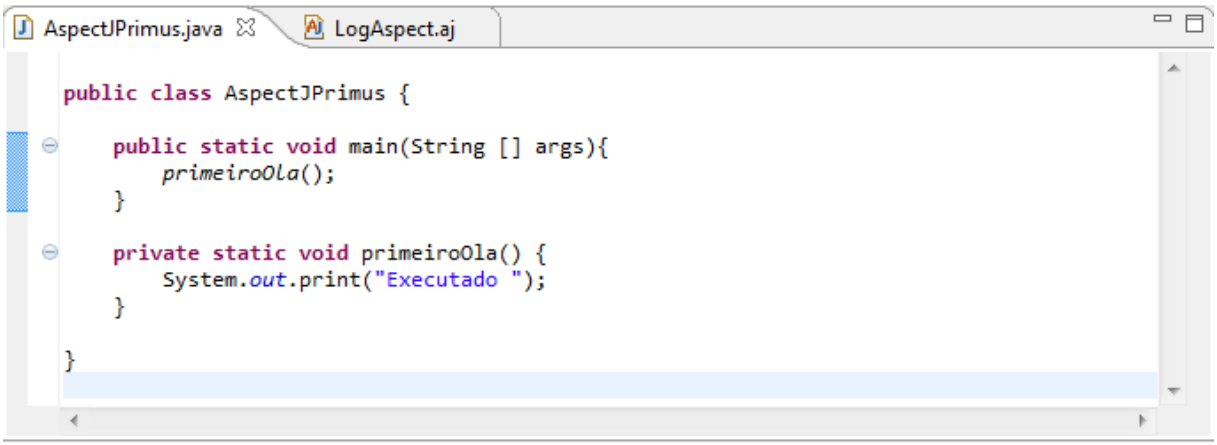

Figura 7: Classe Java - AspectJPrimus
Fonte: Produzido pelo autor

No Aspecto *LogAspect* é implementado o *pointcut* de nome "ptl", que indica o *join point* " * primeiroOla()" onde será aplicado o *advice after* (Figura 8).

[4] AJDT: AspectJ Development Tools, plugin disponível em <www.eclipse.org/ajdt>.

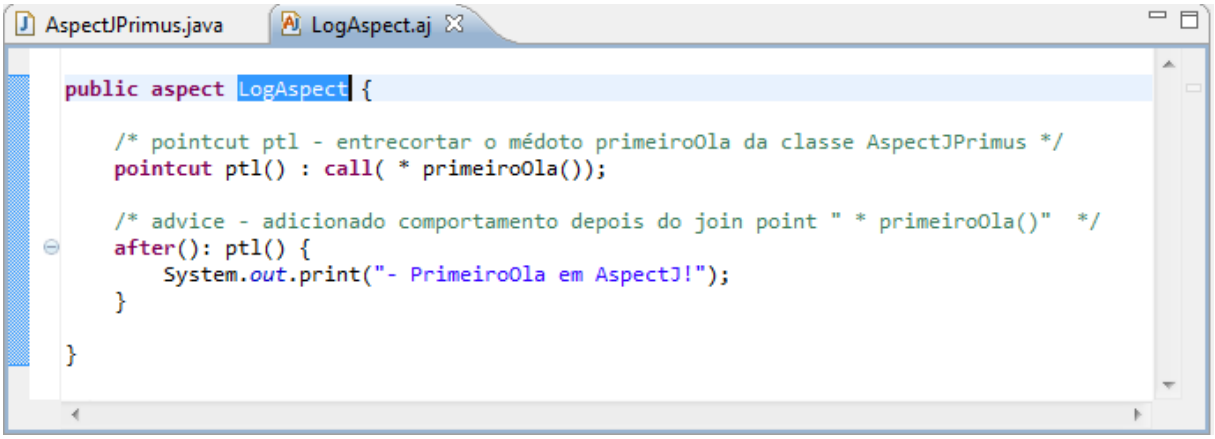

Figura 8: Aspecto AspectJ - LogAspect
Fonte: Produzido pelo autor

Quando o método "*primeiroOla*()" for chamado será imporesso: **`- PrimeiroOla em AspectJ!`**

4 ESTUDO DE CASO

O estudo de caso será baseado em um sistema destinado às empresas do ramo de Centro de Formação de Condutores (CFC).

A proposta do sistema é auxiliar nas tarefas administrativas e operacionais diárias de um CFC. Dentre elas: o cadastro completo de alunos, instrutores, veículos, emissão de contrato do aluno, relatório com agenda de aulas por aluno e por instrutor, contas a receber, livro caixa dentre outras detalhadas nos apêndices A e B deste trabalho. (GONÇALVES *et al*. 2009).

Os apêndices A e B incluem parte da documentação do sistema (em desenvolvimento) utilizado como estudo de caso para este trabalho. Especificamente, o apêndice A é baseado nos requisitos funcionais e não funcionais do sistema e no apêndice B são apresentados os modelos com diagramas de pacotes e de classes.

Nas subseções seguintes é descrito como a AOP foi integrada ao projeto.

4.1 Interesses Transversais (*crosscutting concerns*)

Uma vez identificados os interesses transversais – com base nos requisitos não funcionais do sistema - a AOP propõem separá-los codificando-os em um módulo a parte, por exemplo. Foram identificados os seguintes interesses transversais: *Logging* da aplicação, Controle Transacional e Lista de Controle de Acesso – ACL (*Access Control List*).

4.1.1 Logging

Fazer o *logging* do sistema é uma forma de registrar – em segundo plano – os comportamentos do mesmo.

Para este projeto será utilizado o *framework* log4j[5] para registrar o comportamento do sistema. Através dos seus níveis de *logging* (*DEBUG*, *INFO*, *WARN*, *ERROR* e *FATAL*) (APACHE, 2012) será possível verificar avisos e erros de codificação no ambiente de desenvolvimento e ajudará na manutenção do sistema quando estiver em produção.

4.1.2 Controle Transacional

O controle transacional é necessário em situações em que seja necessária a confirmação do sucesso da execução de várias operações. Caso uma (ou mais) falhe, as ações das demais devem ser canceladas ou desfeitas.

4.1.3 Lista de Controle de Acesso (ACL)

Na área de Ciência da Computação uma Lista de Controle de Acesso (também conhecida como ACL) é definida com uma lista de quem tem permissão de acessar determinado serviço ou função. Por exemplo, o usuário "adao" tem permissão para visualizar usuários cadastrados no sistema, mas não tem permissão de incluir, alterar ou apagar qualquer usuário. Este controle pode ser definido em uma ACL.

4.2 Uso do AspectJ ao Projeto

O projeto, alvo deste estudo de caso é desenvolvido na linguagem Java, consequentemente os interesses transversais serão desenvolvidos em AspectJ.

A proposta é, através da AOP, em um só módulo programar o *logging* do sistema de forma mais simples. E para atender aos interesses transversais do sistema, foi acrescentado o módulo *autoescola-aspects* com essa responsabilidade (Figura 9). Os demais módulos apresentados da figura 9 são descritos no Apendice B deste trabalho.

[5] Apache Logging Services, projeto de código aberto para logging de sistemas, desenvolvido pela Apache Foundation.

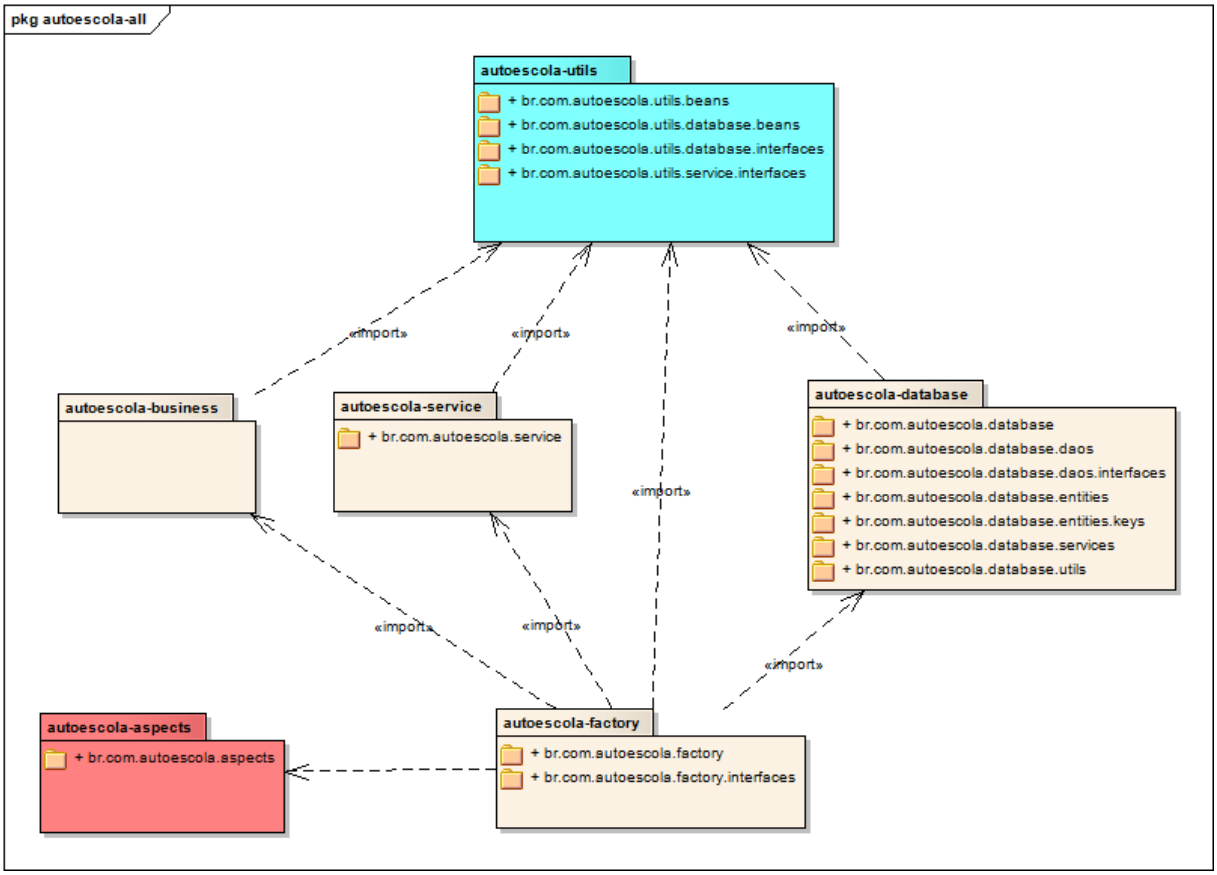

Figura 9 : Diagrama de Pacotes do Projeto Auto Escola com módulo de Aspectos
Fonte: Criado pelo autor

Os aspectos contidos no módulo *autoescola-aspects* serão inseridos no contexto da aplicação através do módulo de gerenciamento *autoescola-factory*.

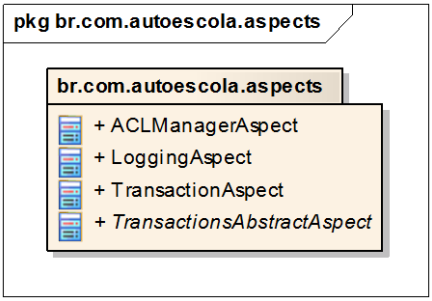

Figura 10: Módulo dos Aspectos
Fonte: criado pelo autor.

4.2.1 Logging em AspectJ

O *logging* do sistema é desenvolvido utilizando a biblioteca *Log4J (Figura 11)*. Na figura 12 pode ser observado o uso da classe *org.apache.log4j.Logger* na classe *GenericDao*.

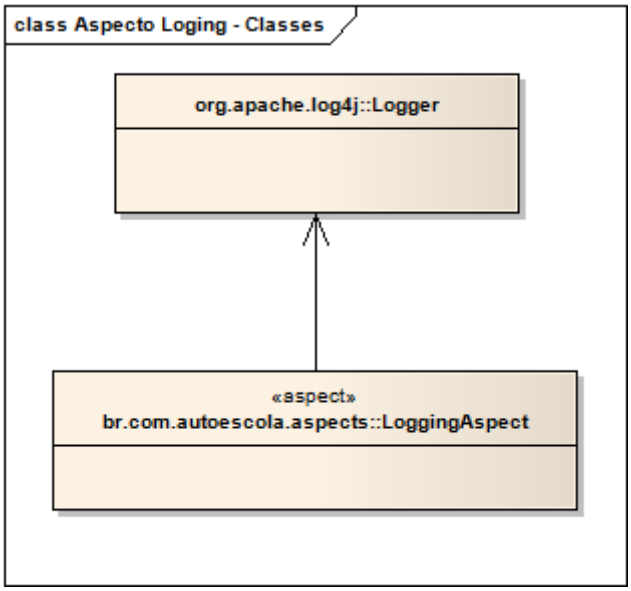

Figura 11: Modelo de classes do aspecto LoggingAspect

A *GenericDao* é uma classe abstrata que é estendida pelas classes DAOs concretas. É criado o objeto *logger*, da classe *Logger*, no escopo da classe *GererricDao* (Figura 12). É criado um *log* informativo (*INFO*) sempre que um DAO é criado, registrando qual entidade (*Entity*) o DAO está relacionado. Uma mensagem parecida com a seguinte será impressa:

```
20-10-2012 22:34:18,828; [main]; INFO; br.com.autoescola.database.daos.CargoDao;
- Criando um DAO para persitencia da classe [class
br.com.autoescola.database.entities.CargoEntity].
```

```java
public abstract class GenericDao<T extends BaseEntity, ID extends Serializable>
        implements IGenericDao<T, ID> {

    /** Logger. */
    private final Logger logger = Logger.getLogger(this.getClass());

    /**  Usado para retornar informações sobre a classe (entity).  */
    private final Class<T> persistentClass;

    /** Session with the database. */
    private Session session;

    @SuppressWarnings("unchecked")
    public GenericDao() {
        this.persistentClass = (Class<T>) ((ParameterizedType) this.getClass()
                .getGenericSuperclass()).getActualTypeArguments()[0];

        this.session = this.getCurrentSession();

        if (this.logger.isInfoEnabled()) {
            this.logger.info(String.format(
                    "Criando um DAO para persitencia da classe [%s].",
                    this.getPersistentClass()));
        }
    }
```

Figura 12: Implementação da Classe Logger na classe GenericDao
Fonte: Criado pelo autor

Em AspectJ, o *log* é separado e codificado em um aspecto (Figura 13).

```java
public aspect LoggingAspect {

    private final Logger logger = Logger.getLogger(this.getClass());

    /* Pointcuts: entrecortar o quando um objeto DAO é criado */
    private pointcut pctGenericDao():
        execution( public br.com.autoescola.database.daos.*.new() );

    /* advice */
    before(): pctGenericDao(){

        Type type = this.getClass().getGenericSuperclass();
        ParameterizedType pt = (ParameterizedType) type;

        if (this.logger.isInfoEnabled()) {
            this.logger.info(String.format(
                    "Criando um DAO para persitencia da classe [%s].",
                    pt.getActualTypeArguments()[0] ));
        }
    }
```

Figura 13: Implementação da Logger no Aspecto *LoggingAspect*
Fonte: Criado pelo autor

No aspecto *LoggingAspect* o *pointcut*[6] *pctGenericDao* entrecorta o método construtor das classes concretas dos DAOs. A expressão indica: *execution* – quanto o método for executado; modificador de acesso *public*; nome da classe totalmente qualificada, neste caso qualquer classe do pacote *br.com.autoescola.database.daos*; *new* – quando objeto for instanciado.

O *advice before*() insere um trecho de código antes da execução do método. Um *advice* pode, ainda, inserir seu trecho de código depois da interceptação de um *join point* (*advice* a*fter*); ou somente se executar normalmente, sem lançar exceções (*after() returning*); ou quando lançar alguma exceção (*after() throwing*). Quanto é necessário utilizar o estado de antes e depois da execução pode ser usado a*round()* (XEROX CORPORATION, 2012).

[6] Esta é uma das formas de codificação sugerida na documentação do Spring AOP.

O código do aspecto *LoggingAspect* (Figura 14) é ilustrativo e ainda não é o suficiente para fazer o *logging* de toda aplicação, mas a título de exemplo, é representado abaixo.

```
public aspect LoggingAspect {

    private final Logger logger = Logger.getLogger(this.getClass());

    /* Pointcuts: entrecortar o quando um objeto DAO é criado */
    private pointcut pctGenericDao():
        execution( public br.com.autoescola.database.daos.*.new() );

    /* advice */
    before(): pctGenericDao(){

        Type type = this.getClass().getGenericSuperclass();
        ParameterizedType pt = (ParameterizedType) type;

        if (this.logger.isInfoEnabled()) {
            this.logger.info(String.format(
                    "Criando um DAO para persitencia da classe [%s].",
                    pt.getActualTypeArguments()[0] ));
        }
    }

    /* Pointcuts: entrecortar os services do módulo database */
    private pointcut pctServicesDB():
        execution( public br.com.autoescola.database..*Service.new() );

    before(): pctGenericDao(){

        ParameterizedType pType = (ParameterizedType) this.getClass()
                    .getGenericSuperclass();
        Type[] genericTypes = pType.getActualTypeArguments();

        if (this.logger.isInfoEnabled()) {
            this.logger
                .info(String
                    .format("Criando um service com Bean [%s], Entity [%s], "+
                            "keyBean [%s] e keyEntity [%s].",
                        genericTypes[0],
                        genericTypes[1],
                        genericTypes[2],
                        genericTypes[(genericTypes.length == 4) ? 3 : 2]));
        }
    }

    /* Pointcuts: entrecortar o método startApplication que inicia
     * a aplicação para acesso via RMI */
    private pointcut pctStartApplication():
        execution( * br.com.autoescola.factory.*.startApplication() );

    /* advice */
    before(): pctStartApplication(){
        if ( logger.isInfoEnabled() )
            logger.info("Servidor RMI INICIADA - Auto Escola!");
    }
```

Figura 14: Aspecto LoggingAspect
Fonte: Criado pelo autor

O código do *logging*, que era repetido na aplicação, poderá ser excluído do restante da aplicação evitando o espalhamento de código (Figura 15).

```
public abstract class GenericDao<T extends BaseEntity, ID extends Serializable>
        implements IGenericDao<T, ID> {

    /**  Usado para retornar informações sobre a classe (entity).  */
    private final Class<T> persistentClass;

    /** Session com database. */
    private Session session;

    @SuppressWarnings("unchecked")
    public GenericDao() {
        this.persistentClass = (Class<T>) ((ParameterizedType) this.getClass()
                .getGenericSuperclass()).getActualTypeArguments()[0];

        this.session = this.getCurrentSession();
    }
```

Figura 15: Classe abstrata GenericDao com *logging*
Fonte: Criado pelo autor

A mesma implementação ocorre na classe *GenericServiceWithKey* e nas demais classes nas quais se deseja realizar o *logging*.

4.2.2 Controle Transacional com AspectJ

No Sistema para Auto-Escola é utilizado o *framework* Hibernate [7] para realisar persistências no banco de dados. E para tratar métodos transacionais isolados, a exemplo dos métodos ***save, update*** e ***delete*** (Figura 16) da classe genérica **GenericDao**, é relativamente fácil. A complexidade aumenta quando surge a necesidade de um método transacional chamar outro método transacinal.

```
@Override
@SuppressWarnings("unchecked")
public ID save(T entity) {
    this.getSession().beginTransaction();
    Serializable id = this.getSession().save(entity);
    this.getSession().getTransaction().commit();

    return (ID) id;
}

@SuppressWarnings("unchecked")
@Override
public void update(T entity) {
    entity = (T) this.getSession().merge(entity);

    this.getSession().beginTransaction();
    this.getSession().update(entity);
    this.getSession().getTransaction().commit();
}

@SuppressWarnings("unchecked")
@Override
public void delete(T entity) {
    entity = (T) this.getSession().merge(entity);

    this.getSession().beginTransaction();
    this.getSession().delete(entity);
    this.getSession().getTransaction().commit();
}
```

Figura 16: Implementação comum de Transação no Hibernate
Fonte: Criado pelo autor

Nesse caso será necessário considerar os dois níveis de transação e somente o mais externo deve ficar responsável pelo gerenciamento da transação (COUTINHO, 2013).

Para o sistema em estudo os níveis de transação são apresentados no diagrama de sequência a seguir (Figura 17). No diagrama no primeiro cenário o método transacional *update(Bean)* é o gerenciador da transação, realizando o *commit* ou *rollback*. No segundo o método *inserts(Entity, Entity)* tem essa responsabilidade.

[7] Hibernate – Framework de Persistência Relacional para Java. Disponível em <http://www.hibernate.org>.

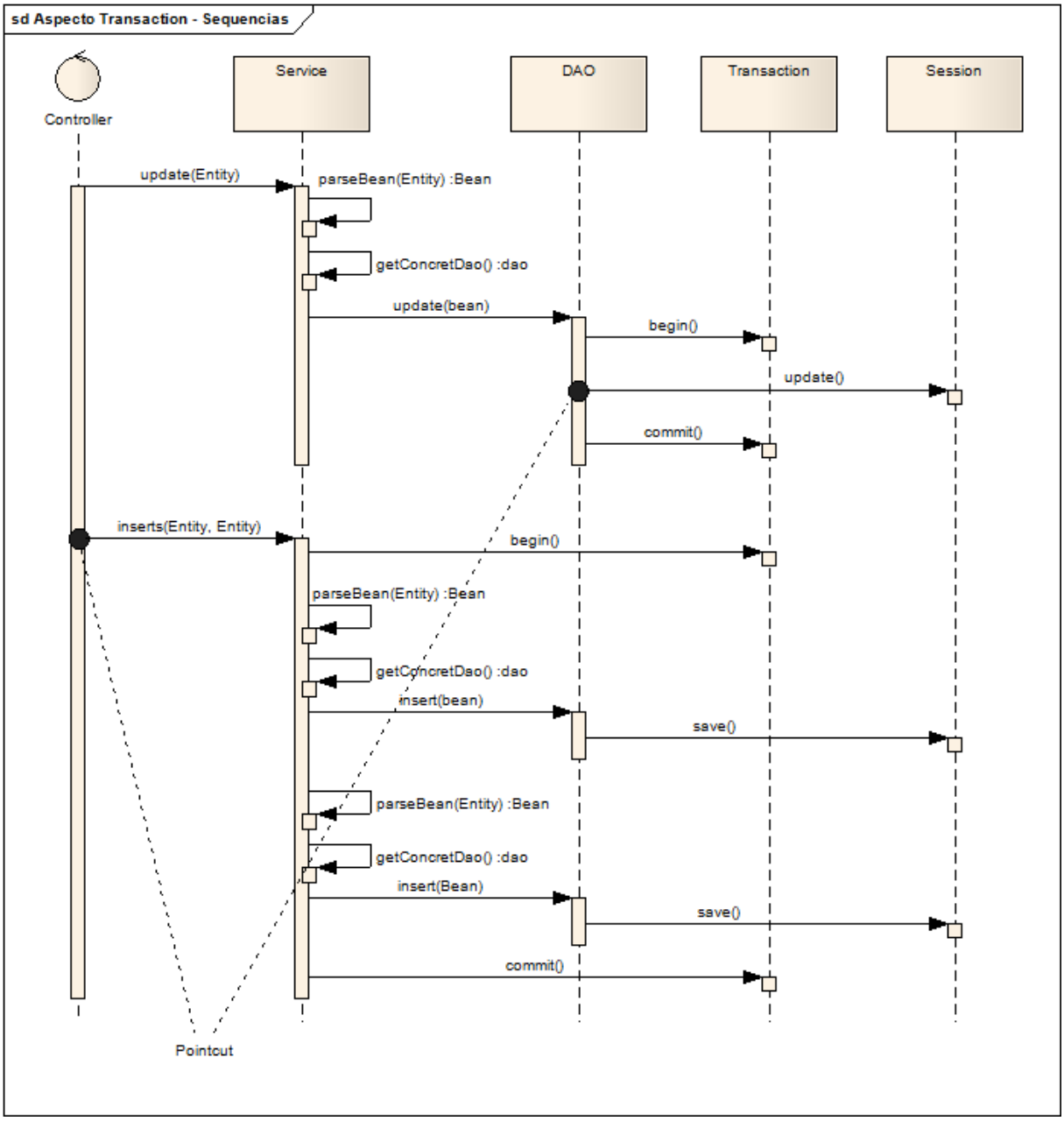

Figura 17: Diagrama de sequência de níveis de transações
Fonte: Criado pelo autor

A legenda *pointcut*, na Figura 17, indica possíveis "alvos" de *pointcuts*, pois é onde ocorre transações.

Portanto para o gerenciamento de transações com Hibernate foi criado o aspecto abstrato *TransactionsAbstractAspect* (Figura 20) com responsabilidade de capturar os métodos transacionais da aplicação e fazer o gerenciamento da transação do método mais externo. Para isso é preciso implementar, no aspecto concreto *TransactionAspect* (Figura 19), um *pointcut* para capturar os métodos transacionais. Um *advice*, implementado no aspecto

TransactionsAbstractAspect, chamará o gerenciamento de transação do Hibernate (*beginTransaction*(), *commit*() ou *rollback*()) (COUTINHO, 2012).

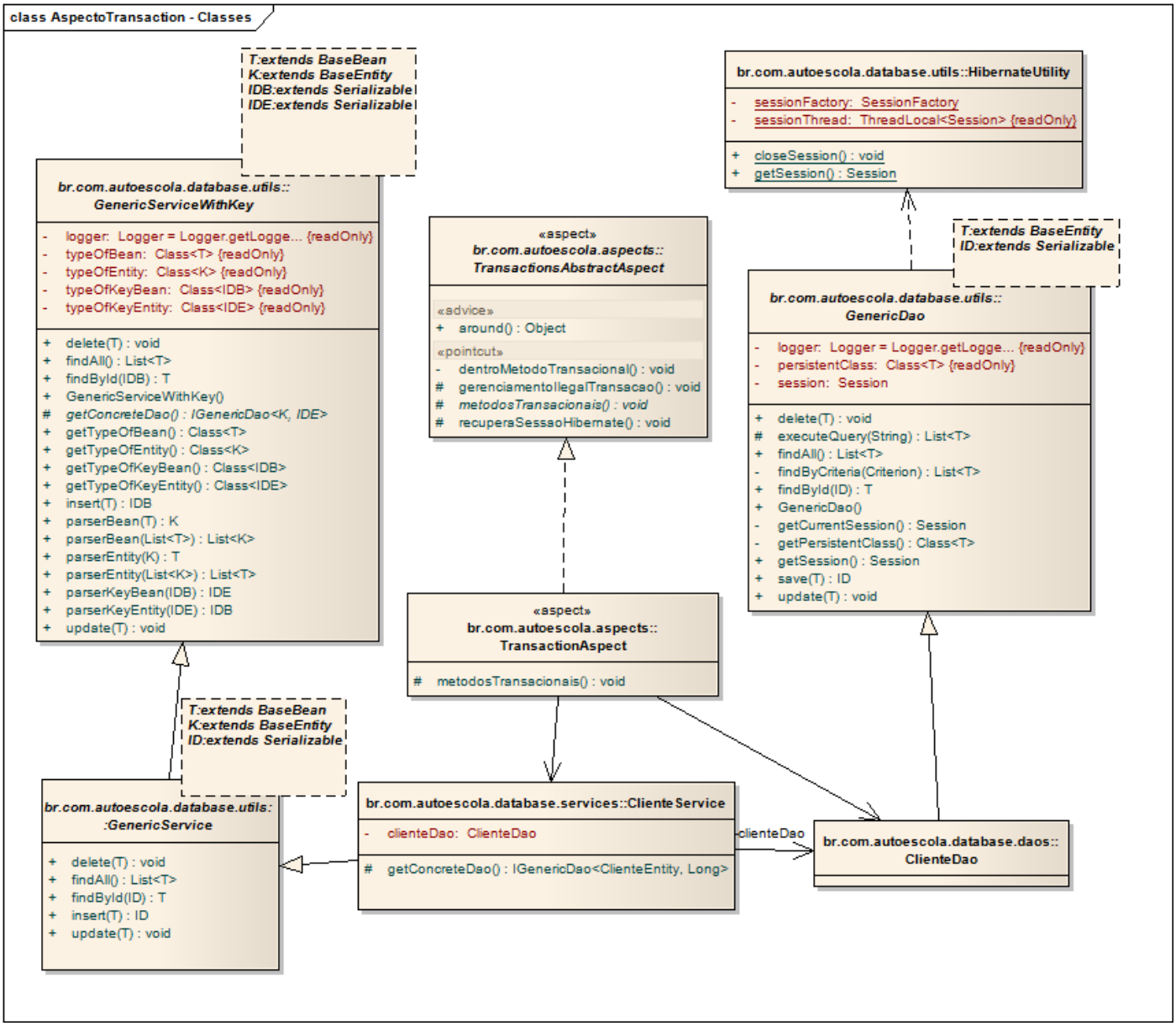

Figura 18: Diagrama de classes DAO com Aspecto de Transação
Fonte: Criado pelo autor

O código do aspecto é mostrado nas figuras abaixo, os *imports* e *package* foram omitidos:

```
public aspect TransactionAspect extends TransactionsAbstractAspect {
   /* Pointcuts: entrecortar os métodos transacionais da aplicação */
   protected pointcut metodosTransacionais():
        execution( public * br.com.autoescola.database.services..insert*(..) ) ||
        execution( public * br.com.autoescola.database.services..update*(..) ) ||
        execution( public * br.com.autoescola.database.services..delete*(..) ) ||
        execution( public * br.com.autoescola.database.services..inserts*(..) );
}
```

Figura 19: Aspecto concreto *TransactionAspect*
Fonte: Criado pelo autor

```
public abstract aspect TransactionsAbstractAspect {

    public abstract static class ContextoTransacaoHibernate {
        private Session session;
        private Object returnValue;

        public abstract void run();

        public Object getReturnValue() {
            return returnValue;
        }

        public void setReturnValue(Object value) {
            this.returnValue = value;
        }

        public Session getSession() {
            return session;
        }

        public void setSession(Session sessao) {
            this.session = sessao;
        }
    }

    protected abstract pointcut metodosTransacionais();

    protected pointcut recuperaSessaoHibernate() :
              call(org.hibernate.Session+ SessionFactory.*Session(..));

    private pointcut dentroMetodoTransacional( ContextoTransacaoHibernate contexto ) :
              cflow(execution(* ContextoTransacaoHibernate.run()) && this(contexto));

    Object around() : metodosTransacionais() &&
              !dentroMetodoTransacional(ContextoTransacaoHibernate) {

        ContextoTransacaoHibernate context = new ContextoTransacaoHibernate() {
            public void run() {
                try {
                    setReturnValue(proceed());
                    if (getSession() != null && getSession().isOpen()) {
                        getSession().getTransaction().commit();
                    }
                } catch (Exception e) {
                    if (getSession() != null && getSession().isOpen()) {
                        getSession().getTransaction().rollback();
                    }
                    throw new RuntimeException(e);
                } finally {
                    if (getSession() != null) {
                        getSession().close();
                    }
                }
            }
        };
```

```
    Object around(final ContextoTransacaoHibernate context) :
              recuperaSessaoHibernate() && dentroMetodoTransacional(context) {
        if (context.getSession() == null || !context.getSession().isOpen()) {
            context.setSession((Session) proceed(context));
            context.getSession().beginTransaction();
        }
        return context.getSession();
    }

    protected pointcut gerenciamentoIlegalTransacao() : (call(* Session.*Transaction(..)) ||
              call(* Transaction.*(..))) && !within(br.com.autoescola.aspects.*+);

    declare warning : gerenciamentoIlegalTransacao() :
              "Não é permitido usar métodos de gerenciamento de transação diretamente.";

}
```

Figura 20: Aspecto abstrato TransactionsAbstractAspect
Fonte: COUTINHO, 2012

Análise do código do aspecto *TransactionsAbstractAspect*:

- **ContextoTransacaoHibernate** – Classe interna utilizada para armazenar as informações para compartilhar no contexto de uma transação. Também é definido um método abstrato **run ()**, o qual será responsável por envolver o conteúdo original do método transacional com o código de gerenciamento da transação.

Definição de quatro *pointcuts* que serão utilizados no aspecto:

- O primeiro *pointcut*, **metodosTransacionais ()**, será o responsável por capturar os métodos da aplicação que estarão sob o controle transacional. Esse *pointcut* é abstrato e, portanto, não tem uma implementação. Isso porque os métodos transacionais são métodos específicos de uma determinada aplicação. Sendo assim, esse *pointcut* precisará de uma implementação específica para cada aplicação;
- O segundo pointcut, **recuperaSessaoHibernate()**, captura os métodos que são usados para a obtenção da Sessão do Hibernate. A implementação detecta as chamadas de qualquer método da classe **org.hibernate.SessionFactory** cujo nome termina com "Session" (Ex.: **openSession()**, **getCurrentSession()**, etc.) e possua um retorno do tipo **org.hibernate.Session**;
- O terceiro pointcut, **dentroMetodoTransacional()**, é utilizado para detecta a execução de qualquer método dentro do fluxo de execução de um método transacional. Esse *pointcut* também expõe o "Contexto Transacional" para que esse possa ser usado nos *advices*;

- O último *pointcut*, **gerenciamentoIlegalTransacao()**, captura chamadas ilegais aos métodos de gerenciamento de transação feitas pela aplicação. Esta implementação detecta chamadas a métodos da interface **org.hibernate.Transaction**, bem como métodos para obtenção/criação de transações a partir da interface **org.hibernate.Session**.
- Em seguida temos dois *advices* que serão executados "em torno" (**around**) dos *joinpoints* capturados: O primeiro *advice* é responsável por realizar as chamadas aos métodos de controle de transação do Hibernate nos momentos apropriados. A instrução **Object around() : metodosTransacionais() && !dentroMetodoTransacional(ContextoTransacaoHibernate)** indica que o *advice* deve ser executado para métodos transacionais que não estejam dentro do fluxo de outro método transacional. Dessa forma apenas o método transacional mais externo será substituído por esse *advice*. A chamada a **proceed()** invoca o conteúdo original do método que está sendo substituído pelo *advice* e salva seu retorno no atributo **returnValue** do contexto transacional. Logo em seguida verificamos se a sessão e transação estão válidas para realizar o **commit()**. Se ao invocar o conteúdo original do método, através de uma chamada a **proceed()** uma exceção for gerada, esta será capturada pelo bloco **catch** e então a transação será cancelada através da chamada ao método **rollback()**; em seguida uma **RuntimeException**, encapsulando a exceção original, será lançada para que o código que invocou o método transacional seja capaz de saber que algo deu errado na execução desse método. Por fim, retornamos o valor de retorno armazenado no contexto transacional; No segundo *advice* visa garantir que a mesma sessão será utilizada durante todo o contexto transacional. Para isso é substituído o conteúdo dos métodos capturados pelo *pointcut* **recuperaSessaoHibernate()** e que foram chamados dentro do fluxo de execução do contexto transacional.
- Finalmente temos uma declaração *declare warning* para os *joinpoints* capturados pelo *pointcut* **gerenciamentoIlegalTransacao()**. Isso irá gerar um *warning* de compilação para cada uso ilegal de métodos de gerenciamento de transação. Essa técnica é conhecida como *Policy Enforcement* e serve para garantir que as políticas definidas na arquitetura serão seguidas pelos desenvolvedores (COUTINHO, 2012).

4.2.3 Lista de Controle de Acesso (ACL) com AspectJ

A Lista de Controle de Acesso ou ACL (*Access Control List*) é a forma de gerenciar as permissões da aplicação com alta granularidade. ACL é implementado numa estrutura de árvore. Existe geralmente uma árvore de usuários e/ou grupos, e uma árvore de recursos (privilégios) o que permite tratar permissões de forma granulada. Na árvore abaixo é exemplificado, superficialmente, o ACL aplicado no sistema:

- Aplicação (Negado: todos)
    - Administradores (Permitido: usuários, clientes, pagamentos)
        - João (Permitido: contrato, funcionário, veículos)
        - Maria
    - Atendentes (Permitido: clientes)
        - José (Negado: excluir cliente)
    - Alunos (Permitido: Ver seus dados de aluno)
        - Ana

Os diagramas relacionados com ACL utilizando a AOP seguem nas figuras 21 e 23:

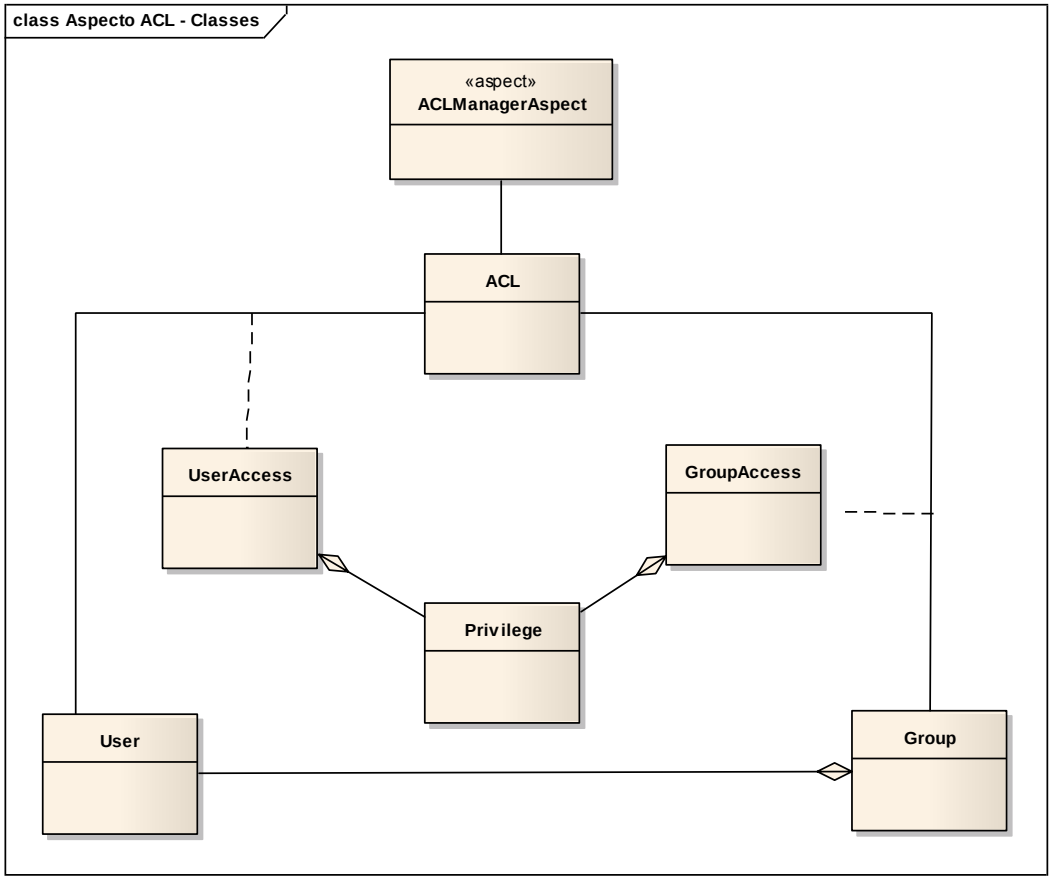

Figura 21: Diagrama de classes do ACL
Fonte: Criado pelo autor

A estrutura do ACL permitirá gerenciar as permissões de acesso a aplicação pela própria aplicação. Entretanto, no contexto da aplicação da AOP será implementado no aspecto ACLManager apenas a consulta as permissões de acesso do usuário autenticado na aplicação. Este aspecto percore a estrutura de árvore na qual o usuário está vinculado. Por exemplo: Aplicação->Atendentes->José. Cada um desses pontos tem permissões diferentes, e usa-se a permissão mais específica relacionada ao usuário José.

O aspecto *ACLManagerAspect* será implementado de forma parecida com o código da Figura 22, onde observa-se que os "recursos" – entenda-se recurso o conjunto objeto-ação (cliente-insert) - serão identificados como *joinpoint pctIsAllowed*. A permissão é validada no antes (*before*) que o método seja executado.

```
public aspect ACLManagerAspect {

    /* pointcut: identifica os "recursos"  */
    protected pointcut pctIsAllowed():
        execution( * br.com.autoescola..*Service.*(..) );

    before(): pctIsAllowed(){
        /* Deve Recuperar usuário na sessão do contexto da aplicação
         * Implementar o ACL do usuário
         * Checar permissão específica ao recurso
         * * implementação pendente *
         */
    };
}
```

Figura 22: Exemplo para Aspecto ACLManagerAspect
Fonte: Criado pelo autor

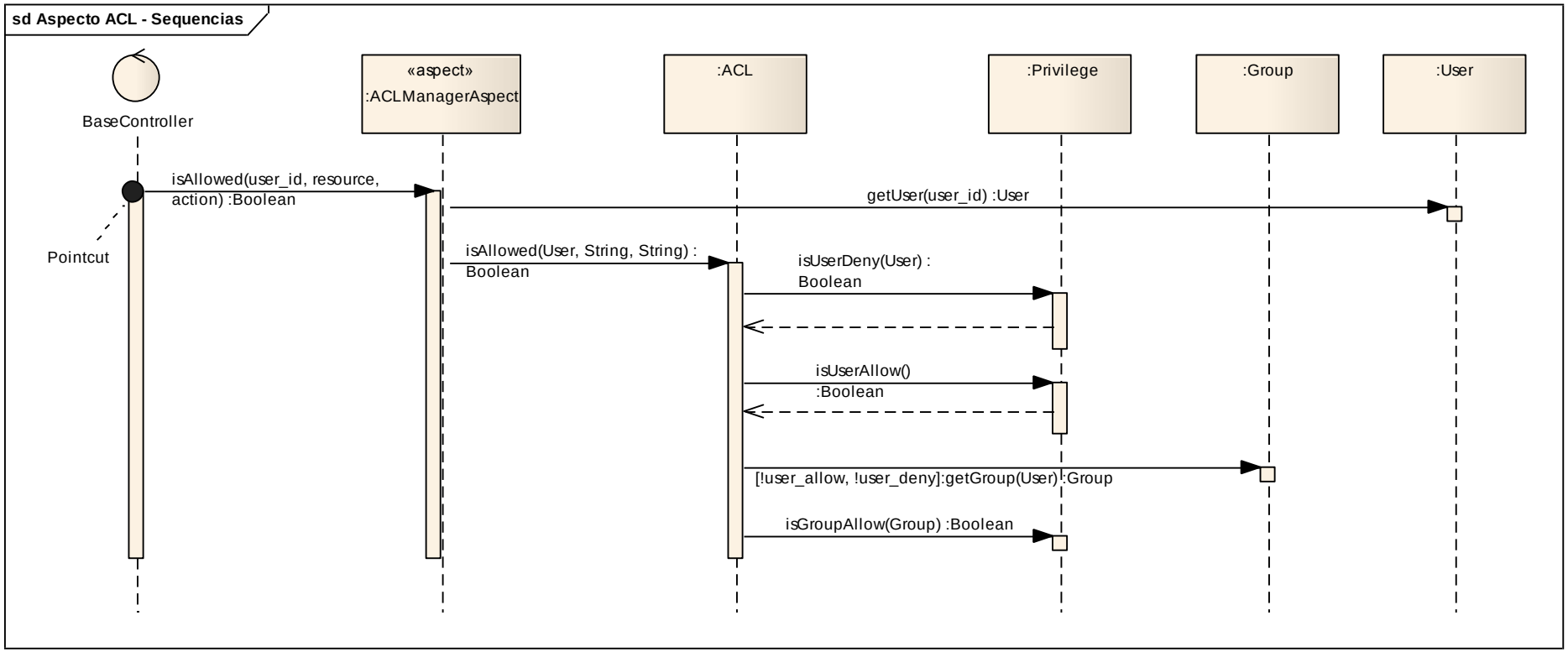

Figura 23: Diagrama de Sequencia do ACL com aspecto
Fonte: Criado pelo autor

Esta abordagem permite eliminar os trechos de código que se repete na aplicação para testaria o nível de permissão do usuário em cada ação tomada.

5 CONSIDERAÇÕES FINAIS

O princípio de separação e composição de interesses do software potencializa as possibilidades de se projetar um software flexível, de fácil compreensão, melhor para se manter e com número menor de linhas de código. Outro benefício do investimento em AOP é a possibilidade de identificar os interesses transversais, evitando o entrelaçamento e espalhamento de código e agrupá-los como partes de um módulo específico do sistema. A adoção desse princípio desde a fase de análise do projeto pode diminuir a complexidade do sistema em nível de modelagem, inclusive. Isso terá reflexos positivos na compreensão, manutenibilidade e reusabilidade do sistema.

Nesse contexto, a estrutura do projeto de desenvolvimento de sistema para Centros de Formação de Condutores (objeto do estudo de caso) foi repensada com intuito de considerar a separação de interesses, inclusive os transversais, inicialmente desconsiderados na análise orientada a objetos devido a limitações em suas representações como objetos do sistema. Como resultado prático, uma estrutura de sistema com melhor distribuição dos seus interesses em módulos. Foi definido um módulo específico para os aspectos, composto pelos aspectos *LoggingAspect, ACLManagerAspect, TransactionAspect* e *TransactionsAbstractAspect*.

O *logging* da classe *GenericDao* - observado na figura 12 - ocupa seis linhas de código. E no aspecto *LoggingAspect* (Figura 13) o *logging* é realizado com o mesmo número de linhas de código. Entretanto, o aspecto - que abrange todas as classes DAOs concretas - possibilita atingir cada método específico de cada uma destas classes. E com apenas o acréscimo da expressão "**|| execution(public br.com.autoescola.database.services.*())**" (no *pointcut pctGenericDao)* o *logging* será estendido a todas as classes *services* do mesmo módulo. Logo, o espalhamento de código foi, de fato, evitado e, consequentemente, "economizadas" várias linhas de código com utilização da AOP.

A AOP permitiu, ainda, resolver elegantemente um problema de gerenciamento de transações. Um cenário onde é necessário realizar mais de uma operação (de *insert, update* e/ou *delete*) na mesma transação, resume o problema. O gerenciamento desses tipos de transações de forma eficiente, como baixo acoplamento é uma tarefa difícil e complexa, mesmo utilizando a OOP e seus vários padrões de projeto. Este problema foi resolvido utilizando os recursos da AOP, por meio da linguagem AspectJ, como mostrado na sessão 4.2.2 deste trabalho. Os aspectos *TransactionsAbstractAspect* e *TransactionAspect* isola todo o código relativo ao controle transacional e possibilita o gerenciamento transacional em níveis

mais altos da aplicação. Evitando o entrelaçamento de código na aplicação e mantendo um baixo acoplamento.

REFERÊNCIAS

ANDRADE, Igor A. F. de; SILVA, Marcus V. A.; ARAUJO, Antonio C. L. de. **Programação orientada a aspectos: Um estudo sobre sua utilização na indústria de software**. Disponível em <http://www.cesmac.com.br/erbase2010/papers/wticg/65808.pdf> Acessado em 28 set. 2012.

APACHE Logging Services. Disponível em <http://logging.apache.org/log4j/1.2/manual.html>. Acessado em 14 out. 2012.

ASPECT Oriented Programming with Spring. Disponível em <http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/aop.html>, Acessado em 20 out. 2012.

ASPECTJ Development Tools (AJDT). Disponível em <http://www.eclipse.org/ajdt/>. Acessado em 15 out. 2012.

BRANQUINHO, A. A. B. **Projeto Northwind**. Publicado em 14 mar. 2012. Disponível em <http://wpattern.com/blog/search.aspx?q=projeto%20northwind&page=1>. Acessado em 01 out. 2012.

COUTINHO, Paulo C. **Hibernate e AspectJ – Gerenciamento de transação de forma 100% transparente**. Revista Java Magazine. v.60. Disponível em < http://www.devmedia.com.br/artigo-java-magazine-60-hibernate-e-aspectj/10215>. Acessado em 22 dez. 2012.

DOZER, Java Bean mapper. Disponível em <http://dozer.sourceforge.net/>. Acessado em 15 out. 2012.

ECLIPE IDE for Java EE Developers. . Disponível em <http://www.eclipse.org/downloads/>. Acessado em 15 out. 2012.

GONÇALVES, Adão F. C. *et al*. **PROJETO DE DESENVOLVIMENTO DO SIAGE-CFC (Sistema de Apoio a Gestão de Centro de Formação de Condutores)**. 2009. 192f. Trabalho de Conclusão de Curso – Faculdade Projeção, Brasília – Distrito Federal.

HORSTMANN, Cay S.; CORNELL, Gary. **Java CORE, Volume I - Fundamentos**. 8 ed. São Paulo: Pearson, 2010.

LADDAD, Ramnivas. **AspectJ IN ACTION – Pratical AspectJ-Oriented Programming**. Greenwich: Manning, 2003.

LOG4J Library Logging for Java. Disponível em <http://logging.apache.org/log4j/1.2/index.html>. Acessado em 15 out. 2012

MAVEN Project. Disponível em <http://maven.apache.org>. Acessado em 15 out. 2012.

NELSON, Torsten. **Programação Orientada a Aspectos com AspectJ**. Disponível em <http://www.aspectos.org/courses/aulasaop/curso_poa.html> acessado em 25 de jul. 2012.

RESENDE, Antonio Maria Pereira de; SILVA, Claudiney Calixto da. **Programação Orientada a Aspectos em Java.** Rio de Janeiro: Brasport, 2005.

SILVA, Lirene Fernandes da; LEITE, Julio Cesar Sampaio do Prado. **Uma Linguagem de Modelagem de Requisitos Orientada a Aspectos**. In: Workshop de Engenharia de Requisitos (WER 2005), 13-14 Jun. 2005, Porto, Portugal.

SILVA, Lirene Fernandes da; LEITE, Julio Cesar Sampaio do Prado. **Integração de Características Transversais Durante a Modelagem de Requisitos**. In: Simpósio Brasileiro de Engenharia de Software (SBES-2005), 3-7 Out. 2005, Uberlândia-MG, Brasil.

SPRING Framework. Disponível em <http://www.springsource.org/>. Acessado em 15 out. 2012.

WINCK, D. V. ; GOETTEN, Vicente. **AspectJ em 20 minutos - Java Free.org**, [s.d.], 4, disponível em <http://javafree.uol.com.br/artigo/871488/AspectJ-em-20-minutos.html> Acessado em 03 jun 2012.

XEROX CORPORATION, **The AspectJ™ Programming Guide**. Palo Alto Research Center, Incorporated. 2002-2003. Disponível em <http://www.eclipse.org/aspectj/doc/next/progguide/index.html>. Acessado em 15 out. 2012.

APÊNDICE A – Requisitos do Sistema para Centros de Formação de Condutores

A seguir os Requisitos Funcionais (RF) e Não Funcionais (RNF). Na figura 9 é apresentado o diagrama de casos de uso do sistema, que relaciona os requisitos funcionais do sistema.

RNF 1 – Requisitos de segurança

Requisitos relacionados com a segurança do sistema.

- RNF 1.1 – Auditoria
- RNF 1.2 – Confidencialidade
    - Criptografia
    - Autenticação
- RNF 1.3 – Integridade
- RNF 1.4 – Disponibilidade

RNF 2 – Requisitos de Comunicação

Requisitos relacionados com a comunicação entre os sistemas e a integração destes.

- RNF 2.1 – Mesmo que a base principal não esteja disponível deve-se possibilitar a operação do sistema.

RNF 3 – Persistência

Requisitos relacionados com a persistência dos dados seja em bancos de dados (DB) ou em arquivos.

- RNF 3.1 – Persistência em DB
- RNF 3.1 – Persistência em arquivo

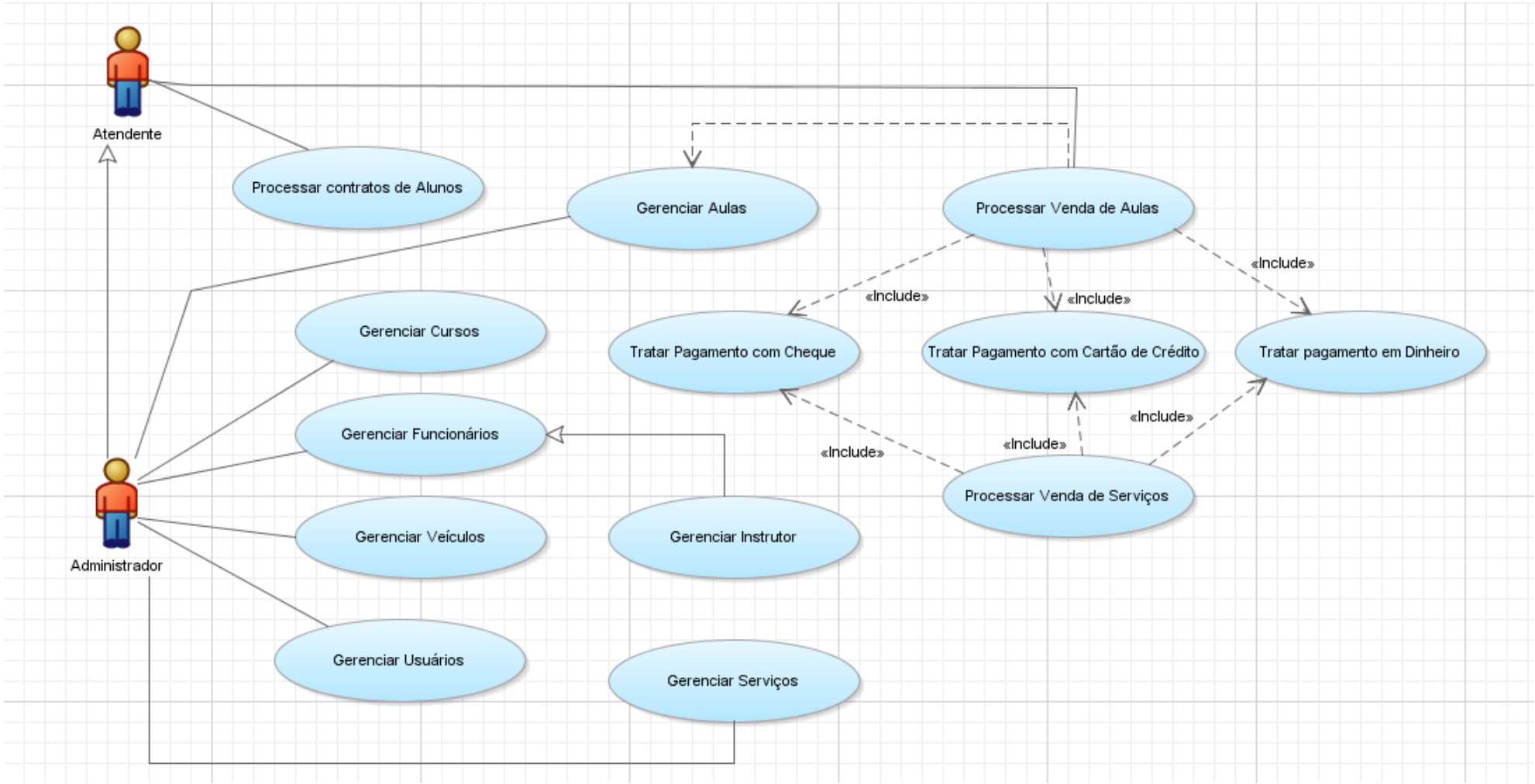

Figura 24: Estudo de Caso – Diagrama de Caso de Uso
Fonte: Criado pelo autor

A seguir os requisitos relacionados às funcionalidades do Sistema.

RF 1 Gerenciar Funcionários

- RF 1.1 – Cadastrar Funcionários;
- RF 1.2 – Consultar Funcionários;
- RF 1.3 – Alterar Funcionários;
- RF 1.4 – Excluir Funcionários;
- RF 1.5 – Gerar Relatório com Aulas agendadas do Funcionário Instrutor;
- RF 1.6 – Associar Instrutor a um ou mais Veículos; e
- RF 1.7 – Vincular funcionário a uma das filiais se for o caso.

RF 2 Gerenciar Veículos

- RF 2.1 – Cadastrar Veículos;
- RF 2.2 – Alterar Veículos;
- RF 2.3 – Excluir Veículos;
- RF 2.4 – Consultar Veículos;
- RF 2.6 – Associar veículos a uma categoria de CNH;

RF 3 Gerenciar Aulas

- RF 3.1 – Cadastrar Aulas;
- RF 3.2 – Alterar Aulas;
- RF 3.3 – Excluir Aulas;
- RF 3.4 – Consultar Aulas;
- RF 3.5 – Associar aula a um curso;
- RF 3.6 – Exibir Aulas em grade de calendário e horário, informando os horários disponíveis.

RF 4 Gerenciar Serviços

- RF 4.1 – Cadastrar Serviços;
- RF 4.2 – Alterar Serviços;
- RF 4.3 – Excluir Serviços;
- RF 4.4 – Consultar Serviços;
- RF 4.5 – Associar Serviço a um Curso

RF 5 Gerenciar Cursos

- RF 5.1 – Cadastrar Cursos;
- RF 5.2 – Alterar Cursos;
- RF 5.3 – Excluir Cursos;
- RF 5.4 – Consultar Cursos;
- RF 5.4 – Associar curso a um tipo de CFC (A ou B);

RF 6 Gerenciar Usuários

- RF 6.1 – Cadastrar Usuários;
- RF 6.2 – Alterar Usuários;
- RF 6.3 – Excluir Usuários;
- RF 6.4 – Consultar Usuários;

RF 7 Gerenciar Alunos

- RF 7.1 – Cadastrar Alunos;
- RF 7.2 – Alterar Alunos;
- RF 7.3 – Excluir Alunos;
- RF 7.4 – Consultar Alunos;
- RF 7.5 – Gerar Relatório com Aulas agendadas Aluno;
- RF 7.5 – Gerar Relatório de freqüência do Aluno nas aulas;
- RF 7.6 – Marcação de presença na aula através de leitor biométrico;
- RF 7.7 – Gerar Contrato com dados do processo do aluno.

RF 8 Processar Venda de Aulas e Serviços

- RF 8.1 – Cadastrar Venda;
- RF 8.2 – Alterar Vendas (restrito ao administrador);
- RF 8.3 – Excluir Vendas (restrito ao administrador);
- RF 8.4 – Consultar Vendas;
- RF 8.4 – Associar Aluno, Aulas e/ou Serviços a vendas;
- RF 8.4 – Consultar Vendas;

RF 8 Processar Pagamentos

- RF 8.1 – Incluir Pagamentos referentes a uma venda;
- RF 8.2 – Alterar Pagamentos (restrito ao administrador);
- RF 8.3 – Excluir Pagamentos (restrito ao administrador);
- RF 8.4 – Consultar Pagamentos;

A tabela 1 apresenta uma relação de regras de negócio identificadas em análises de processos de empresas do ramo de Auto-Escolas.

Nº	Descrição
RN01	Carga horária do curso de 20 horas/aula, sendo 50 minutos por aula.
RN02	No máximo 3 horas/aula por dia
RN03	A matrícula do aluno só é concluída após a aprovação no exame médico e no exame teórico, no CFC "A".
RN04	Bloqueio de aluno com pagamento em atraso.
RN05	Apenas alunos adimplentes são listados no "mapa de aula por instrutor"
RN06	Se a forma de pagamento for "promissória", o aluno deve ter quitado o valor total para realizar exame.
RN07	"Cancelamento de aula agendada" pode ser realizado somente por um usuário administrador.
RN08	O "cancelamento de aula agendada" pode ser realizado somente com 24 (vinte e quatro) horas de antecedência.
RN09	O usuário com perfil de "atendente" tem todos os privilégios do perfil "aluno", além de permissões específicas.
RN10	O usuário com perfil de "administrador" tem todos os privilégios do perfil "atendente", além de permissões específicas.
RN11	Validação de CPF e CNPJ conforme algoritmo descrito no ANEXO D.
RN12	Numero da matrícula: será composto por sete dígitos: os cinco primeiros serão os dígitos da identificação (id) do aluno no banco de dados (chave primária) acrescidos de zeros (0) à esquerda, até completar os cinco caracteres; os dois últimos correspondem ao ano corrente (ex.: 09 para 2009).
RN13	Validação de datas: Uma data é válida se o mês estiver entre 1 e 12, o dia estiver entre 1 e 31 e o ano estiver entre 1900 e 2099.
RN14	Campo obrigatório: Campos que devem ser informados.
RN15	Validações e restrições de caracteres.
RN16	Validação da placa do veículo: será composto por sete caracteres: os cinco primeiros serão letras do alfabeto (A-Z), os quatro últimos serão dígitos numéricos.
RN17	Validação da Carteira Nacional de Habilitação – CNH. ANEXO B
RN18	O numero do RENACH, terá obrigatoriamente 11 caracteres conforme determina o Conselho Nacional de Transito – CONTRAN (Resolução da expedição da CNH, Anexo B deste trabalho)

Tabela 01: Lista de regras de negócio

APÊNDICE B – Modelos do Sistema para Centros de Formação de Condutores

Para auxiliar em seu desenvolvimento são utilizadas ferramentas como: Apache Maven[8], utilizado para gerenciar os vários módulos do sistema e suas dependências. Cada módulo é representado por um subprojeto vinculado a um projeto principal chamado, neste caso, o *autoescola-all*. Como IDE é utilizado o Eclipse Indigo9 com os plugins:

- M2E (Maven 2 Eclipse): Plugin do Maven para o Eclipse;
- AspectJ Development Tools[10]: ferramenta para criação de aspectos em Java;
- Hibernate[11] 4.0.1 para configuração do banco de dados e persistência dos dados;
- Spring[12]: utilizadas as funcionalidades de injeção de dependência, RMI e suporte a AOP para controle transacional;
- JUnit: para testes unitários;
- Log4J[13]: para o *logging* de informações do sistema; e
- Dozer: para o mapeamento e conversão de *Beans* em *Entities* e vice-versa.

O Diagrama de pacotes, na figura 11, é comporto pelos pacotes que formam a estrutura do sistema Auto Escola. As responsabilidades funcionais são separadas nos cinco pacotes principais e em seus sub-pacotes. Os principais são: *autoescola-business, autoescola-database, autoescola-factory, autoescola-service e autoescola-utils.*

[8] Apache Maven, um software de gestão de software e suas dependências. Fonte: http://maven.apache.org.
[9] Eclipse IDE for Java EE Developers. Fonte: http://www.eclipse.org/downloads/.
[10] Eplipse AspectJ Development Tools (AJDT). Fonte: http://www.eclipse.org/ajdt/.
[11] Hibernate Framework de persistência. Fonte: http://www.hibernate.org.
[12] Spring Framework. Fonte: http://www.springsource.org/.
[13] Log4J, biblioteca de logging em Java. Fonte: http://logging.apache.org/log4j/1.2/index.html.

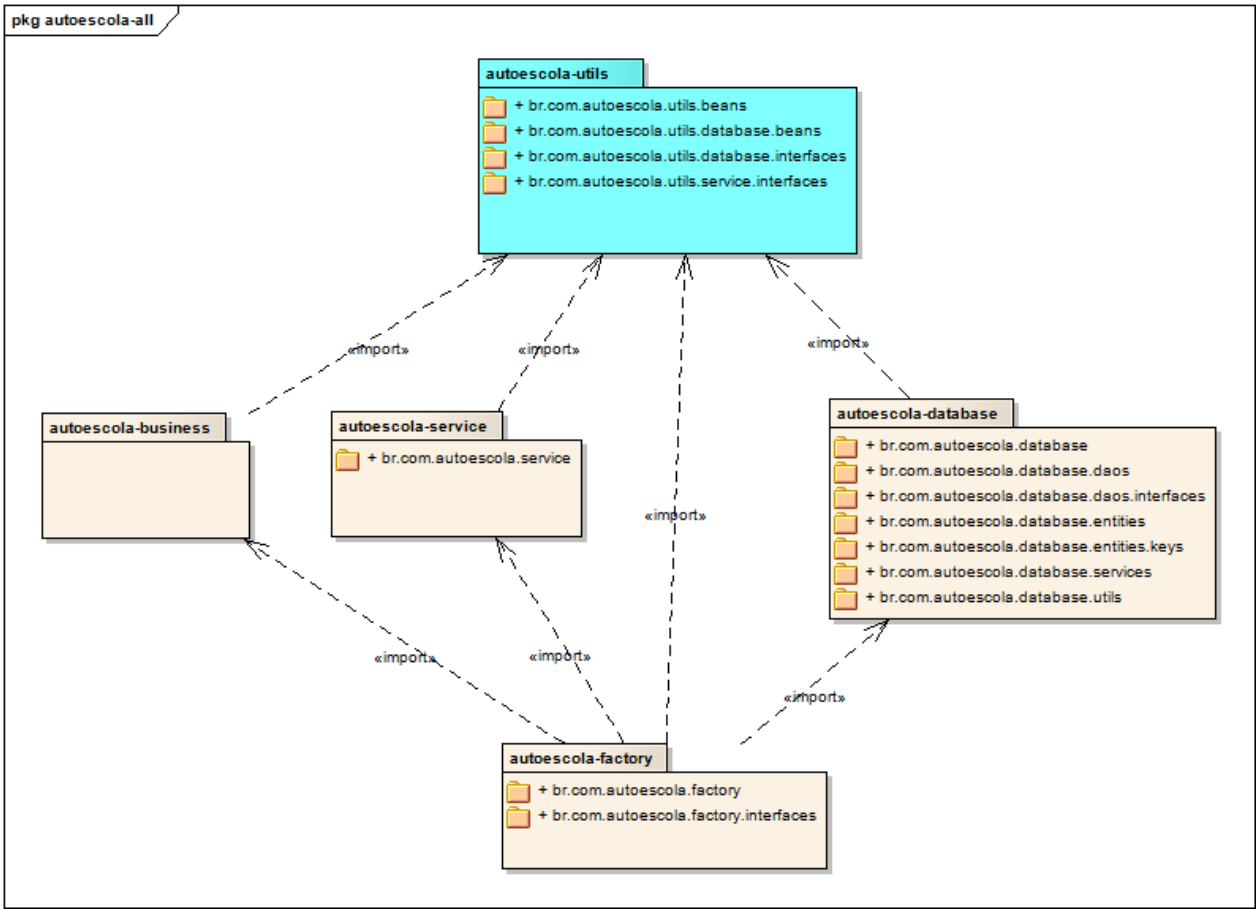

Figura 25: Diagrama de Pacotes do Projeto Auto Escola
Fonte: Criado pelo autor

É estabelecido o mínino de dependência entre os módulos a fim de manter o encapsulamento e separação dos "interesses". O controle de dependência entre os módulos e a disponibilização dos serviços é realizado no pacote *autoescola-factory*. No pacote *autoescola-business* serão desenvolvidas classes que implementam as regras de negócio do sistema. No módulo *autoescola-service* serão criados serviços remotos utilizando RMI (*Remote Method Invocation*, em português: Invocação de Métodos Remotos) para controlar o acesso aos dados pelas aplicações ou módulos clientes RMI. O módulo *autoescola-database* é composto por classes responsáveis pelo acesso e manipulação dos dados no banco de dados. E classes utilitárias – que não implementam comportamento – compõem o pacote *autoescola-utils*, que está acessível por todos os outros módulos (BRANQUINHO, 2012). Os módulos *autoescola-database, autoescola-business e autoescola-service* são os principais do sistema. Estes têm um baixo acoplamento e suas classes são orientadas a *interface*, o que facilita na manutenção e torna o sistema extensível[14].

[14] Sistema extensível: refere-se à facilidade em incrementá-lo com novos módulos e/ou integrá-lo a outros sistemas.

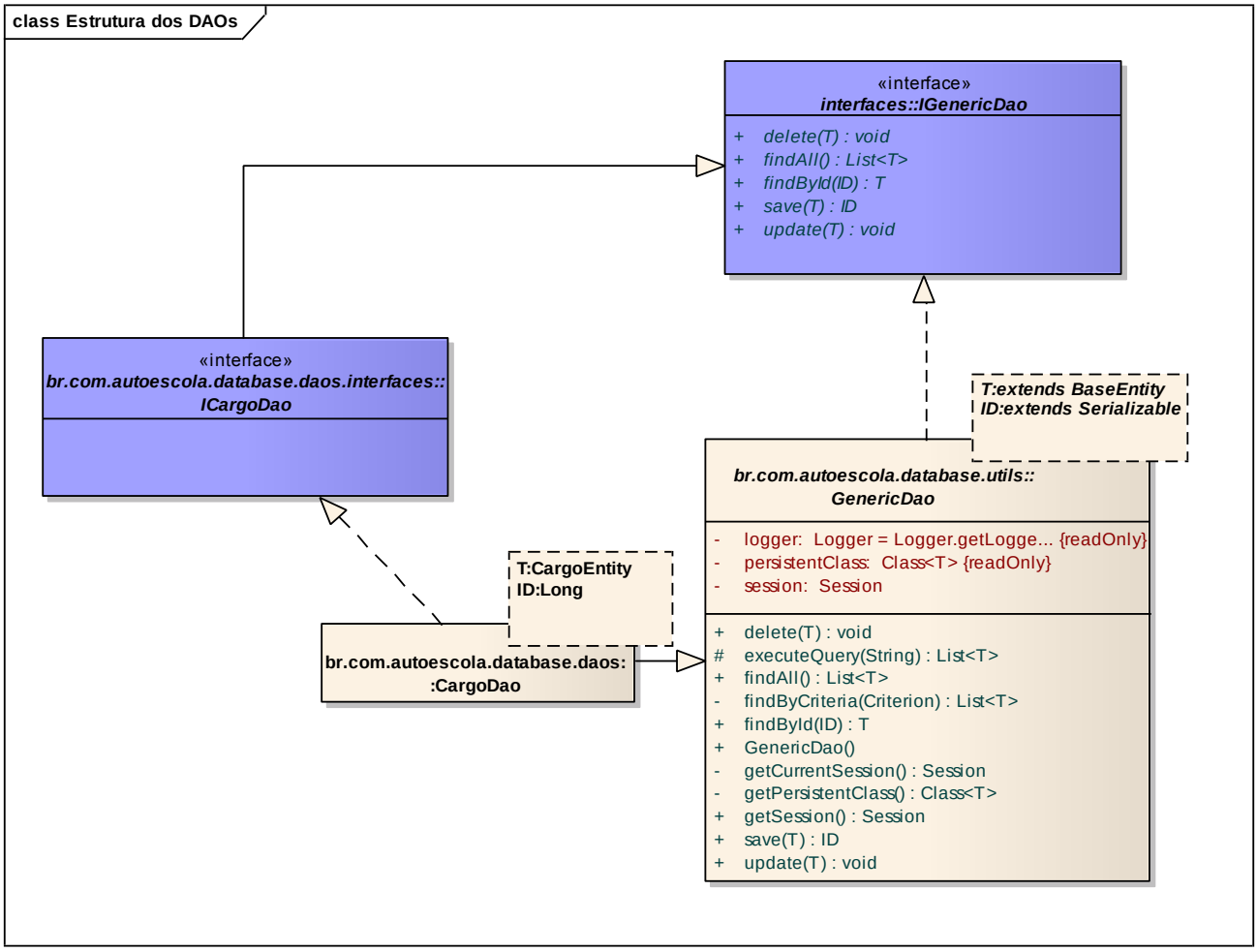

Figura 26: Exemplo de Modelo de DAOs do Projeto Auto Escola.
Fonte: Criado pelo autor

No Diagrama de Modelos de DAOs (figura 11) é demonstrada a estrutura de DAOs do projeto. Neste diagrama é modelado apenas o DAO *CargoDao*[15] – a título de exemplo. Cada DAO terá sua correspondente *interface* e herda a *GenericDao*, que implementas métodos comuns a todos (BRANQUINHO, 2012). A relação completa de DAOs, alguns serviços (como na classe *CargoService*) e suas *interfaces*, e classes utilitárias específicas do módulo Database constam na Figura 12.

[15] Os demais exemplos de códigos e diagramas utilizados nesse trabalho terão como base o objetos relacionados com a entidade CargoEntity, a exemplo da CargoDao.

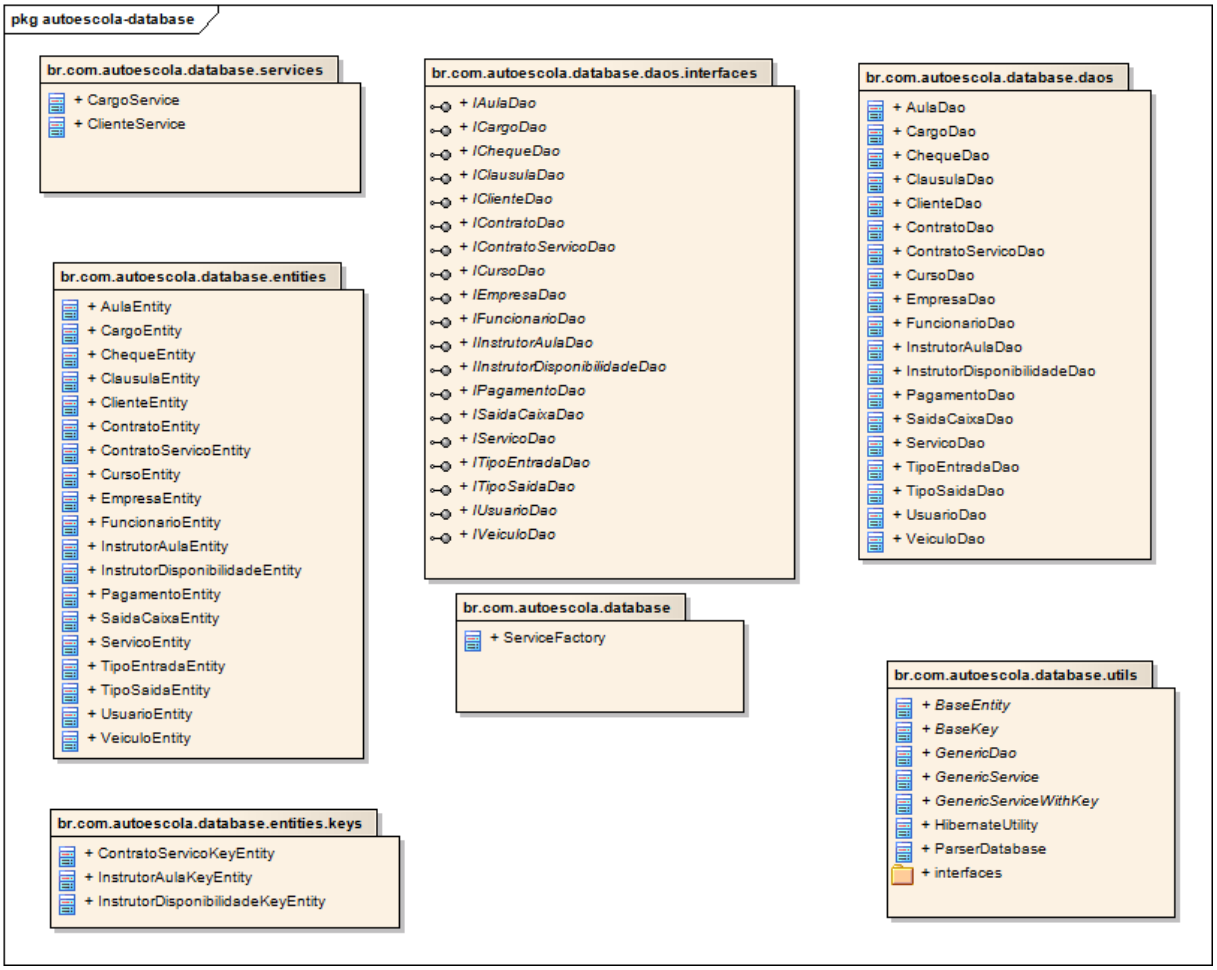

Figura 27: Pacote do módulo Database
Fonte: criado pelo autor

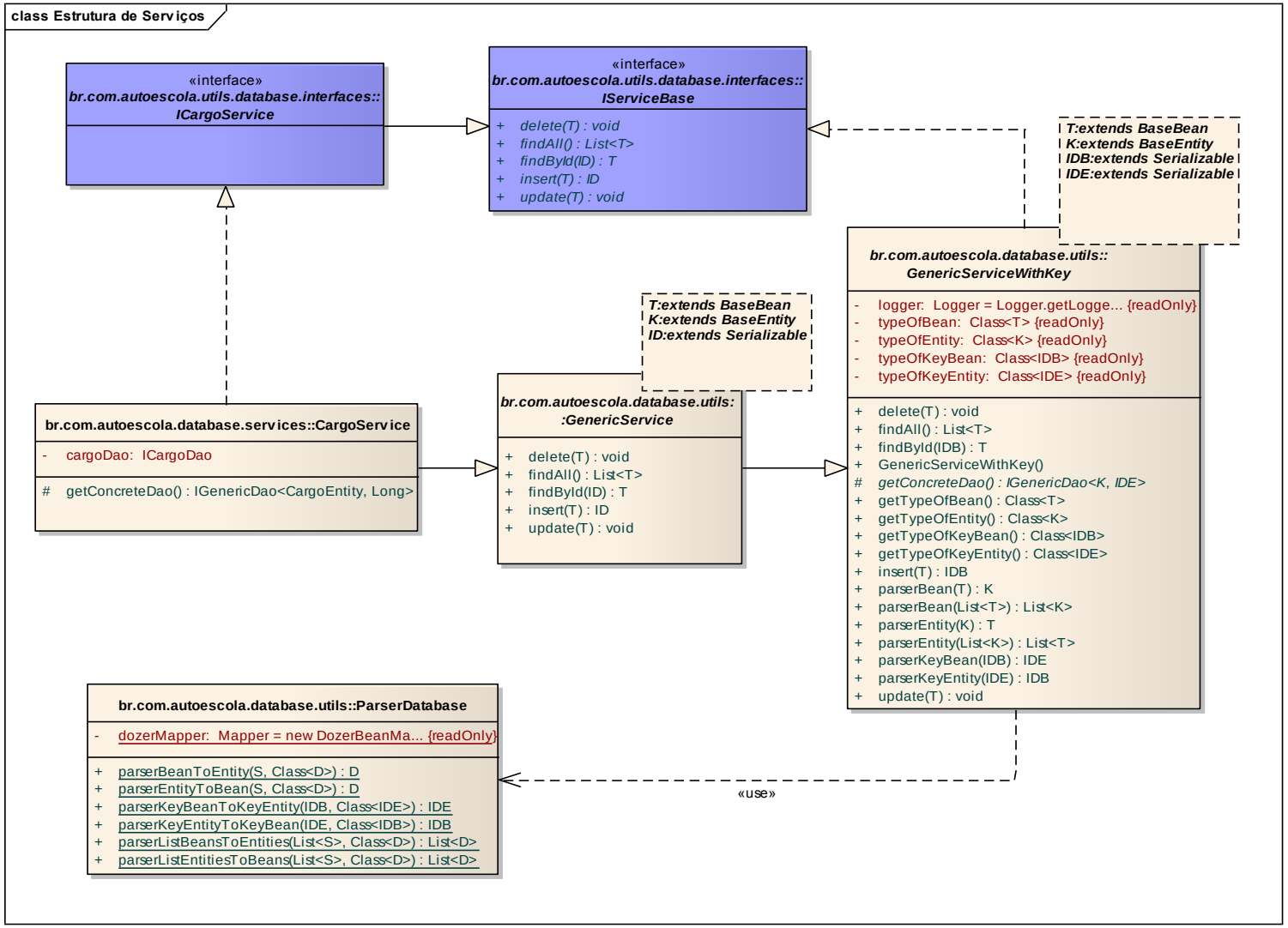

Figura 28: Diagrama do modelo de Serviços do Projeto Auto Escola
Fonte: Criado pelo autor

No modelo de serviços (Figura 13), as classes *services* herdam da *GenericService* os métodos comuns (*delete, findAll, findById, insert, update*) e implementas seus específicos. A classe *ParserDatabase* auxilia na conversão da *Entity* em *Bean* e vice-versa, através da biblioteca *Dozer*[16], e são usadas pela classes genéricas *GenericServiveWithKey* e *GenericService*.

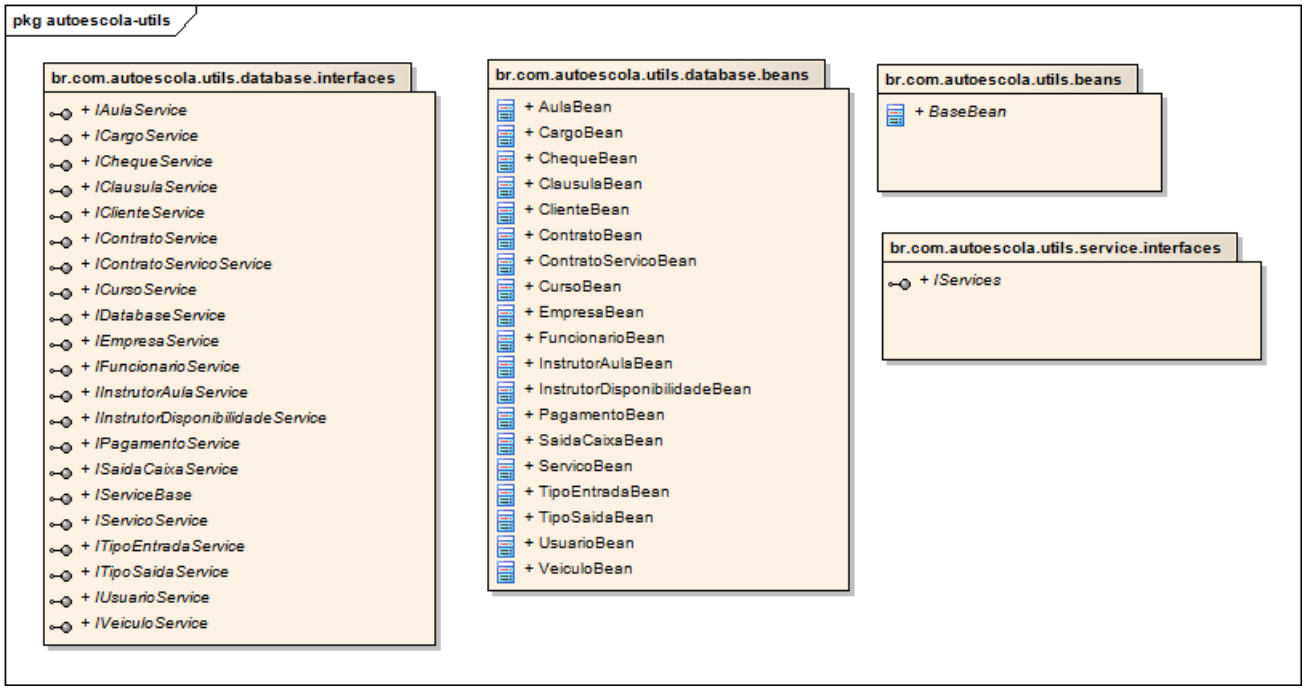

Figura 29: Pacote do módulo Utils
Fonte: criado pelo autor

O módulo utilitário (*Utils*) é composto pelos *Beans* e as *interfaces* para os *Services* correspondentes. Essas interfaces de serviços serão disponibilizados aos outros módulos - e potencialmente, a outros sistemas - através da interface *IServices* (Figura 14).

Nesta seção foi, sumariamente, descrito o projeto "alvo" do estudo de caso para este trabalho. Mas ainda não aplicando os conceitos da AOP ao projeto. Isto será feito na seção seguinte.

[16] Dozer, "mapeador" Java Bean. Fonte: http://dozer.sourceforge.net/