FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

**DANIEL BRUNO FERNANDES CONRADO**

# IMPLEMENTAÇÃO DA PERSISTÊNCIA UTILIZANDO JPA E ASPECTJ

MARÍLIA
2009

DANIEL BRUNO FERNANDES CONRADO

# IMPLEMENTAÇÃO DA PERSISTÊNCIA UTILIZANDO JPA E ASPECTJ

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Ms. PAULO AUGUSTO NARDI

MARÍLIA
2009

Conrado, Daniel Bruno Fernandes
Implementação da persistência utilizando JPA e AspectJ / Daniel Bruno Fernandes Conrado; orientador: Paulo Augusto Nardi. Marília, SP: [s.n.], 2009.
71 f.

Trabalho de Curso (Graduação em Ciência da Computação) – Curso de Ciência da Computação, Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, Marília, 2009.

1. Programação Orientada a Objetos 2. Programação Orientada a Aspectos 3. Persistência 4. ORM 5. Java Persistence API 6. AspectJ

CDD: 005.1

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
*BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO*

**TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL**

Daniel Bruno Fernandes Conrado

IMPLEMENTAÇÃO DA PERSISTÊNCIA UTILIZANDO JPA E ASPECTJ

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 9,0 ( nove )

Orientador: Paulo Augusto Nardi

1°. Examinador: Fabio Lucio Meira

2°. Examinador: Leonardo Castro Botega

Marília, 02 de dezembro de 2009.

*Aos meus irmãos, Juninho e Bruna.*

# AGRADECIMENTOS

Agradeço a Deus por dar sentido a minha vida, por todos os favores imerecidos que tenho recebido e pelas pessoas que fizeram e fazem parte da minha história.

À minha querida esposa, Yumi (ゆみ), pelo amor, carinho, compreensão, apoio, e por simplesmente ser. Você é tudo na minha vida. あいしてる.

À minha mãe, Dinha, por ter me ensinado a arte de aprender a viver e a cantar. Ao meu padrasto, Ronaldo, e ao meu pai, Dalton.

Aos meus sogros, Gilson e Esther, por todos os conselhos, todo o apoio e carinho que me deram.

Aos meus amigos Tsen e Steve, por terem acreditado em mim, me dado forças para continuar e motivos para não desistir.

Ao meu orientador, Paulo Augusto Nardi, pelo auxílio, amizade e por todo conhecimento passado.

Ao professor Elvis Fusco pelo apoio e amizade.

Ao meu amigo Abraão, pelas loucuras e ideias malucas.

Ao meu amigo Bicudo, pelos confrontos ideológicos, por ter sido meu padrinho de casamento e por ter chegado mais de doze horas atrasado na cerimônia.

Ao pastor Gilberto Stéfano pelo exemplo de vida. À Igreja Batista da Fé, pelo companheirismo.

À turma do LAS (Laboratório de Arquitetura de Sistemas), Antonio, Bruno, Denison e Ernesto por tornar agradáveis estes últimos dias de trabalho.

A todos os alunos da minha classe.

Aos demais professores, pelo conhecimento passado.

Ao pessoal da copa da UNIVEM, pelo café com simpatia.

*"Não acumuleis para vós outros tesouros sobre a terra, onde a traça e a ferrugem corroem e onde ladrões escavam e roubam; mas ajuntai para vós outros tesouros no céu, onde traça nem ferrugem corrói, e onde ladrões não escavam, nem roubam; porque, onde está o teu tesouro, aí estará também o teu coração."*

**Mateus cap. 6, v. 19-21.**

CONRADO, Daniel Bruno Fernandes. **Implementação da persistência utilizando JPA e ASPECTJ.** 2009. 71 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2009.

## RESUMO

Grande parte das aplicações comerciais atuais são desenvolvidas utilizando a programação orientada a objetos e realizam a persistência em bancos de dados relacionais. Porém, existem características do modelo relacional e do modelo orientado a objeto que levam a uma incompatibilidade entre estes dois modelos. A técnica de mapeamento objeto-relacional foi desenvolvida para reduzir tal incompatibilidade, contribuindo para a comunicação entre objetos e tabelas. Uma implementação desta técnica, utilizada na indústria de software, é o framework Java Persistence API. Com a utilização deste framework, porém, há um entrelaçamento entre o código funcional e o de persistência e ainda o espalhamento deste último por toda a aplicação. Isto acontece porque há uma carência por parte da programação orientada a objetos de abstrações adequadas para a modularização de códigos relativos a utilização do framework. Devido a esta carência, surgiu a programação orientada a aspectos, a qual provê novas abstrações capazes de modularizar tais códigos, e o framework AspectJ, que estende a linguagem Java com tais abstrações. Este trabalho tem como objetivo propôr uma implementação orientada a aspectos do uso do framework JPA em uma aplicação, de tal forma que não haja entrelaçamento e espalhamento do código relativo a este framework.

**Palavras-chave**: Programação Orientada a Objetos. Programação Orientada a Aspectos. Persistência. ORM. Java Persistence API. AspectJ.

CONRADO, Daniel Bruno Fernandes. **Proposta de implementação da persistência utilizando JPA e ASPECTJ.** 2009. 71 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino "Eurípides Soares da Rocha", Marília, 2009.

## ABSTRACT

Most current commercial applications have been developed using object-oriented programming and perform persistence in relational databases. However, there are features of the relational model and object-oriented model leading to a impedance mismatch between them. The object-relational mapping technique was developed to reduce this impedance mismatch, contributing to the communication between objects and tables. A widely used implementation of this technique in the software industry is the Java Persistence API. By using this framework, however, there is an interrelationship between the functional code and the persistence code, and further the spread of the latter throughout the application. This is due to the lack of abstractions suitable for modularization of this code by object-oriented programming. Because of this lack that the aspect-oriented programming emerged, which provides new abstractions capable of modularize such codes, and the AspectJ framework, which extends the Java language with such abstractions. This monograph aims to propose an aspect-oriented implementation of the use of the JPA framework in an application, so there is no interlacing and spreading code for this framework.

**Keywords:** Object-Oriented Programming. Aspect-Oriented Programming. Persistence. ORM. Java Persistence API. AspectJ.

# LISTA DE FIGURAS

# SUMÁRIO

# INTRODUÇÃO

A abordagem orientada a objetos para o desenvolvimento de *software* é atualmente de uso comum, particularmente para o desenvolvimento de sistemas interativos (SOMMERVILLE, 2007). A Programação Orientada a Objetos (POO) surgiu para facilitar o desenvolvimento de *softwares* complexos através de uma forma natural de expressar entidades pertencentes a um determinado domínio.

Apesar da POO ser amplamente aplicada no desenvolvimento de processos, bancos de dados relacionais são os mais utilizados para o armazenamento de informações, resultando em conflito de paradigmas, pois o modelo relacional é baseado na teoria dos conjuntos e considera apenas dados, enquanto o modelo orientado a objetos relaciona dados com comportamentos (ELMASRI; NAVATHE, 2005).

Para solucionar este conflito, foi desenvolvida a técnica de mapeamento objeto-relacional ou *object-relational mapping* (ORM), que consiste no desenvolvimento de um adaptador entre os dois modelos (AMBLER, 2009). Este adaptador é responsável por realizar a persistência dos objetos em bancos de dados relacionais. Entretanto, muito esforço por parte dos desenvolvedores é empregado para a construção deste adaptador. Por isso, padrões de projeto de ORM foram desenvolvidos, como *Persistence Layer* (YODER, 1998), *Active Record* (FOWLER, 2003) e *Data Access Object* (ALUR et al., 2003).

A partir de então, vários *frameworks* foram desenvolvidos baseados nestes padrões para acelerar o desenvolvimento de ORM, como Hibernate (HIBERNATE, 2009), SubSonic (SUBSONIC, 2009) e *Java Persistence API* (JPA) (THE JAVA..., 2009). Este último é o padrão adotado pela plataforma Java (DEITEL, 2003) para a persistência de objetos em bancos de dados relacionais e tem sido amplamente utilizado pelos programadores que utilizam a linguagem Java.

A utilização deste *framework*, porém, resulta em entrelaçamento de código funcional com o código relativo à persistência. Além disso, a utilização de recursos do *framework* fica espalhada por toda a aplicação. Isto ocorre porque a POO não fornece abstrações adequadas para a modularização deste tipo de código, que é considerado um interesse transversal da aplicação (CAMARGO, 2006).

A Programação Orientada a Aspectos (POA) surgiu com o objetivo de modularizar interesses transversais, como persistência, segurança, concorrência, entre outros. A linguagem

AspectJ (KICZALES et al, 2001) fornece uma extensão orientada a aspectos para a linguagem Java e possui construções capazes de implementar tal modularização.

Vários trabalhos foram realizados com o objetivo de implementar o interesse de persistência utilizando a POA, dentre eles estão (CAMARGO et al., 2003; RASHID; CHITCHYAN, 2003; SOARES et al., 2002). Entretanto, tais trabalhos não se beneficiaram dos *frameworks* supracitados. A pesquisa apresentada neste trabalho visa contribuir para a modularização da persistência utilizando *frameworks*.

Este trabalho tem como objetivo verificar se é possível implementar o interesse de persistência utilizando JPA como *framework* de ORM e AspectJ para modularizar sua utilização. Para isso, desenvolve-se uma aplicação orientada a objetos com dois cadastros simples e implementa-se um módulo orientado a aspectos do interesse de persistência.

O capítulo 1 aborda aspectos da programação orientada a objetos, no capítulo 2 encontram-se algumas formas de persistência de objetos e o capítulo 3 aborda o *framework* JPA. No capítulo 4 encontram-se um estudo sobre AspectJ e alguns trabalhos relacionados. O capítulo 5 apresenta a proposta de implementação utilizando tais tecnologias e o capítulo 6 apresenta o estudo de caso realizado. A conclusão encontra-se no capítulo 7.

# CAPÍTULO 1 – PROGRAMAÇÃO ORIENTADA A OBJETOS

A Programação Orientada a Objetos (POO) organiza um sistema em uma coleção de objetos separados que incorporam tanto a estrutura quanto o comportamento dos dados. Isto contrasta com técnicas de programação anteriores, como o paradigma estruturado (SCHACH, 2006) onde dados e comportamento estão pouco conectados (RUMBAUGH; BLAHA, 2006). Algumas linguagens de programação orientadas a objetos são: Java (DEITEL, 2003), C++ (STROUSTRUP, 1997), Ruby (FLANAGAN; MATSUMOTO, 2008) e Python (FEHILY, 2002).

Os objetos de um sistema podem ser concebidos através da análise de requisitos e utilizados para representar os dados do sistema e seu processamento. Para algumas classes de sistemas, o desenvolvimento orientado a objetos é um meio natural de expressar as entidades do mundo real que são manipuladas pelo sistema, embora existam entidades mais abstratas que são mais difíceis de modelar (SOMMERVILLE, 2007). Por exemplo, há sistemas que lidam com entidades concretas, como carros, livros e instrumentos musicais, as quais possuem atributos e comportamentos claramente identificáveis, e existem sistemas que manipulam entidades conceituais, como políticas de escalonamento de processos e processadores de texto, que não possuem necessariamente uma interface simples, ou seja, é difícil identificar suas propriedades.

## 1.1 Características da POO

Segundo Rumbaugh e Blaha (2006), há discussões sobre as características exatas da POO, entretanto elas incluem quatro aspectos: identidade, classificação, herança e polimorfismo.

**Identidade** define dados quantizados em entidades distintas e distinguíveis como objetos. Dois objetos são distintos mesmo que os valores de seus atributos sejam iguais. Em linguagens de programação orientadas a objetos, é comum a identificação ser feita através de endereços de memória. De forma análoga, gêmeos univitelinos, apesar de serem idênticos, são duas pessoas distintas.

**Classificação** é uma abstração em relação a um conjunto de objetos com atributos e operações comuns. Segundo Sommerville (2007), os objetos são instâncias da classe de

objeto, e muitos objetos diferentes podem ser criados a partir de uma classe. Por exemplo, os objetos João e Maria pertencem à classe Pessoa e foram construídos a partir dela.

Uma classe define quais atributos e operações os objetos criados a partir dela terão. Por exemplo, a classe Pessoa define um atributo *nome* e uma operação *andar*, logo todos os objetos criados a partir da classe Pessoa terão o atributo *nome* e a operação *andar*, e o algoritmo desta operação é o mesmo para todos.

Cada objeto possui uma referência implícita à sua classe. Assim, todo objeto tem conhecimento de qual classe pertence.

**Herança** define um relacionamento hierárquico entre classes. Quando uma classe *B* herda (ou estende) uma classe *A*, *B* é conhecida como *subclasse* de *A* e, consequentemente, *A* é conhecida como *superclasse* de *B*. É um relacionamento conhecido como "é uma", ou seja, toda instância de *B* é também uma instância de *A* e tudo o que *A* possui, *B* também possui (JIA, 2000). Dadas as classes *Pessoa* e *Professor*, sendo a última subclasse da primeira, conclui-se que um objeto da classe *Professor* é também um objeto da classe *Pessoa*.

Segundo Rumbaugh e Blaha (2006, p. 2), "**Polimorfismo** significa que a mesma operação pode se comportar de forma diferente para diferentes classes". Em um jogo de xadrez, por exemplo, a operação *mover* tem comportamento diferente para cada peça. Esta operação, quando executada pela peça que representa o bispo, realiza movimentos apenas pelas diagonais do tabuleiro, enquanto a peça que representa o cavalo é movida em um padrão semelhante à letra "L". O algoritmo que invoca esta operação não precisa saber qual tipo de peça está manipulando para que a operação correta seja executada.

### 1.1.1 Classes

As classes definem uma estrutura específica de uma determinada abstração. Uma classe representa um conjunto de objetos e define seus atributos e operações (com suas respectivas implementações). Possui um nome único que transmite o seu significado dentro do domínio do sistema (FAKURA, 2000). Por exemplo, a classe Vendedor pode representar todas as pessoas que fazem parte do grupo de "vendedores de carros em uma concessionária".

Uma classe separa a interface da implementação. A interface representa a parte pública da classe e a implementação, a parte privada. Os longos anos de experiência da engenharia de software mostraram que é importante proteger os dados contra acessos

inesperados, indevidos e indesejados provenientes de outras partes do sistema. Esta separação faz com que um objeto forneça uma interface bem definida de comunicação com os demais objetos e os livra de saber detalhes de sua implementação. Desta forma, a estrutura interna do objeto pode ser alterada sem a necessidade de alterar todos os demais objetos que o manipulam. Esta técnica chama-se *encapsulamento*, uma característica das linguagens de programação orientadas a objetos (SCHACH, 2005). Por exemplo, um sistema bancário tem uma classe Conta que representa a estrutura de um conjunto de contas bancárias. As operações que podem ser feitas são *sacar*, *depositar* e *verificarSaldo,* e representam a parte pública da classe. Todos os demais componentes do sistema sabem que objetos da classe Conta possuem seus respectivos saldos, porém, desconhecem a forma como esta informação é implementada. Além disso, estes componentes não precisam ter conhecimento deste detalhe para manipular objetos da classe Conta. Qualquer alteração na forma como a classe Conta mantém seu saldo é invisível ao sistema, desde que o comportamento esperado de sua parte pública permaneça. A **Figura 1** mostra uma possível implementação desta classe.

```
public class Conta {

	private float saldo;

	public Conta() {
		saldo = 0;
	}

	public void sacar(float quantia) {
		saldo -= quantia;
	}

	public void depositar(float quantia) {
		saldo += quantia;
	}

	public float verificarSaldo() {
		return saldo;
	}
}
```

Figura 1 – Possível implementação em Java da classe Conta

A parte privada de uma classe é acessível apenas a partir da mesma; subclasses não possuem acesso a esta parte. Entretanto, a POO fornece uma maneira de compartilhar informações entre superclasses e subclasses, mantendo-as inacessíveis por outras classes, não pertencentes à hierarquia. A parte da classe que é compartilhada com suas subclasses é chamada de *protegida* (*protected*) (SEBESTA, 2009).

As classes podem ser modeladas através da UML, Linguagem Unificada de Modelagem (*Unified Modeling Language*) (BOOCH et al., 2007). Existem outras técnicas de modelagem, como a OMT (*Object Modeling Tecnique*) (RUMBAUGH et al., 1993), a OOSE (*Object-Oriented Software Engineering*) (JACOBSON, 1992) e o Método de Coad-Yourdon (*Coad-Yourdon* Method) (YOURDON; COAD, 1990). O símbolo da UML para uma classe é uma caixa dividida em três seções: nome da classe, atributos e operações. O nome da classe é centralizado e escrito em negrito. A notação de um atributo é seu nome em fonte normal e alinhado à esquerda, e pode conter detalhes adicionais, como tipo e valor padrão. Um sinal de dois pontos precede o tipo e um sinal de igual precede o valor padrão. As operações são listadas também em fonte normal e alinhadas à esquerda. A lista de argumentos e o tipo de resultado são opcionais e podem suceder as operações. Caso a lista de argumentos seja especificada, deve estar entre parênteses, e cada argumento separado por vírgula. O sinal de dois pontos precede os tipos dos argumentos e também o tipo de resultado da operação. A visibilidade dos membros da classe também podem ser modeladas. Um sinal de subtração precedendo o membro torna-o privado, um sinal de adição torna-o público e o sinal # torna-o protegido. (RUMBAUGH; BLAHA, 2006; BOOCH et al., 2007). A **Figura 2** mostra a classe Conta, apresentada anteriormente, modelada através da UML.

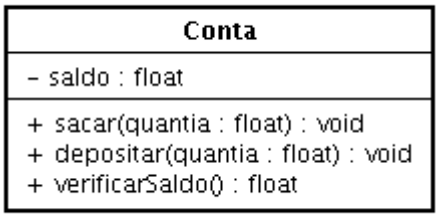


Figura 2 – Classe Conta modelada com a UML

### 1.1.2 Associações

Uma associação entre classes descreve um grupo de possíveis ligações (conexões), com estrutura e semântica comuns, entre objetos destas classes. As associações estão para as ligações assim como uma classe está para objetos. Portanto, uma ligação é uma instância de uma associação. Normalmente são descritas como verbos, dentro do domínio do sistema. Por exemplo, as classes Funcionário e Empresa, em um sistema de folha de pagamento, possui uma associação de nome *trabalhaPara*. Instâncias desta associação podem representar

situações como: João trabalha para a IBM, Maria trabalha para a Sun, e assim por diante. Uma associação é geralmente implementada através de referências de um objeto para outro, ou seja, um objeto possui um atributo que é um outro objeto. Por exemplo, a associação *trabalhaPara* pode ser implementada através da inclusão de um atributo de tipo da classe Empresa na classe Funcionário (RUMBAUGH; BLAHA, 2006). A **Figura 3** exibe uma implementação desta associação.

```
public class Funcionario {

	private String nome;

	/**
	* Implementação da associação trabalhaPara
	* entre as classes Empresa e Funcionario
	*
	*/
	private Empresa empresa;

	...
}
```

Figura 3 – Fragmento do código em Java da classe Funcionário que implementa a associação *trabalhaPara*.

Apesar da implementação de associações como referências ser perfeitamente aceitável, a modelagem não deve ser feita desta maneira (conforme parte (a) da **Figura 4**). Uma ligação entre dois objetos não faz parte de nenhum objeto em particular, mas depende de ambos.

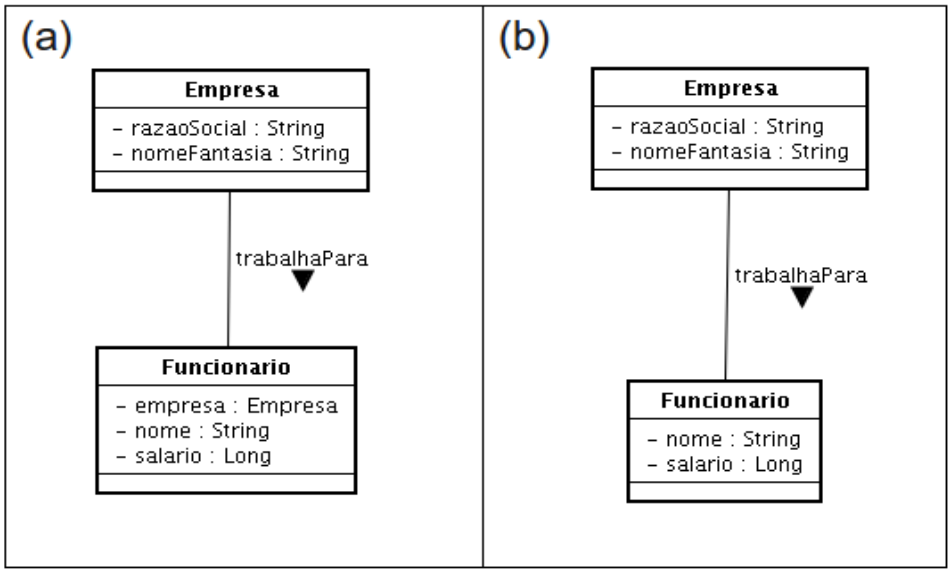


Figura 4 – Associação *trabalhaPara* modelada com referências e sem referências. A parte (b) é a forma correta

As associações transcendem as classes, de modo que não podem ser privadas a uma classe. Portanto, deve-se considerar as associações em igualdade com as classes. Embora a classe funcionário tenha um atributo de tipo Empresa, uma empresa não faz parte de um funcionário. A parte (b) da **Figura 4** mostra a forma correta de modelagem.

Uma associação tem pelo menos duas extremidades. Cada uma é representada por uma propriedade conectada ao tipo da extremidade. As propriedades podem ser nomeadas. Podem existir mais de uma extremidade da mesma classe. Se uma extremidade possuir uma propriedade que é do tipo de alguma classe pertencente à associação, então a associação é *navegável* a partir desta extremidade (OMG, 2009). Por exemplo, em um sistema de vendas tem-se as classes Venda e ItemVenda. Existe uma associação chamada *possui* entre estas classes representando a afirmação "uma venda possui itens de venda". A propriedade que representa a extremidade ItemVenda chama-se *item*. Esta associação foi implementada através de referências. A classe Venda possui um atributo *itens*, o qual é uma coleção de objetos da classe ItemVenda. Portanto, a associação é navegável a partir da classe Venda. A associação pode ser navegável a partir de ItemVenda somente se esta possuir um atributo do tipo da classe Venda.

O símbolo UML para associações binárias é uma linha sólida entre as classes. Uma associação entre três ou mais classes é representada por um losango que une as linhas que as conectam. O nome da associação pode ser especificado perto do símbolo de associação e não deve se aproximar demais das classes envolvidas a ponto de ser confundido com as propriedades. Em associações binárias, pode-se representar a direção de interpretação da associação através de uma ponta de flecha triangular e sólida na frente ou ao invés de seu nome. A **Figura 5** mostra um exemplo de uma associação binária.


Figura 5 – Associação *possui* entre as classes Venda e ItemVenda

Um associação pode representar uma agregação. Agregação é um tipo especial de associação que define um relacionamento parte-do-todo. Um computador, por exemplo, consiste em vários periféricos, como CPU, monitor, teclado e *mouse*. Afirma-se que estas partes *agregam* um computador. A notação da UML para uma agregação é um losango vazio

colocado na extremidade agregada. Apenas associações binárias podem ser agregações. A **Figura 6** exibe este exemplo.

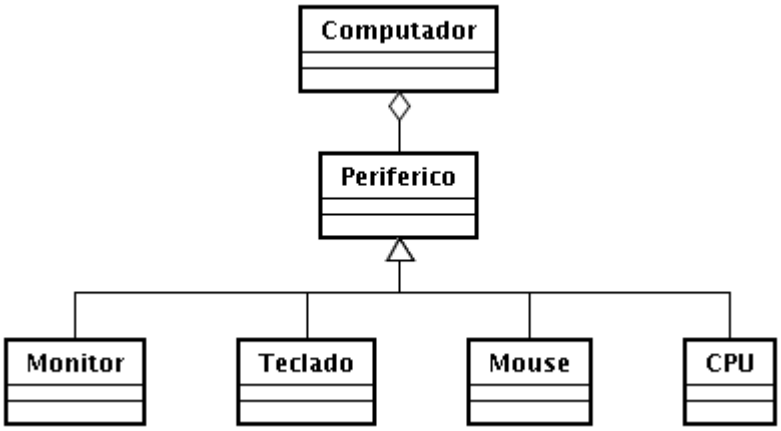


Figura 6 – Computador é agregado por monitor, teclado, mouse e CPU

Existe um tipo especial de agregação chamado *composição*. É considerada uma agregação forte pois as partes que compõem o objeto só existem se este existir. Difere da agregação normal no que tange às partes pois, no exemplo do computador, um monitor existe mesmo se o computador não existir. Quando um objeto composto é destruído, todas as suas partes também são. Por exemplo, um pedido de compra é composto por vários itens, cada qual associado a um produto. Se o pedido for cancelado (destruído), todos os seus itens também serão, pois um item de pedido não tem sentido semântico se não estiver ligado a um pedido. A notação da UML para composição é semelhante à da agregação, exceto pelo fato de que o losango é preenchido (OMG, 2009; SCHACH, 2005). A **Figura 7** exemplifica uma composição.

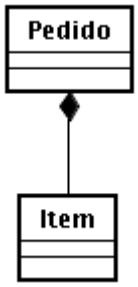


Figura 7 – A classe pedido é composta pela classe item

### 1.1.3 Multiplicidade

Multiplicidade é uma restrição aplicada à uma associação que define a quantidade de objetos de uma extremidade que podem estar conectados a um único objeto da outra extremidade. Ela pode ser informada em zero ou mais extremidades da associação. É normalmente descrita como sendo “um” ou “muitos”, entretanto, pode ser um subconjunto de inteiros positivos. Uma extremidade com multiplicidade “muitos” indica que vários objetos desta podem estar ligados a um único objeto da extremidade oposta. Porém, existe apenas uma ligação entre um par de objetos (exceto para *bags* e *sequences*, vistas mais adiante). Se houver necessidade de mais de uma ligação entre este par, deve-se criar mais associações.

A notação da UML para multiplicidade varia de um número fixo à um intervalo, incluindo o caractere asterisco (*). Utiliza-se dois pontos seguidos para denotar um intervalo. Por exemplo, a classe Carro está associada a cinco objetos da classe Pneu, portanto coloca-se a notação “5” na extremidade Pneu da linha de associação. Já a classe Conta pode ter zero ou mais transferências, por isso anota-se com “0..*”. A **Figura 8** apresenta este exemplo. Também são válidas anotações como “0..3” (zero ou até três) e “0..1, 3..4, 6..*” (zero ou mais, exceto dois e cinco).

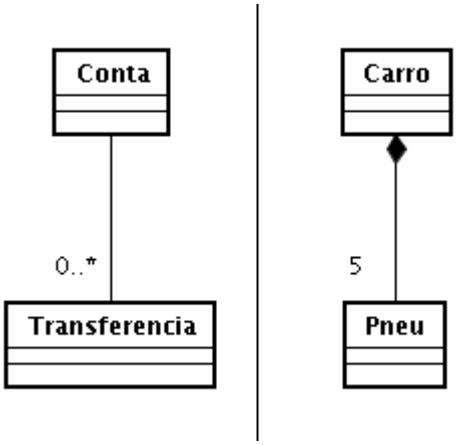


Figura 8 – Exemplo de multiplicidade

A UML fornece três anotações relacionadas à multiplicidade “muitos”: *ordered*, *bag* e *sequence*. *Ordered* é utilizado em situações onde os objetos devem aparecer ordenados. Um exemplo de sua utilização é na modelagem de uma tela de computador, onde várias janelas são abertas e a ordem de abertura deve ser respeitada. Esta anotação é utilizada quando a

ordenação é parte inerente da associação.

A anotação *bag* representa objetos não ordenados e permite duplicatas, ou seja, podem existir várias ligações para um par de objetos. A anotação *sequence* é semelhante à *bag* exceto pelo fato de que os objetos são ordenados.

As anotações *ordered*, *bag* e *sequence* são representadas respectivamente pelos símbolos {ordered}, {bag} e {sequence}, os quais são colocados perto da linha da extremidade "muitos" (RUMBAUGH; BLAHA, 2006). A **Figura 9** exibe um exemplo das anotações *ordered* e *sequence*, através da modelagem de uma agenda que contém tarefas e contatos. Os contatos devem estar ordenados mas não podem estar duplicados, enquanto que as tarefas devem estar ordenadas e aceitam repetições.

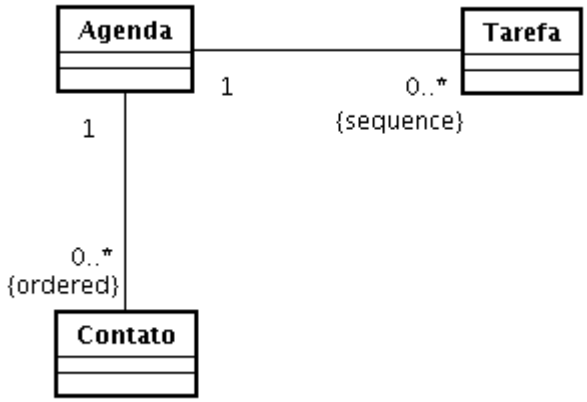


Figura 9 – Exemplo das anotações *ordered* e *sequence*

### 1.1.4 Herança

Em um domínio, algumas classes apresentam atributos e operações semelhantes. Em um modelo de sistema de concessionária, as classes *Carro*, *Motocicleta* e *Caminhão* possuem o atributo *placa* e a operação *mover* em comum. A implementação isolada destas classes resulta em código duplicado e consequente dificuldade de manutenção. A POO oferece um mecanismo chamado *herança* que encapsula código comum entre classes. A aplicação deste mecanismo neste exemplo resulta em uma classe *Automóvel*, com o atributo *placa* e a operação *mover,* e as demais classes herdam estes membros.

As classes podem ser organizadas em uma hierarquia de herança, em que classes mais gerais (ou mais abstratas) são apresentadas no topo da hierarquia (SOMMERVILLE,

2007). O relacionamento entre uma subclasse e sua superclasse é chamado de *generalização*, e o inverso chama-se *especialização*. Uma superclasse é uma generalização de suas subclasses e uma subclasse é uma especialização de sua superclasse. A interpretação deste relacionamento geralmente é "é um", pois instâncias da subclasse são também instâncias da superclasse (RUMBAUGH; BLAHA, 2006). Por exemplo, dadas as classes *Pessoa* e *Estudante*, em que esta é subclasse daquela, afirma-se que um objeto da classe *Estudante* "é um" objeto da classe *Pessoa*. A classe *Estudante*, além de ter os membros da classe *Pessoa*, pode ter seus próprios atributos e operações, e inclusive modificar as operações herdadas. No exemplo do jogo de xadrez, tem-se as classes *Bispo* e *Cavalo* que são especializações da classe Peça, conforme **Figura 10**. Esta, por sua vez, possui a operação *mover*. As subclasses implementam esta operação de acordo com suas necessidades, ou seja, a classe *Bispo* define que seus objetos deverão se movimentar pelas diagonais do tabuleiro, enquanto que a classe *Cavalo* define a movimentação em "L".

O símbolo da UML para generalização é uma grande seta vazada que parte da subclasse até a superclasse.

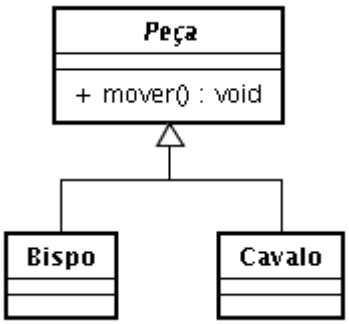


Figura 10 – Hierarquia da classe *Peça*

# CAPÍTULO 2 – PERSISTÊNCIA DE OBJETOS

A persistência é o armazenamento de dados de um sistema em mídia secundária de armazenamento, como discos rígidos, pen-drives, etc (RASHID; CHITCHYAN, 2003). Em sistemas orientados a objetos, a persistência permite que objetos vivam além de seu ambiente de execução (WITTHAWASKUL; JOHNSON, 2003). A persistência deve garantir que o *layout* dos objetos armazenados em disco seja exatamente igual ao *layout* destes em memória (HORSTMAN; CORNELL, 2001).

## 2.1 Serialização de objetos

A plataforma Java fornece um mecanismo para persistir objetos chamado *serialização*. Ele também é utilizado para enviar objetos por uma rede de computadores. Este mecanismo consiste em converter os dados do objeto, como seus atributos e classe, em uma sequência de *bytes* que é enviada por um *fluxo(stream) de saída*. Em Java, entende-se por *fluxo de saída* um objeto no qual é possível escrever uma sequência de *bytes*. Este objeto é uma instância da classe *OutputStream*, do pacote *java.io*, ou de uma de suas subclasses. Há também *fluxos de entrada*, que são objetos que permitem a leitura de uma sequência de *bytes* e são instâncias da classe *InputStream* (HORSTMAN; CORNELL, 2001).

Existem mais de 60 tipos de fluxos diferentes na plataforma Java. Por exemplo, há as classes *ZipInputStream*, do pacote *java.util.zip*, e *AudioInputStream*, do pacote *javax.sound.sampled*, que são próprias para a leitura de arquivos ZIP e de áudio, respectivamente (JAVA..., 2009).

É possível combinar os fluxos para obter algumas vantagens na manipulação de diferentes tipos de dados. Por exemplo, a classe *FileOutputStream*, utilizada para escrever uma sequência de *bytes* em um determinado arquivo, não provê meios de escrever tipos de dados mais especializados, como inteiros e números de ponto flutuante. Por outro lado, a classe *DataOutputStream* contém operações que escrevem os tipos primitivos da linguagem Java, como *int*, *float* e *double*. Uma aplicação que precise escrever estes tipos primitivos em um arquivo pode combinar estes dois fluxos, ou seja, pode utilizar o fluxo *FileOutputStream*, que escreve em arquivos, junto com o fluxo *DataOutputStream*, que escreve tipos primitivos (HORSTMAN; CORNELL, 2001). A parte (a) da **Figura 11** exibe o código em Java da classe

executável *Fluxos.java*, que exemplifica a utilização destes dois fluxos citados, e a parte (b) mostra a saída gerada por esta classe, no término de sua execução.

(a)

```
import java.io.*;

public class Fluxos {

       public static void main(String[] args) throws IOException {

              FileOutputStream fileStream = new FileOutputStream("primitivos.dat");
              DataOutputStream dataStream = new DataOutputStream(fileStream);

              dataStream.writeDouble(123.4d);
              dataStream.writeFloat(10.3f);
              dataStream.writeInt(5);

              dataStream.close();

              FileInputStream primitivos = new FileInputStream("primitivos.dat");
              DataInputStream leitor = new DataInputStream(primitivos);

              System.out.println("Escrevi double:" + leitor.readDouble());
              System.out.println("Escrevi float: " + leitor.readFloat());
              System.out.println("Escrevi int:   " + leitor.readInt());

              leitor.close();
       }
}
```

(b)

```
Escrevi double:123.4
Escrevi float: 10.3
Escrevi int:   5
```

Figura 11 – Classe *Fluxos.java* e saída gerada

Na linha 8, o fluxo que escreve no arquivo *primitivos.dat* é passado como argumento ao construtor do fluxo que escreve tipos primitivos. Desta forma, toda escrita feita por este fluxo é redirecionada àquele fluxo. O processo de leitura de *primitivos.dat* é semelhante ao de escrita. As classes utilizadas foram *FileInputStream* e *DataInputStream*, que lê dados de um arquivo e lê tipos primitivos de uma sequência de *bytes*, respectivamente. A sequência de leitura deve ser a mesma de gravação. Neste exemplo, os dados foram gravados e lidos como *double*, *float* e *int*, nesta ordem (JAVA, 2009).

Para persistir objetos em arquivos, pode-se utilizar a classe *ObjectOutputStream* juntamente com a classe *FileOutputStream*. A classe *ObjectOutputStream* contém a operação *writeObject*, responsável por converter o objeto passado por parâmetro em uma sequência de *bytes*, e a classe *ObjectInputStream* contém a operação *readObject*, responsável por converter

uma sequência de *bytes* em um objeto. Apenas objetos que implementam a interface *Serializable*, do pacote *java.io*, podem ser persistidos. Esta interface não possui nenhuma operação e sua utilidade é confinada à indicação de classes cujos objetos são passíveis de serialização.

Quando um objeto é persistido e posteriormente recuperado, seu endereço de memória é geralmente diferente de seu endereço antigo. Isto pode ser considerado um problema quando persiste-se objetos que contém referências a outros objetos, pois as referências são internamente endereços de memória. Entretanto, o mecanismo de serialização garante que um objeto referenciado por dois ou mais objetos continue sendo o mesmo objeto referenciado, mesmo alterando seu endereço de memória, durante a *desserialização*. Por exemplo, dois objetos da classe *Funcionário* possuem uma referência a um mesmo objeto da classe *Subordinado*. Ao persistir estes objetos em um único arquivo e recuperá-los novamente, o mecanismo criará uma única instância da classe *Subordinado* e a ligará aos demais objetos. Isto acontece porque é atribuído um número de série a todo objeto enviado a um fluxo. Por isso este mecanismo tem o nome de serialização. Ao enviar um objeto a um fluxo, o mecanismo verifica se este objeto já foi enviado e, em caso afirmativo, é enviado apenas seu número de série (HORSTMAN; CORNELL, 2001).

O framework de persistência Prevayler (PREVAYLER, 2009) utiliza o mecanismo de serialização para persistir objetos. Os objetos são mantidos em memória principal e as alterações são registradas através de objetos de comando, que implementam o padrão de projeto *Command* (GAMMA et al., 1995), os quais são persistidos em arquivos de *log*. Periodicamente todos os objetos do sistema são persistidos em um arquivo chamado de *snapshot*. Durante a reinicialização do sistema, o último *snapshot* é recuperado, e todos os objetos de comando que foram gravados em arquivos de *log* são executados, voltando o último estado da aplicação.

## 2.2 Bancos de dados relacionais

Bancos de dados relacionais são bancos de dados que seguem o modelo relacional, descrito por Codd (1970) e são amplamente utilizados na persistência de dados, principalmente em sistemas comerciais.

Os bancos de dados relacionais são dados organizados em *tabelas*. Cada linha de

uma tabela é um *relacionamento* entre valores e é considerada uma tupla (*tuple*), isto é, uma sequência de valores cuja repetição é permitida (TUPLE, 2009); os valores são descritos pelas colunas da tabela e cada coluna tem um nome único. Uma tabela é uma coleção de tais relacionamentos e corresponde ao conceito matemático de *relação*. O modelo relacional proposto por Codd organiza o banco de dados em um conjunto de relações. Este modelo atraiu a atenção devido à sua simplicidade e base matemática (SILBERSCHATZ et al., 1999; ELMASRI; NAVATHE, 2005). A **Figura 12** exemplifica uma tabela de funcionários que contém duas colunas, *NOME* e *SALÁRIO*. Cada linha da tabela contém valores relacionados; a primeira linha indica que o funcionário José da Silva possui salário de 1500 reais.

| NOME | SALÁRIO |
|---|---|
| José da Silva | 1500 |
| Maria Caetano | 1200 |
| Eugenio de Souza | 1900 |
| Camilo Pereira | 2100 |
| Fabia Bernardes | 1700 |
| Manuela Fontes | 2000 |

Figura 12 – Tabela de funcionários

É utilizado um *sistema gerenciador de bancos de dados relacionais* (SGBDR) para acesso ao banco de dados. Um SGBDR é um conjunto de programas especializados na manipulação dos dados de um banco de dados relacional e proporciona aos seus usuários uma visão abstrata destes dados, isto é, acaba por ocultar determinados detalhes sobre a forma de armazenamento e manutenção desses dados (SILBERSCHATZ et al., 1999). São exemplos de SGBDR's: Oracle (ORACLE, 2009), MySql (MYSQL, 2009) e Microsoft SQL Server (MICROSOFT, 2009).

Um SGBDR fornece três níveis de abstração do banco de dados: *físico*, *lógico* e *visão*. O nível físico é o mais baixo nível de abstração e descreve a forma *como* os dados estão armazenados; suas estruturas complexas de baixo nível são expostas. O nível lógico vem imediatamente acima do nível físico e descreve *quais* dados estão armazenados e quais os inter-relacionamentos entre eles. Este nível é utilizado pelos administradores do banco de dados. O nível de visão é o mais alto nível de abstração e descreve apenas parte do banco de dados. É utilizado pela maioria dos usuários de bancos de dados, uma vez que eles não precisam conhecer todas as informações contidas no banco. Vários níveis de visão podem ser criados, porém há apenas um nível físico e um lógico (SILBERSCHATZ et al., 1999).

A modelagem de um banco de dados pode ser feita através do *modelo entidade-relacionamento* (MER) e esta pode ser representada pelo *diagrama entidade-relacionamento* (DER), ambos propostos por Peter Chen (CHEN, 1976). O MER define um banco de dados como conjuntos de entidades e conjuntos de relacionamentos entre essas entidades. Tanto as entidades quanto os relacionamentos são descritos por atributos e um nome. Cada entidade é identificada unicamente dentro de um conjunto de entidades através de um subconjunto de seus atributos, denominados de *atributos-chave*. Por exemplo, em um sistema de gerência de projetos, um projeto é uma entidade que pertence ao conjunto de entidades *Projeto* e é identificado unicamente por seu código. Um funcionário, que é uma entidade pertencente ao conjunto *Funcionário*, é identificado pelo registro F-1 e é responsável pelo projeto P-1, portanto existe um relacionamento entre este funcionário e o projeto. Este relacionamento faz parte do conjunto de relacionamentos *Responsabilidade*.

Existem conjuntos de entidades que não possuem atributos suficientes para identificar unicamente cada uma de suas entidades. Tais conjuntos são denominados *conjuntos de entidades fracas*. De forma inversa, conjuntos que possuem univocidade garantida são chamados de *conjuntos de entidades fortes*. Para que um determinado conjunto de entidades fracas *A* seja significativo, deve fazer parte de um conjunto de relacionamentos *R* juntamente com um conjunto de entidades *B*, tal que uma entidade em *B* pode se relacionar com várias entidades em *A*, mas uma entidade em *A* pode estar relacionada a apenas uma entidade em *B*; este tipo de relacionamento é denominado *muitos para um*. Apesar de um conjunto de entidades fracas não possuir atributos-chave suficientes para a univocidade, deve possuir atributos que identifiquem um conjunto de entidades que dependem de uma determinada entidade forte. O conjunto destes atributos é denominado *identificador* (SILBERSCHATZ et al., 1999). Por exemplo, tem-se os conjuntos de entidades *Empréstimo* e *Pagamento* em que este último possui os atributos *número* e *valor*. Cada pagamento possui um número único dentro do conjunto de pagamentos feitos a um determinado empréstimo. Dois pagamentos feitos a empréstimos diferentes podem possuir o mesmo número, assim, *Pagamento* é um conjunto de entidades fracas. Todavia, dois pagamentos a um mesmo empréstimo não podem possuir o mesmo número, portanto, o atributo *número* os identificam dentro do conjunto de pagamentos feitos a este empréstimo.

No DER, os conjuntos de entidades são representados por retângulos, e cada um contém o nome do conjunto representado. Os atributos são representados por elipses

conectadas a seus respectivos conjuntos através de linhas; atributos-chave são sublinhados. Os relacionamentos são representados por losangos e os conjuntos de entidades que participam dos relacionamentos são conectados por linhas. Os conjuntos de entidades fracas e os conjuntos de relacionamentos que assim os definem são representados por linhas duplas, e os atributos identificadores são sublinhados com uma linha tracejada. A **Figura 13** mostra um exemplo de DER.

Após a modelagem de um banco de dados através do MER, o mesmo pode ser convertido em tabelas. De forma geral, um conjunto de entidades é transformado em uma tabela, onde seus atributos são mapeados em colunas e cada entidade fica representada por uma linha. Isto não ocorre, por exemplo, com atributos *multivalorados*, que aceitam mais de um valor; estes são convertidos em tabelas ao invés de serem mapeados para campos.

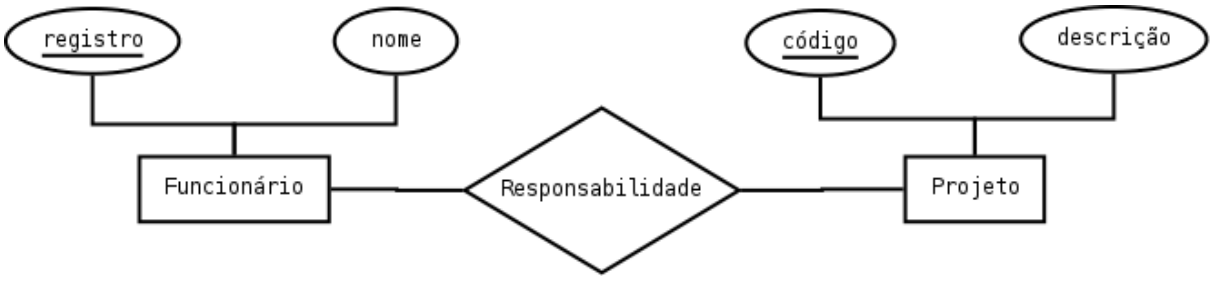


Figura 13 – Exemplo de DER

Os conjuntos de relacionamentos podem ser transformados em tabelas cujo conjunto de campos é composto pelos atributos-chave dos conjuntos de entidades participantes mais o conjunto de atributos do próprio conjunto de relacionamentos. Porém, há a possibilidade de combinar a tabela resultante com alguma outra tabela gerada pelos conjuntos de entidades participantes do relacionamento. A conversão de um conjunto de relacionamentos *AB* entre os conjuntos de entidades *A* e *B*, de acordo com o esquema de conversão previamente descrito, resulta em três tabelas: *A*, *B* e *AB*. Mas, se entidades em A podem se relacionar com apenas uma entidade em B, e esta pode se relacionar com várias em *A,* denotando um relacionamento muitos para um, e se cada entidade em *A* depender da existência de alguma entidade em *B*, pode-se combinar as tabelas *A* e *AB* para formar uma única tabela *AB*, composta pela união das colunas de ambas as tabelas (SILBERSCHATZ et al., 1999). Por exemplo, tem-se os conjuntos de entidades *Pedido* e *Item* e um conjunto de relacionamentos muitos para um

*Pedido_Item* entre estes conjuntos. A conversão destes conjuntos resulta em três tabelas: *Pedido, Item* e *Pedido_Item*. Como o relacionamento é muitos para um e um item depende da existência de um pedido, pode-se combinar as tabelas *Item* e *Pedido_Item*. Na **Figura 14** a tabela *Item* recebe as colunas da *Pedido_Item*.

Os SGBDR's atuais possuem uma linguagem integrada subdividida em dois tipos: *data definition language* (DDL) e *data manipulation language* (DML). A DDL é utilizada para a definição do nível físico, lógico e de visão. Já a DML é utilizada para consultas e alterações. A linguagem integrada mais utilizada é a *Structured Query Language* (SQL). Existem outras linguagens, como a *Query-By-Example* (QBE), *Quel* e *Datalog*.

A SQL é baseada na álgebra relacional. A álgebra relacional define operações que podem ser feitas em uma ou mais relações, de forma a obter uma relação como resultado. A esta relação as mesmas operações podem ser aplicadas para obter outra relação, e assim por diante. Uma sequência de operações forma uma expressão de álgebra relacional, cujos resultados sempre serão uma relação que representa o resultado de uma consulta de banco de dados. As operações fundamentais são *seleção* (select), *projeção* (project), *produto cartesiano* (cartesian product), *união* (union), *diferença entre conjuntos* (set difference) e *renomear* (rename). Este subcapítulo apresenta três operações: seleção, *projeção* e *produto cartesiano*. Existem outras operações, como *intersecção entre conjuntos* (set intersection), *junção natural* (natural join), *divisão* (division) e *designação* (assignment), que são definidas em termos das fundamentais.

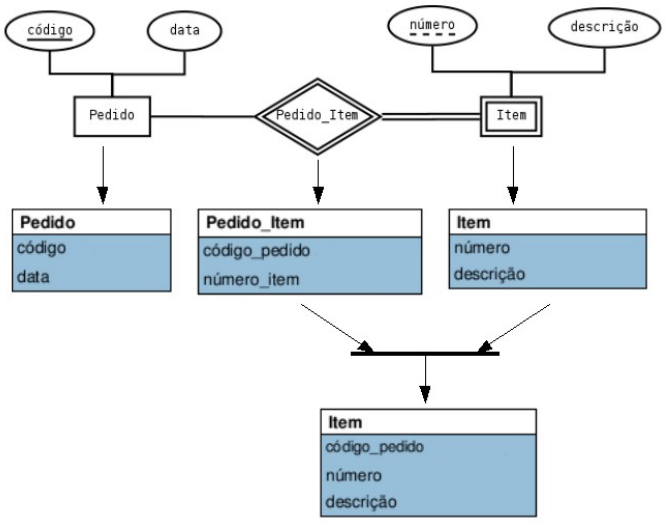


Figura 14 – Conversão de DER para tabelas

A operação *seleção* seleciona tuplas de uma relação que satisfaçam a uma determinada condição. É representada pela letra grega sigma (σ), a condição aparece subscrita e a relação alvo é dada entre parênteses. Como exemplo, a seleção de todos os projetos que custam mais de mil reais da relação *Projeto* pode ser denotada por: $\sigma_{valor > 1000}(Projeto)$.

A operação *projeção* seleciona atributos de uma relação. Se tabela é análoga a relação, pode-se afirmar que esta operação seleciona certas colunas e descarta as demais. Como o resultado obtido é um conjunto, as duplicatas são eliminadas. É representada pela letra pi (π) e uma lista de atributos subscrita separados por vírgula. Por exemplo, a projeção da descrição e do valor dos projetos que custam mais de mil reais pode ser denotada por: $\pi_{descrição, valor}(\sigma_{valor > 1000}(Projeto))$.

A operação *produto cartesiano* é denotada por (x), recebe duas relações como entrada e produz uma relação com todas as combinações de tuplas possíveis. Se as relações *r1* e *r2* possuem respectivamente *t1* e *t2* tuplas, então a expressão *r1* x *r2* obtém uma relação com *t1* vezes *t2* tuplas (SILBERSCHATZ et al., 1999; ELMASRI; NAVATHE, 2005). Por exemplo, para a obtenção de uma relação contendo o nome dos funcionários que possuem dependentes, pode-se realizar um *produto cartesiano* entre as relações *Funcionário* e *Dependente*, selecionar as tuplas cujo código do funcionário, em *Funcionário*, seja igual ao código do funcionário, em *Dependente*, e projetar o nome do funcionário. A seleção é necessária porque se existem três funcionários *João*, *Maria* e *José* e dois dependentes *Pedro* e *Mateus*, o produto cartesiano é {(*João, Pedro*), (*João, Mateus*), (*Maria, Pedro*), (*Maria, Mateus*), (*José, Pedro*), (*José, Mateus*)}; como *Pedro* é dependente apenas de *João,* e *Mateus* é dependente apenas de *Maria*, a relação desejada é {(*João, Pedro*), (*Maria, Mateus*)}. Para a distinção de dois atributos que possuem mesmo nome mas pertencem a relações diferentes, pode-se prefixá-los com o nome da relação seguido de um ponto. Portanto, o exemplo acima pode ser denotado por: $\pi_{Funcionário.nome}(\sigma_{código = código\_funcionário}(Funcionário \times Dependente))$.

A parte da linguagem SQL destinada a consultas possui uma instrução básica chamada de *SELECT*. Esta instrução não possui relação alguma com a operação *select* da álgebra relacional. Outra diferença é que, ao contrário da álgebra relacional, as tabelas em SQL não são conjuntos de tuplas, os quais não permitem duplicatas, mas são considerados *multiconjuntos* (às vezes chamados de *bags – bolsas*) de tuplas. Como exemplo, a consulta anterior pode ser escrita da seguinte forma em SQL: *SELECT* Funcionario.nome *FROM* Funcionario, Dependente *WHERE* codigo = codigo_funcionario. Após a instrução *SELECT*

vem a lista de campos que devem ser exibidos; após a instrução *FROM* vem a lista de tabelas; após a instrução *WHERE* vem a condição de seleção das linhas resultantes do produto cartesiano entre as tabelas referenciadas (ELMASRI; NAVATHE, 2005).

## 2.3 Bancos de dados orientados a objetos e objeto-relacionais

A tecnologia de banco de dados recentemente tem sido utilizada em aplicações alheias ao processamento de dados convencionais, como *projeto auxiliado por computador* (*computer-aided design – CAD*), *engenharia de software auxiliada por computador* (*computer-aided software engineering – CASE*), *bancos de dados multimídia*, entre outros. Aplicações CAD necessitam armazenar projetos de engenharia, incluindo componentes do item projetado e seus inter-relacionamentos, aplicações CASE armazenam códigos-fonte, dependências e histórico de desenvolvimento de um software e bancos de dados multimídia contém imagens, vídeos, áudio e afins. O modelo relacional não supre de maneira adequada as exigências destas aplicações e mesmo aplicações financeiras tradicionais que tem se tornado mais complexas com o tempo. Os bancos de dados orientados a objetos e objeto-relacionais foram propostos para fazer face às necessidades destas aplicações.

Os sistemas gerenciadores de bancos de dados orientados a objetos (SGBDOO) seguem o paradigma orientado a objetos para a modelagem de dados e utilizam a linguagem *object query language* (OQL) para definição e consulta de objetos. Por outro lado, os sistemas gerenciadores de bancos de dados objeto-relacionais (SGBDOR) unem conceitos do modelo orientado a objetos com o modelo relacional e a linguagem utilizada é a SQL com extensões orientadas a objetos chamada de $SQL_3$ (ELMASRI; NAVATHE, 2005). São exemplos de SGBDOO's: ObjectStore (OBJECTSTORE, 2009), Versant (VERSANT, 2009) e Caché (CACHÉ, 2009); e exemplos de SGBDOR's: Oracle (ORACLE, 2009) e PostgreSQL (POSTGRESQL, 2009).

# CAPÍTULO 3 – JAVA PERSISTENCE API (JPA)

JPA é uma API (*application programming interface*) Java de gerenciamento da persistência e mapeamento objeto-relacional (*object-relational mapping - ORM*) que pode ser utilizada nas plataformas *enterprise edition* (Java EE) e *standard edition* (Java SE). A JPA é especificada pelo *Java Community Process* (JCP) (THE JAVA..., 2009) e pode ser implementada para gerar um produto comercial ou livre, chamado de *provedor* (DEMICHIEL; KEITH, 2006). Entre os provedores estão Hibernate (HIBERNATE, 2009), OpenJPA (OPENJPA, 2009) e EclipseLink (ECLIPSELINK, 2009). Este capítulo aborda a versão 1 desta API.

## 3.1 Mapeamento objeto-relacional (ORM)

ORM é uma técnica de tradução do modelo orientado a objetos para o modelo relacional. Para a elaboração de um ORM é necessário considerar diversos aspectos, tanto do modelo orientado a objeto quanto do modelo relacional. Esta técnica surgiu para diminuir o impedimento, conhecido como *impedance mismatch,* entre estes dois modelos. Tal impedimento existe dadas as diferenças entre o paradigma relacional, que considera apenas dados, e o paradigma orientado a objetos, que considera dados e comportamento. Para acessar informações na POO, atravessa-se os objetos através de seus relacionamentos enquanto que no modelo relacional realiza-se junções de tuplas de tabelas. Além disso, relacionamentos muitos para muitos, que são suportados nativamente pela POO, são implementados através de dois relacionamentos um para muitos no paradigma relacional. Portanto, estes dois paradigmas não se encaixam perfeitamente (AMBLER, 2009).

Os atributos de uma classe podem ser mapeados para zero ou mais colunas em um banco relacional. Um atributo não é mapeado para uma coluna quando este representa dados transientes, ou seja, temporários. Por exemplo, uma classe *Boletim* pode conter um atributo *média*, o qual é derivado de um vetor de notas e, portanto, pode ser transiente. Há atributos que são mapeados em mais de uma coluna, os quais são referências a objetos que compõem o estado do objeto proprietário. Por exemplo, uma classe *Cliente* pode conter um atributo *endereço* que referencie um objeto da classe *Endereço*, a qual contém os atributos *logradouro*, *número* e *bairro*; o atributo *endereço* pode ser mapeado em três colunas correspondentes aos

atributos da classe *Endereço*.

Segundo Ambler (2009), existem dados que um objeto deve manter, os quais são necessários para a sua persistência, chamados de *shadow informations*. Tais dados tipicamente incluem atributos que representam chaves primárias não naturais, ou seja, que não possuem nenhum significado para o modelo de negócio, e atributos de controle de concorrência, como atributos de controle de versão de tuplas. A UML, porém, convencionou que *shadow informations* não devem ser modeladas.

### 3.1.1 Mapeamento de classes

As classes são mapeadas em tabelas do banco de dados. Os relacionamentos entre classes podem ser mapeados como chaves estrangeiras ou tabelas, dependendo do tipo de relacionamento. Como um banco de dados não suporta herança nativamente, foram desenvolvidas quatro maneiras de mapear uma hierarquia de classes para tabelas, as quais são: *uma tabela para uma hierarquia*, *uma tabela para cada classe*, *uma tabela para cada classe concreta* e *classes para uma estrutura genérica de tabelas*.

#### 3.1.1.1 Uma tabela para uma hierarquia

Nesta abordagem, os atributos de todas as classes pertencentes à uma determinada hierarquia são mapeados a colunas de uma única tabela. Geralmente, esta tabela tem o mesmo nome da classe raiz da hierarquia. Cada linha da tabela representa um objeto. Uma coluna extra é criada para identificar qual classe este objeto pertence. Esta coluna é chamada de *coluna discriminadora*. Por exemplo, uma coluna discriminadora pode receber o valor *F* para indicar que uma determinada tupla representa um funcionário, o valor *C* para indicar que é um cliente ou o valor *A* para indicar ambos. Quando houver combinação de classes, como quando um objeto pode ser tanto um cliente quanto um funcionário, sugere-se a utilização de várias colunas booleanas. A coluna discriminadora do exemplo anterior pode ser substituída pelas colunas *é_funcionário* e *é_cliente* (AMBLER, 2009).

Esta abordagem proporciona bom desempenho e simplicidade. Se novas classes surgirem na hierarquia, é necessário apenas adicionar mais colunas na tabela. Porém, há um potencial desperdício de espaço, visto que nem todas as colunas são preenchidas. Além disso,

em casos de grandes hierarquias a tabela pode crescer demasiadamente.

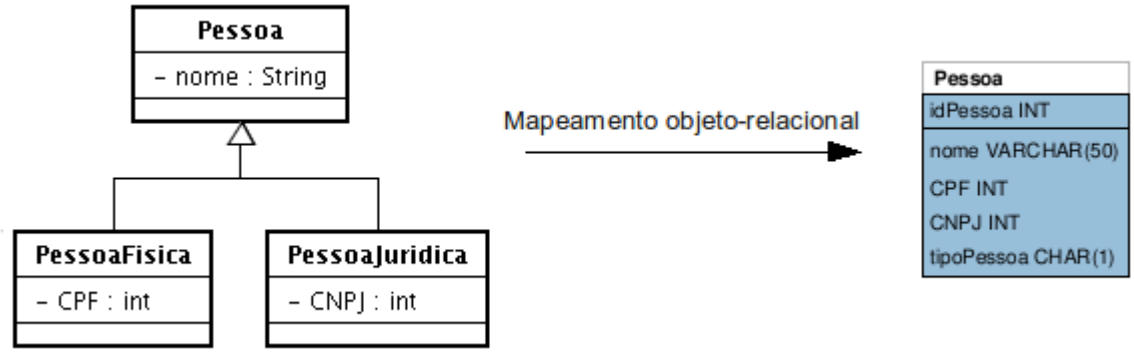


Figura 15 – Abordagem de uma tabela para uma hierarquia

A **Figura 15** exibe uma hierarquia de classes mapeada conforme abordagem previamente descrita. A tabela tem o mesmo nome da classe raiz da hierarquia, uma coluna de chave primária sem significado dentro do modelo de negócio chamada *idPessoa* e uma coluna discriminadora chamada *tipoPessoa*.

### 3.1.1.2 Uma tabela para cada classe concreta

Nesta abordagem, cada classe concreta é mapeada a uma tabela do banco de dados. Assim, todos os atributos de uma determinada classe, inclusive os atributos herdados, são mapeados a colunas de uma mesma tabela. Na **Figura 16** tem-se a mesma hierarquia de classes mostrada na **Figura 15** mapeada em duas tabelas.

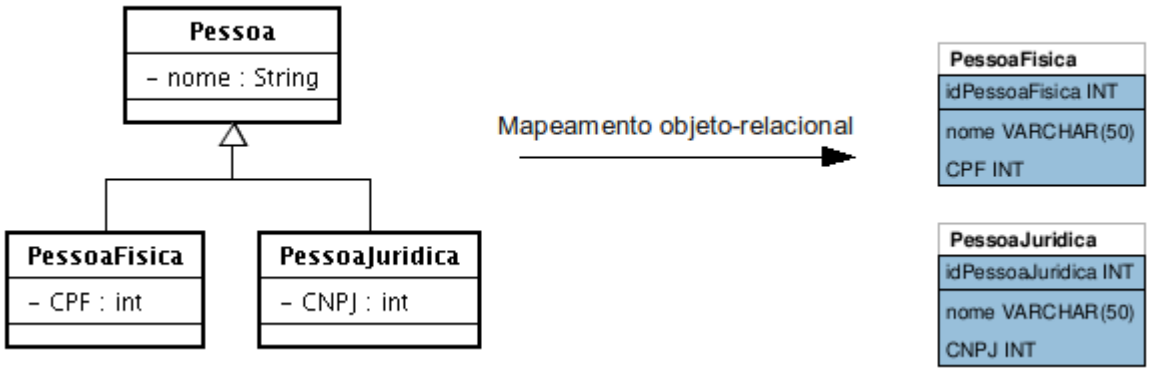


Figura 16 – Abordagem de uma tabela para cada classe concreta

Com esta abordagem, a manutenção do banco de dados se torna complexa pois qualquer alteração na classe raiz implica na alteração de todas as tabelas pertencentes à

hierarquia (AMBLER, 2009).

### 3.1.1.3 Uma tabela para cada classe

Nesta abordagem, cada classe é mapeada a uma tabela do banco de dados, com uma coluna por atributo e qualquer *shadow informations* necessárias. Esta é a mais simples das abordagens pois, como o mapeamento entre classes e tabelas é direto, sua interpretação é facilitada. Não há desperdício de espaço em disco pois todas colunas são preenchidas. Entretanto, o desempenho de consultas e atualizações é potencialmente mais lento pois há a necessidade de percorrer um maior número de tabelas (AMBLER, 2009).

A **Figura 17** mostra a mesma hierarquia mapeada através desta abordagem. As tabelas *PessoaFisica* e *PessoaJuridica* receberam chaves primárias que também são chaves estrangeiras, as quais referenciam uma tupla da tabela *Pessoa*. Para gravar um objeto da classe *PessoaFisica*, deve-se inserir primeiro uma tupla na tabela *Pessoa* para depois inserir uma tupla na tabela *PessoaFisica*.

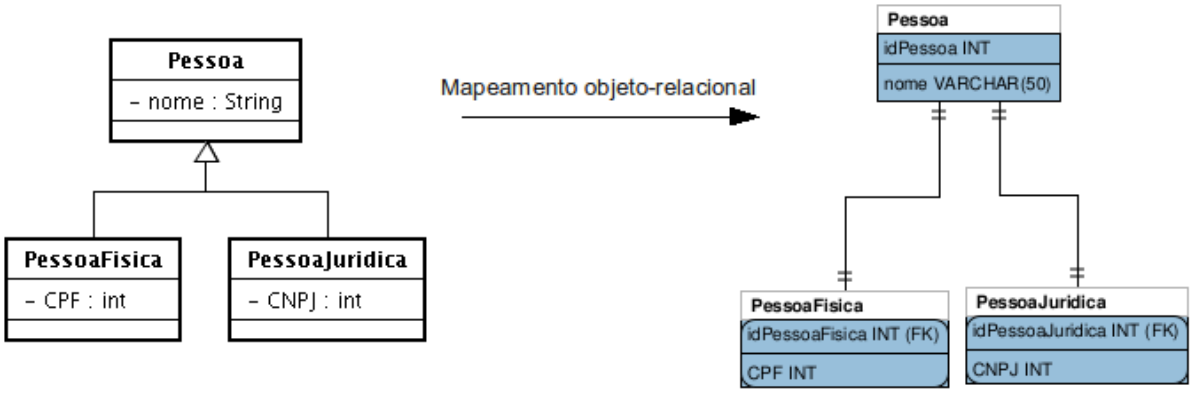


Figura 17 – Abordagem de uma tabela para cada classe

### 3.1.1.4 Classes para uma estrutura genérica de tabelas

Nesta abordagem, as metainformações das classes e objetos são armazenadas em tabelas e, por isso, algumas vezes é chamada de *abordagem dirigida a metadados*.

A **Figura 18** mostra um modelo de banco de dados capaz de armazenar valores de atributos e relacionamentos de herança. Para simplificar, este modelo não suporta associações

entre objetos e assume que todo objeto tenha um identificador simples e único, a ser gravado na coluna *idObjeto* da tabela *Valor*. O valor de um atributo é armazenado em uma tupla da tabela *Valor*; assim, para armazenar um objeto com dez atributos, são necessárias dez tuplas desta tabela.

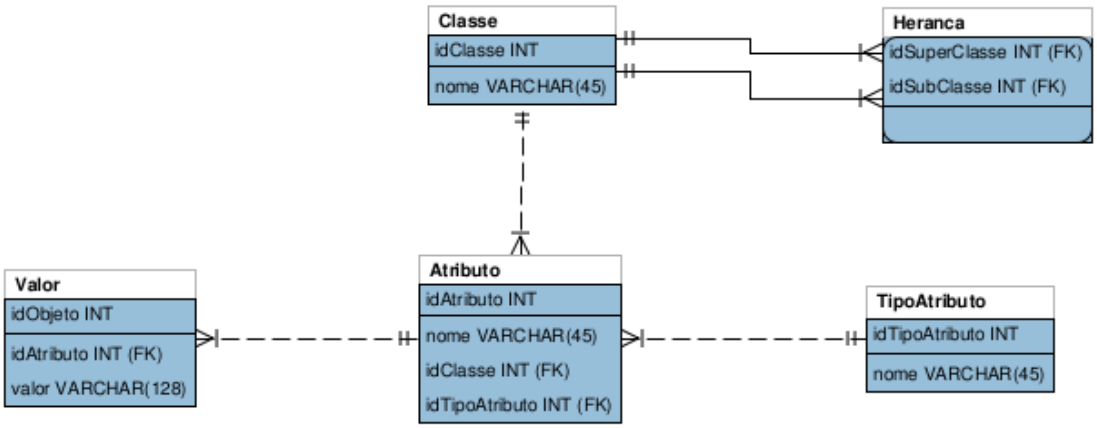


Figura 18 – Abordagem de classes para uma estrutura genérica de tabelas

Apesar de ser flexível, a implementação é complexa e o desempenho é lento pois para recuperar um único objeto, há a necessidade de acessar várias linhas do banco de dados. Esta abordagem é indicada para aplicações complexas que gerenciam pequenas quantidades de dados (AMBLER, 2009).

### 3.1.2 Mapeamento de relacionamentos

Os relacionamentos entre objetos são classificados pela multiplicidade e direcionabilidade. Em termos de multiplicidade, os relacionamentos podem ser um para um, um para muitos e muitos para muitos. Em termos de direcionabilidade, podem ser unidirecionais e bidirecionais.

Em um relacionamento bidirecional entre dois objetos, um possui uma referência ao outro e vice versa. Já em relacionamentos unidirecionais apenas um objeto possui uma referência ao outro, ou seja, apenas um dos objetos sabe sobre o outro ao qual ele está relacionado.

O relacionamento um para um entre as classes *Carro* e *Motor*, por exemplo, pode ser

implementado através da inclusão de um atributo *motor* em *Carro* e duas operações de acesso, *getMotor()* e *setMotor()*, em que a primeira retorna este atributo e a segunda atribui a ele um valor. Se o relacionamento for bidirecional, também haverá em *Motor* um atributo *carro* e as operações *getCarro()* e *setCarro()*. Operações cujo nome começa com *set* ou *get* são denominadas operações de acesso e são convencionadas pela especificação da arquitetura de componentes *JavaBeans* (FLANAGAN, 1999).

O relacionamento um para muitos pode ser implementado ou através da inclusão de uma coleção de objetos na classe "um" ou da inclusão de uma referência a um objeto da classe "um" no lado "muitos". Se o relacionamento for bidirecional, os dois atributos devem ser implementados. Por exemplo, a classe *Carro* tem um relacionamento um para muitos com a classe *Pneu*. Assim, a classe *Carro* recebe uma coleção de objetos da classe *Pneu* e esta pode ser implementada através de estruturas de dados como listas, vetores, conjuntos, entre outros. Em Java, podemos implementar as coleções como objetos da classe *ArrayList*, que encapsula um vetor, ou da classe *LinkedList*, que encapsula uma lista encadeada. A **Figura 19** exibe um trecho de código em que o relacionamento foi implementado através da inclusão de um atributo *pneus*, o qual é um objeto da classe *ArrayList*.

```
public class Carro {

	private Motor motor;
	private ArrayList<Pneu> pneus;

	…
}
```

Figura 19 – Implementação em Java do relacionamento um para muitos entre as classes *Carro* e *Pneu*

Um relacionamento muitos para muitos entre dois objetos pode ser implementado através da inclusão de uma coleção em cada um, de tal forma que, dadas as classes *A* e *B* em que *a* pertence ao conjunto de objetos da classe *A* e *b* pertence ao conjunto de objetos da classe *B*, *a* recebe uma coleção de *b* e *b* recebe uma coleção de *a*. Por exemplo, um funcionário pode estar associado a várias tarefas e uma tarefa pode estar associada a vários funcionários. Assim, a classe *Funcionário* possui uma coleção de objetos da classe *Tarefa* e esta possui uma coleção de objetos da classe *Funcionário*.

Existem situações em que um relacionamento possui seus próprios atributos, os quais não pertencem a nenhuma classe envolvida no relacionamento. Por exemplo, o

relacionamento muitos para muitos entre *Funcionário* e *Tarefa* pode possuir os atributos *atribuidoQuando* e *término*, e a forma de implementação descrita anteriormente já não é suficiente. Em casos como este, é implementada uma classe associativa, a qual recebe os atributos do relacionamento, uma referência a um objeto da classe *Funcionário* e uma referência a um objeto da classe *Tarefa*. A **Figura 20** exibe um trecho da implementação em Java do relacionamento muitos para muitos entre as classes *Funcionário* e *Tarefa*, representado pela classe associativa *TarefaAssociada*.

```
public class Funcionario {
  private String nome;
  private ArrayList<TarefaAssociada> tarefas
  ...
}

public class Tarefa {
  private String descricao;
  ...
}
```

```
public class TarefaAssociada {
  private Funcionario funcionario;
  private Tarefa tarefa;
  private Date atribuidoQuando;
  private Date termino;
  ...
}
```

Figura 20 – Implementação em Java de um relacionamento muitos para muitos com atributos

Os relacionamentos entre classes são implementados com chaves estrangeiras em bancos de dados relacionais. Uma chave estrangeira pertencente a uma tabela referencia a chave primária de outra tabela. Dadas as classes *A*, com os atributos $a_1, a_2, ..., a_n$, e *B*, com os atributos $b_1, b_2, ..., b_n$, o mapeamento em tabelas destas classes resulta nas tabelas *TA*, com as colunas $ta_1, ta_2, ..., ta_n$, e *TB*, com as colunas $tb_1, tb_2, ..., tb_n$. Há um relacionamento um para um entre *A* e *B*, em que *A* possui um atributo *b*, o qual é uma referência a um objeto da classe *B*. O mapeamento deste relacionamento é feito através da inclusão de uma chave estrangeira em *TA*, a qual é uma referência à chave primária de *TB*. Na **Figura 21** tem-se duas classes de um jogo de computador, *Lutador* e *Arma*, com um relacionamento um para um mapeado às tabelas *LUTADOR* e *ARMA*. As colunas *LutadorID* e *ArmaID* são *shadow informations* e representam a chaves primárias. A coluna *LutadorID* da tabela *ARMA* é uma chave estrangeira e está relacionada à coluna *LutadorID* da tabela *Lutador*. Nesta figura, esta coluna indica que a faca está associada ao Lutador 1 e o revólver está associado ao Lutador 2.

Os relacionamentos um para muitos são mapeados da mesma maneira, exceto pelo fato de que a chave estrangeira deve estar na tabela do lado muitos; em relacionamentos um

para um, a chave estrangeira pode estar em qualquer um dos lados.

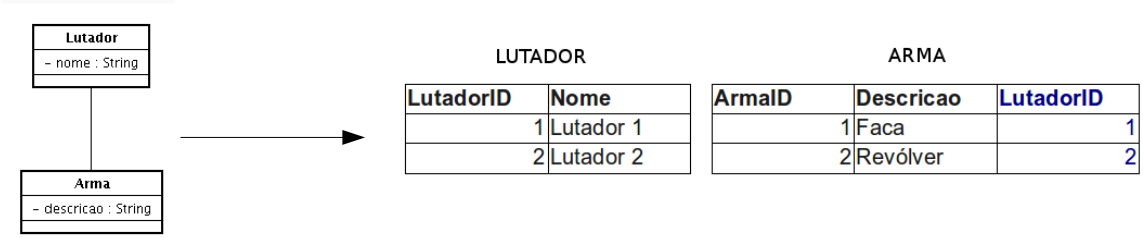


Figura 21 – Mapeamento de um relacionamento um para um

Em um banco de dados relacional, os relacionamentos são inerentemente bidirecionais, já que é possível acessar as tabelas em qualquer direção do relacionamento.

Os relacionamentos muitos para muitos são mapeados através da criação de uma tabela associativa, cujas colunas são a união das chaves primárias das tabelas envolvidas no relacionamento mais seus atributos. Uma tabela associativa está diretamente relacionada a uma classe associativa, porém, esta pode não existir caso o relacionamento em questão não possua atributos próprios (AMBLER, 2009). Na **Figura 22**, tem-se o mapeamento em tabelas das classes contidas na **Figura 20**.


Figura 22 – Relacionamento muitos para muitos mapeados em tabelas

## 3.2 ORM com JPA

Para realizar o mapeamento objeto-relacional com a JPA pode-se utilizar arquivos XML (BRAY, 2008) ou *anotações* (*annotations*) (GOSLING, 2005), as quais são abordadas neste subcapítulo. Os arquivos XML podem ser utilizados em conjunto com as anotações para sobrescreve-las ou para adicionar mais configurações (DEMICHIEL; KEITH, 2006).

### 3.2.1 Entidades

Os objetos persistentes são denominados *entidades* e não dependem necessariamente da JPA, ou seja, não precisam estender nenhuma classe pertencente à JPA. Uma classe de entidades deve ter a anotação *Entity*, um de seus construtores não deve ter parâmetro algum e a mesma não pode ser *final*, nem qualquer um de seus atributos persistentes ou operações. A palavra reservada *final*, quando aplicada a uma classe, indica que esta não pode ser estendida, ou seja, não pode haver subclasses desta; quando aplicada a um atributo, torna-o somente leitura e, quando aplicada a um método, o impede de ser sobrescrito. As classes de entidades podem ser abstratas e estender ou serem estendidas por outras classes não persistentes. Seus atributos não podem ser acessados diretamente, apenas por operações de acesso ou operações de negócio. É possível definir um atributo não persistente dentro de uma classe persistente através da anotação *Transient*.

A JPA pode manter o estado persistente de uma classe de entidades acessando seus atributos diretamente ou através de operações de acesso que seguem o estilo *JavaBeans*. A primeira forma é o *acesso baseado em atributos* (*field-based access*) e a segunda, *acesso baseado em propriedades* (*property-based access*). A segunda forma permite que operações de acesso com regras de negócio sejam invocadas durante a persistência. Nesta forma, só podem ser anotadas as operações de recuperação, ou seja, cujo nome inicie com *get* ou *is* (em caso de atributo booleano).

Toda entidade deve possuir uma chave primária, a qual é definida por um atributo simples ou um objeto composto. Atributos de chave primária, inclusive os atributos de um objeto composto, podem ser qualquer tipo primitivo Java ou objetos das classes *String* ou *Date*. Este atributo deve possuir a anotação *Id* e só pode ser implementado na raiz de uma hierarquia de classes. Por exemplo, dadas as classes *PessoaFísica* e *PessoaJurídica*, as quais são subclasses de *Pessoa*, o atributo de chave primária deve ser implementado na classe *Pessoa*, a qual é a raiz desta hierarquia.

A JPA relaciona nomes de classes e atributos com nomes de tabelas e de colunas, além de definir um padrão de nomeação de chaves estrangeiras e tabelas associativas. Desta forma, estes dados só devem ser informados quando estiverem fora do padrão (DEMICHIEL; KEITH, 2006). A parte (a) da **Figura 23** mostra um trecho de código Java da classe *Funcionario* anotada com *Entity* e o atributo *codigo* anotado com *Id*. A parte (b) mostra a

tabela a qual a classe é mapeada.

Anotações adicionais são utilizadas quando informações do banco de dados estão fora do padrão definido pela JPA, por exemplo, em caso de o nome da tabela não corresponder ao nome da classe, utiliza-se a anotação *Table* em conjunto com *Entity*, e informa-se na propriedade *name* o nome da tabela desejada. Se o nome da tabela na parte (b) da **Figura 23** fosse *TBL_FUNC*, seria necessário colocar *@Table(name="TBL_FUNC")* logo abaixo da anotação *Entity* na classe *Funcionario*. De forma semelhante, o nome das colunas pode ser especificado pela propriedade *name* da anotação *Column*, a qual pode ser colocada acima ou atrás do atributo, em caso de acesso baseado em atributos, ou da operação, em caso de acesso baseado em propriedades.

a)

```
@Entity
public class Funcionario {

	@Id
	private int codigo;
	private String nome;
	private Date admissao;
	private double salario;

	...
}
```

b)

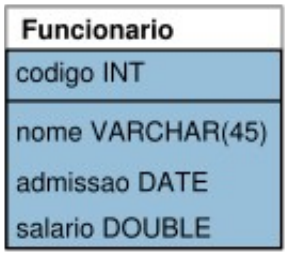


Figura 23 – Classe *Funcionario* e respectiva tabela

### 3.2.2 Mapeamento de hierarquias

Utiliza-se a anotação *Inheritance* na classe raiz de uma hierarquia para definir a estratégia de mapeamento. Esta anotação possui a propriedade *strategy* e é do tipo enumerado *InheritanceType*, o qual possui três valores, cada um relacionado a uma estratégia. Estes valores são *SINGLE_TABLE*, que representa *uma tabela para uma hierarquia*, *TABLE_PER_CLASS*, que representa *uma tabela para cada classe concreta*, e *JOINED*, que representa *uma tabela para cada classe*.

A anotação *DiscriminatorColumn* indica a coluna discriminadora para as estratégias *SINGLE_TABLE* e *JOINED*. Esta anotação possui a propriedade *discriminatorType* de tipo enumerado *DiscriminatorType* e aceita três tipos de dados para representar o valor discriminador, os quais são *STRING*, *CHAR* e *INTEGER*. Esta anotação só deve ser utilizada

na classe raiz. As classes pertencentes à hierarquia podem utilizar a anotação *DiscriminatorValue* para indicar, através de sua propriedade *value*, o valor discriminador (DEMICHIEL; KEITH, 2006). Na **Figura 24** tem-se uma implementação da estratégia *JOINED*, em que a coluna discriminadora é *tipoPessoa*, o tipo de valor é *STRING*, registros de *PessoaFisica* e *PessoaJuridica* são identificados por *F* e *J*, respectivamente.

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="tipoPessoa", discriminatorType=DiscriminatorType.STRING)
public abstract class Pessoa {

       @Id
       private int id;
       private String nome;

       ...

}
```

```
@Entity
@DiscriminatorValue(value="F")
public class PessoaFisica extends
Pessoa {

       private int cpf;

       ...

}
```

```
@Entity
@DiscriminatorValue(value="J")
public class PessoaJuridica extends
Pessoa {

       private int cnpj;

       ...

}
```

Figura 24 – Exemplo de mapeamento JOINED

### 3.2.3 Mapeamento de relacionamentos

Os relacionamentos um para um, um para muitos, muitos para um e muitos para muitos são mapeados através das anotações *OneToOne*, *OneToMany*, *ManyToOne* e *ManyToMany*, respectivamente. Todo relacionamento possui um lado *proprietário*, o qual possui a chave estrangeira, e um lado *inverso*, caso seja bidirecional. Em relacionamentos bidirecionais, ambos os lados são anotados e o lado inverso deve utilizar a propriedade *mappedBy* para indicar o atributo do lado proprietário ao qual está relacionado. Na **Figura 25**, tem-se as classes *Funcionario* e *VagaEstacionamento* nas partes (a) e (b), respectivamente, as quais possuem um relacionamento um para um bidirecional. Para realizar o mapeamento, a anotação *OneToOne* foi colocada no atributo *vaga* da classe *Funcionario* e no atributo *proprietario* da classe *VagaEstacionamento*. Como a classe *Funcionario* é o lado proprietário, a propriedade *mappedBy* da anotação colocada em *VagaEstacionamento* foi informada com o

nome do atributo *vaga*. A parte (c) da **Figura 25** mostra as tabelas as quais as classes foram mapeadas.

a)

```
@Entity
public class Funcionario {
	@Id
	private int codigo;
	private String nome;
	@OneToOne
	private VagaEstacionamento vaga;
	...
}
```

b)

```
@Entity
public class VagaEstacionamento {
	@Id
	private int id;
	private String local;
	@OneToOne(mappedBy="vaga")
	private Funcionario proprietario;
	...
}
```

c)

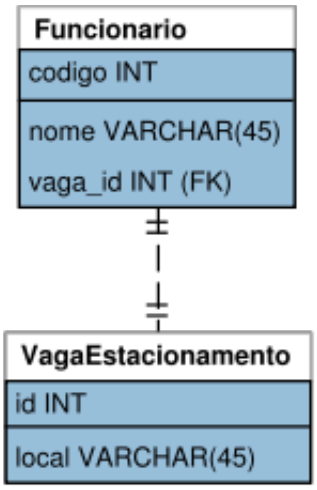


Figura 25 – Relacionamento um para um mapeado com a JPA

Em relacionamentos um para muitos, o lado um deve ser anotado com *OneToMany* e o lado muitos deve ser anotado com *ManyToOne*. Se um relacionamento deste tipo for bidirecional, o lado muitos deve ser obrigatoriamente o proprietário porque contém a chave estrangeira. Por outro lado, se o relacionamento for unidirecional e o lado proprietário o lado um, é necessária uma tabela associativa contendo as chaves primárias dos dois lados. A implementação deste tipo de relacionamento, porém, é opcional para os provedores, o que significa que para as aplicações serem portáveis, devem sempre implementar os relacionamentos um para muitos de forma bidirecional. Nas partes (a), (b) e (c) da **Figura 26** tem-se as classes *Funcionario*, *Dependente* e *Departamento*, respectivamente. *Funcionario* tem relacionamento um para muitos bidirecional com *Dependente* e *Departamento* possui relacionamento unidirecional com *Funcionario.* A parte (d) contém as tabelas mapeadas.

Em relacionamentos muitos para muitos utiliza-se a anotação *ManyToMany*. Esta anotação é aplicada nos dois lados caso o relacionamento seja bidirecional e qualquer um deles pode ser o lado proprietário. Uma tabela associativa é criada para realizar o relacionamento (DEMICHIEL; KEITH, 2006). As partes (a) e (b) da **Figura 27** contém as classes *Produto* e *Tag*, respectivamente, as quais possuem um relacionamento muitos para

muitos bidirecional em que o lado proprietário é representado pela classe *Produto*. A parte (c) contém as tabelas para as quais estas classes foram mapeadas e a tabela associativa *Produto_Tag*, a qual realiza o relacionamento.

a)

```
@Entity
public class Funcionario {
  @Id
  private int codigo;
  private String nome;
  @OneToMany(mappedBy="funcionario")
  private Set<Dependente> dependentes;
  ...
}
```

b)

```
@Entity
public class Dependente {
  @Id
  private int id;
  private String nome;
  @ManyToOne
  private Funcionario funcionario;
  ...
}
```

c)

```
@Entity
public class Departamento {
  @Id
  private int id;
  private String nome;
  @OneToMany
  private Set<Funcionario> funcionarios;
  ...
}
```

d)

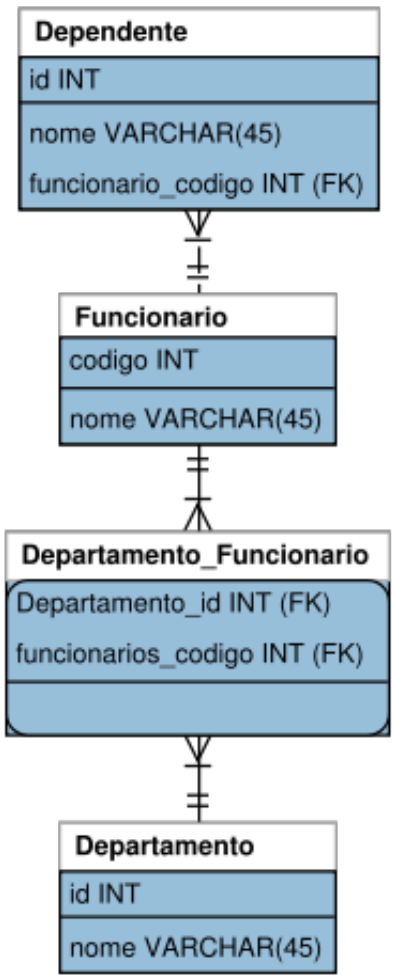


Figura 26 – Relacionamento um para muitos bidirecional e unidirecional

a)

```
@Entity
public class Produto {
  @Id
  private int id;
  private String descricao;
  @ManyToMany
  private Set<Tag> tags;
  ...
}
```

b)

```
@Entity
public class Tag {
  @Id
  private int id;
  private String descricao;
  @ManyToMany(mappedBy="tags")
  private Set<Produto> produtos;
  ...
}
```

c)

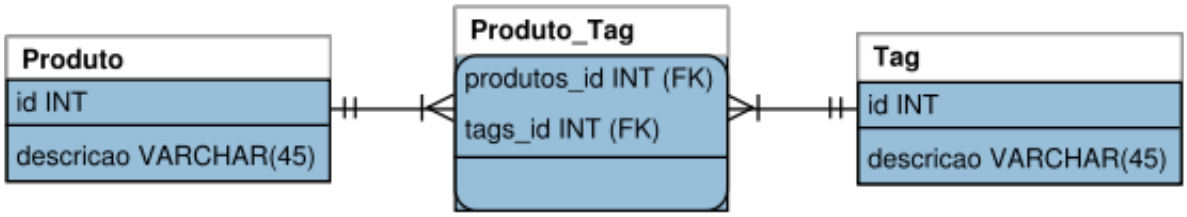


Figura 27 – Mapeamento muitos para muitos entre *Produto* e *Tag*

As operações de persistência em uma determinada entidade podem ser propagadas para as entidades à ela relacionadas. Quando uma entidade *a*, que está relacionada a uma entidade *b*, for persistida, *b* também pode ser persistido. É possível determinar quais operações são propagadas e quais não são. Esta configuração é feita através da propriedade *cascade*, a qual é do tipo enumerado *CascadeType*, e esta está presente em todas as anotações de relacionamento. *CascadeType* possui os valores *PERSIST*, *MERGE*, *REFRESH*, e *REMOVE*, os quais correspondem respectivamente às operações *persist*, *merge*, *refresh* e *remove* de um *EntityManager*, abordado no subcapítulo 3.2.4, e ainda o valor *ALL*, que representa todas as operações. Por exemplo, para indicar que as operações *persist* e *remove*, quando aplicadas a uma entidade *a*, devam ser propagadas a um entidade relacionada *b*, informa-se a propriedade *cascade* na anotação de relacionamento da seguinte forma: *@OneToMany(cascade={ CascadeType.PERSIST, CascadeType.REMOVE })*.

A JPA fornece um mecanismo de carregamento de atributos sob demanda, ou seja, um atributo de um objeto que acabara de ser recuperado do banco de dados será preenchido apenas quando o mesmo for acessado. Por padrão, todos os atributos simples de um objeto são carregados no instante de sua recuperação e todos os relacionamentos são carregados sob demanda. Configura-se um atributo para carregamento sob demanda utilizando a propriedade *fetch*, de tipo enumerado *FetchType*, das anotações. Todas as anotações de relacionamento possuem esta propriedade. Para atributos simples, pode-se utilizar a anotação *Basic*, a qual também possui esta propriedade. *FetchType* possui os valores *EAGER* e *LAZY*, os quais representam carregamento instantâneo e sob demanda, respectivamente. Por exemplo, para carregar sob demanda o atributo *foto* da classe *Cliente*, deve-se anotá-lo com *@Basic(fetch=FetchType.LAZY)* (DEMICHIEL; KEITH, 2006).

### 3.2.4 EntityManager

Um *EntityManager* é um objeto que fornece operações para interagir com um *Persistence Context*, o qual contém as entidades persistentes de uma aplicação, gerencia seus ciclos de vida e as sincronizam com o banco de dados. Com um *EntityManager* é possível persistir, remover e consultar entidades. O banco de dados e as entidades que um *EntityManager* pode gerenciar são definidas por uma *Persistence Unit*. As *Persistence Unit's* são definidas em um arquivo XML chamado *persistence.xml* (DEMICHIEL; KEITH, 2006).

#### 3.2.4.1 Ciclo de vida das entidades

Uma entidade é caracterizada como *new*, *managed*, *detached* e *removed*. A entidade é *new* quando não possui um valor de chave primária e não está associada a um *persistence context*; é *managed* quando possui um valor de chave primária e está associada a um *persistence context*; é *detached* quando possui um valor de chave primária porém não está associada a um *persistence context*; e é *removed* quando possui valor de chave primária, está associada a um *persistence context* e está marcada para remoção do banco de dados.

A operação *persist* do *EntityManager* persiste uma entidade e a torna *managed*. A operação *merge* atualiza o estado da entidade no banco de dados. A operação *refresh* atualiza a entidade com os dados do banco de dados, sobrescrevendo qualquer alteração feita a ela. A operação *remove* torna a entidade *removed*.

Todas estas operações devem ser executadas dentro de uma transação, que pode ser representada por uma *EntityTransaction* ou *UserTransaction*. A transação a ser utilizada é definida pelo tipo de *persistence context*, o qual pode ser *gerenciado pelo conteiner* (*container-managed*) e, portanto, utiliza a *UserTransaction*, ou *gerenciado pela aplicação* (*application-managed*), o qual utiliza a *EntityTransaction*. Este subcapítulo aborda apenas *persistence context gerenciado pela aplicação*.

Uma *EntityTransaction* é obtida através da operação *getTransaction* do *EntityManager* e possui as operações *begin*, *commit* e *rollback*, as quais são utilizadas para iniciar, confirmar e desfazer uma transação.

Um atributo de uma entidade marcado para carregamento sob demanda e que não está carregado for acessado depois de o *EntityManager* ser fechado, o resultado dependerá do provedor. No caso do Hibernate, é lançada a exceção *LazyInitializationException*. A implementação do carregamento sob demanda é opcional aos provedores.

Um *EntityManager* é obtido a partir de um *EntityManagerFactory*, o qual é uma fábrica de *EntityManager's*. Um *EntityManagerFactory* é construído a partir de uma *Persistence Unit* e pode ser utilizado concorrentemente, ao contrário do *EntityManager*. A criação de um *EntityManagerFactory* é mais dispendiosa do que a criação de um *EntityManager*, por isso, recomenda-se a criação de uma única instância de um *EntityManagerFactory*. Para criá-lo, utiliza-se a operação *createEntityManagerFactory* da classe *Persistence*. Esta classe é responsável por instanciar o *EntityManagerFactory* correspondente ao provedor utilizado (DEMICHIEL; KEITH, 2006). A **Figura 28** mostra um trecho de código em que é criado um *EntityManagerFactory* baseado na *Persistence Unit Contas* e um *EntityManager*.

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("Contas");
EntityManager manager = factory.createEntityManager();
```

Figura 28 – Obtenção de um *EntityManagerFactory* e um *EntityManager*

### 3.2.5 Linguagem de consulta de objetos

A JPA define uma linguagem de consulta de objetos chama *Java Persistence Query Language* (JPQL). Ela permite que desenvolvedores escrevam suas consultas independentemente de um banco de dados específico. Ela possui semelhanças com a linguagem SQL e as consultas são expressadas através de metadados, ou seja, das classes e seus relacionamentos. A JPQL possui as instruções *SELECT*, *UPDATE* e *DELETE*.

Em JPQL, tanto as entidades quanto seus atributos são referenciados por seus próprios nomes e não pelo nome das operações de acesso. Para acessar a entidade e seus atributos, é necessário utilizar um *apelido* (*alias*). A partir do apelido é possível acessar atributos das entidades manipuladas pela instrução. Por exemplo, para selecionar todas as entidades *Funcionario* utiliza-se a instrução *SELECT f FROM Funcionario f*. Para selecionar todos os nomes dos funcionários, utiliza-se *SELECT f.nome FROM Funcionario f*. Nestas intruções foi dado o apelido de *f* às entidades manipuladas.

Para acessar entidades relacionadas, deve-se utilizar a palavra *JOIN* seguida do atributo relacionado mais um apelido. Por exemplo, para selecionar os dependentes de todos os funcionários, pode-se utilizar a instrução *SELECT d FROM Funcionario f JOIN f.dependentes d*. Para filtrar resultados utiliza-se a palavra *WHERE*, de forma semelhante a

utilizada em SQL, porém, a JPQL fornece palavras específicas para a manipulação de entidades e coleções, como a MEMBER OF, que é utilizada para verificar se uma determinada entidade está contida em uma coleção. Por exemplo, para selecionar os dependentes do funcionário de código 1 utiliza-se a instrução *SELECT d FROM Funcionario f JOIN f.dependentes d WHERE f.codigo = 1*.

Existem dois tipos de consultas: estáticas e dinâmicas. As consultas estáticas, chamadas de *named queries*, são pré-definidas através de anotações ou arquivos XML e possuem um nome único. São úteis quando a mesma consulta é utilizada em vários locais da aplicação. As consultas dinâmicas são definidas na programação da aplicação. Uma diferença entre elas é que consultas estáticas são verificadas na inicialização da aplicação, enquanto consultas dinâmicas são verificadas no instante de sua execução. As consultas estáticas podem ser definidas através da anotação *NamedQuery*, a qual possui as propriedades *name* e *query*, que representam o nome da consulta e a instrução, respectivamente. Esta anotação deve ser colocada na classe que for primeiramente referenciada pela instrução de consulta.

As consultas são construídas através das operações *createQuery*, *createNamedQuery* e *createNativeQuery* do *EntityManager*, as quais produzem consultas dinâmicas, consultas estáticas e consultas SQL, respectivamente. Estas operações retornam um objeto *Query*, o qual possui as operações *getResultList*, *getSingleResult* e *executeUpdate*, as quais retornam uma lista de objetos, um objeto apenas ou o número de entidades afetadas pela instrução, respectivamente. A operação *executeUpdate* é utilizada para executar instruções *UPDATE* e *DELETE*. O retorno da operação *getResultList* depende da projeção feita pela instrução. Se a instrução projeta apenas uma entidade ou um atributo, a operação retorna uma lista contendo os objetos obtidos. Se a instrução projeta várias entidades ou atributos, a operação retorna uma lista de vetores, cada vetor possuindo os objetos projetados. Por exemplo, a instrução *SELECT f.nome, f.idade FROM Funcionario f* projeta o nome e idade de todos os funcionários, portanto, resulta em uma lista de vetores em que cada vetor possui duas posições contendo o nome e idade de um funcionário em particular, respectivamente.

As consultas podem ser executadas fora de uma transação, entretanto, as entidades obtidas serão *detached* (DEMICHIEL; KEITH, 2006).

# CAPÍTULO 4 – PERSISTÊNCIA COM ASPECTOS

Processos de desenvolvimento de *software* e linguagens de programação possuem um relacionamento de apoio mútuo. Processos lidam com a complexidade de um *software* dividindo-o em pequenos módulos relacionados. Linguagens de programação providenciam meios, como funções, procedimentos e objetos, para implementar estes módulos (KICZALES et al., 1997). Tais módulos são chamados *interesses* de um *software* e um objetivo antigo da engenharia de *software* é separar estes interesses do código-fonte (DIJKSTRA, 1976).

Existem basicamente dois tipos de interesses em um *software*: interesses-base e interesses transversais. Interesses-base estão relacionados às funcionalidades principais de um *software* e interesses transversais são requisitos não-funcionais, como segurança, desempenho, concorrência e persistência (CAMARGO, 2006). Embora estes interesses possam ser separados conceitualmente, a implementação acaba por entrelaçá-los. Os interesses transversais *entrecortam* (*crosscuts*) os interesses-base, resultando em entrelaçamento de código (*code tangling*) e espalhamento de código (*code scattering*). O interesse de *log*, por exemplo, pode ser implementado em uma classe através da inclusão de chamadas a um objeto de *log* em todas as suas operações. Desta forma, esta classe é responsável por implementar tanto o interesse-base quanto o interesse transversal, entrelaçando códigos de diferentes propósitos e violando o *princípio da responsabilidade única* ou *single responsibility principle* (SRP) (SRP, 2009), um princípio da POO que define que uma classe deve ter apenas uma responsabilidade. Além disso, como o interesse de *log* não pode ser modularizado com funções ou objetos, o código responsável pela implementação deste interesse fica espalhado pelas classes da aplicação.

Kiczales et al. (1997) propuseram a Programação Orientada a Aspectos (POA) e a linguagem AspectJ (ASPECTJ, 2009) para contribuir com a modularização dos interesses transversais.

## 4.1 Programação orientada a aspectos

A POA fornece novas abstrações para auxiliar na implementação dos interesses transversais. Na POA tem-se o conceito de *pontos de junção* (*join points*), que são pontos expostos pela aplicação durante sua execução, como chamada de métodos, instanciação de

objetos e lançamentos de exceções. Tais pontos podem ser identificados e categorizados através de construções chamadas de *pontos de corte* (*pointcuts*). Assim como uma consulta em SQL seleciona tuplas de um banco de dados através de um determinado critério, pontos de corte selecionam pontos de junção de uma aplicação. Pontos de corte são capazes de coletar o contexto dos pontos de junção; por exemplo, podem coletar os parâmetros passados ao método, coletar o objeto em execução, entre outros. É possível alterar o comportamento de pontos de junção, selecionados por pontos de corte, através de construções chamadas *adendos* (*advices*). Adendos podem adicionar comportamento antes, depois ou em volta dos pontos de junção. Um adendo em volta de um ponto de junção pode executá-lo zero ou mais vezes. A POA fornece uma construção para alterar a estrutura estática da aplicação, chamada de *declaração de intertipo* (*inter-type declarations*). Com ela é possível incluir atributos e métodos em classes, bem como alterar seus relacionamentos hierárquicos. A implementação de interesses transversais se dá através destas construções. Para agrupar construções relacionadas a um determinado interesse, a POA oferece uma construção chamada *aspecto* (*aspect*). Um aspecto é uma coleção de pontos de corte, adendos e declarações de intertipo correlacionados (LADDAD, 2009).

Processos de desenvolvimento decompõem um *software* em várias peças. Tais peças são implementadas utilizando procedimentos em linguagens procedurais ou classes em linguagens orientadas a objetos. Estas peças, por sua vez, são compostas para produzir o *software* como um todo. Como estas linguagens possuem apenas um mecanismo de composição, a saber, composição de procedimentos ou classes, interesses transversais são compostos manualmente pelos desenvolvedores, resultando em entrelaçamento e espalhamento de código. Para implementar peças transversais, a POA oferece aspectos, porém, como são estruturas diferentes de procedimentos e objetos, um novo mecanismo de composição é necessário. Tal mecanismo é chamado de *weaver* e é responsável por combinar aspectos com procedimentos ou objetos. *Weavers* podem atuar tanto em tempo de execução (*runtime weaving*) quanto em tempo de compilação (*compile-time weaving*) (KICZALES et al., 1997).

AspectJ é uma extensão orientada a aspectos de propósito geral para a linguagem Java e possui uma linguagem para definição de pontos de junção, pontos de corte, adendos e declarações de intertipo (KICZALES, 2001). Ela possui um compilador que realiza *compile-time weaving* e uma classe de tipo *ClassLoader* (CLASSLOADER, 2009) capaz de realizar

*runtime weaving*. *ClassLoader* é uma classe capaz de carregar e executar classes na plataforma Java e pode ser definida no instante da execução da *Java Virtual Machine* (JVM) através da definição da propriedade de sistema *java.system.class.loader*. Esta extensão possui um *script* de execução chamado *aj* que se encarrega de definir o *ClassLoader* para a realização do *runtime weaving*. Esta forma de *weaving* permite que aspectos sejam adicionados ao *software* sem a necessidade de compilação do mesmo. A forma como *weaving* é implementado é descrita em (HILSDALE; RUGUNIN, 2004). O compilador da linguagem é uma classe executável Java e esta extensão possui um *script* que a executa chamado de *ajc*.

Apesar de AspectJ possuir uma linguagem própria, permite que um aspecto possa ser desenvolvido como uma classe contendo anotações específicas da linguagem. Por exemplo, para que uma classe se torne um aspecto, utiliza-se a anotação *Aspect*. Isto permite compilar aspectos com um compilador Java normal, porém, *runtime weaving* deve ser utilizado. Mas este subcapítulo aborda apenas parte da linguagem da AspectJ.

Um aspecto *A* é criado dentro de um arquivo de mesmo nome, semelhante a classes Java. É declarado através da palavra *aspect* seguida de seu nome e um par de chaves. Dentro do par de chaves são declarados pontos de corte, adendos e declarações de intertipo. Exemplo: *aspect A {...}*.

Um ponto de corte é declarado através da palavra *pointcut* seguida de seu nome, uma lista opcional de parâmetros entre parênteses, dois pontos e definições de pontos de junção. Pontos de junção são definidos através de palavras como *call*, *execution*, *initialization*, entre outras. Cada palavra é sucedida de uma construção que expressa assinaturas de métodos e que pode conter caracteres coringas. Por exemplo, para expressar todas as chamadas a um método *m* da classe *A* utiliza-se *pointcut mCalled(): call(void A.m())*. Neste exemplo, o nome do ponto de corte é *mCalled*.

A palavra *call* expressa chamadas a métodos enquanto a palavra *execution* expressa execuções dos mesmos. Quando *call* é utilizada, os métodos selecionados são aqueles que são executados dentro das classes que os chamam, diferentemente de *execution*, em que os métodos selecionados estão dentro de suas próprias classes. Além disso, *call* não é capaz de selecionar pontos de junção em que um determinado método é invocado em uma subclasse através da palavra *super*, indicando que a implementação a ser executada deve ser a da superclasse (THE ASPECTJ..., 2009). Para exemplificar, na parte (a) da **Figura 29** tem-se a classe *A* com o método *m*, na parte (b) tem-se a classe *B*, a qual é subclasse de *A*, e que

reimplementa o método *m* definido por *A*. Na implementação de *m* em *B,* ocorre, na linha 5, uma chamada à implementação feita em *A*. A parte (d) possui a classe *D* com o método *main*, o qual instancia um objeto da classe *B* e invoca o método *m* do mesmo. O ponto de corte *pc1*, definido pelo aspecto *C* que está contido na parte (c) da figura, seleciona chamadas feitas ao método *m* da classe *A* e suas subclasses. Especificamente, este ponto de corte seleciona apenas a linha 5 da classe *D*. Já o ponto de corte *pc2* seleciona todas as execuções do método *m*. Desta forma, é como se ele selecionasse a linha 5 da classe *D* e a linha 5 da classe *B*, pois durante a execução do método *main* da classe *D*, o método *m* será executado duas vezes. O ponto de junção selecionado por *pc1* está dentro de *D*, enquanto os dois pontos de junção selecionados por *pc2* estão dentro de *B* e *A*, respectivamente.

a)

```
1. public class A {
2.
3.   public void m() {
4.     System.out.println("M()");
5.   }
6. }
```

b)

```
1. public class B extends A {
2.
3.   public void m() {
4.     System.out.println("B:M()");
5.     super.m();
6.   }
7.
8. }
```

c)

```
1. public aspect C {
2.
3.   pointcut pc1(): call(void A.m());
4.   pointcut pc2(): execution(void A.m());
5.
6. }
```

d)

```
1. public class D {
2.
3.   public static void main(...) {
4.     A a = new B();
5.     a.m();
6.   }
7.
8. }
```

Figura 29 – diferença entre os tipos de pontos de corte *call* e *execution*

O contexto de pontos de junção pode ser utilizado por adendos para realizar determinadas operações. Para a coleta de contexto de pontos de junção, utiliza-se as palavras *this*, *target* e *args*, as quais coletam o objeto em execução, o objeto cujo método é chamado e os parâmetros passados a este método, respectivamente. O ponto de corte *coletaContexto* da **Figura 30** seleciona toda chamada a um método *m*, com um parâmetro de tipo *int*, da classe *B* e que esteja dentro de *A.* Além disso, este ponto de corte coleta o objeto *a* em execução*,* o objeto *b* em que o método *m* será chamado e o parâmetro *parametro* passado.

```
pointcut coletaContexto(A a, B b, int parametro):
  this(a) && target(b) && call(void m(int)) && args(parametro);
```

Figura 30 – ponto de corte com coleta de contexto

Adendos são declarados através das palavras *before*, *after* e *around*, as quais indicam adendos que são executados antes, depois ou em volta de determinados pontos de junção, respectivamente. Adendos podem receber parâmetros contendo o contexto exposto pelos pontos de corte. Tais parâmetros são declarados como uma lista entre parênteses em frente à especificação do tipo de adendo. O tipo de adendo *after* possui duas construções que permitem definir se o adendo será executado depois que o ponto de junção retornar normalmente (*returning*) ou depois que o mesmo lançar uma exceção (*throwing*). Também é possível coletar o retorno do ponto de junção ou a exceção lançada. Desta forma pode-se especificar para qual exceção o adendo deve ser executado. Na declaração de um adendo, especifica-se os pontos de junção que serão afetados, tanto explicitamente quanto através de pontos de corte. Um adendo possui um corpo semelhante ao de um método. Dentro deste corpo, utiliza-se código Java e tem-se acesso a alguns objetos próprios para manipulação dentro do adendo, como *thisJoinPoint* e *thisJoinPointStaticPart*, os quais fornecem informações sobre o ponto de junção em questão. Na **Figura 31** tem-se o aspecto *IOLogger*, que possui o ponto de corte *handlesIO*, o qual seleciona chamadas a qualquer método, com qualquer quantidade de parâmetros, de qualquer classe pertencente ao pacote *java.io*. Este aspecto também define um adendo nas linhas 5 a 7, o qual exibirá uma mensagem depois que alguma chamada definida por *handlesIO* lançar a exceção *IOException*, do pacote *java.io*.

```
1. public aspect IOLogger {
2.
3.   pointcut handlesIO(): call(* java.io.*.*(..));
4.
5.   after() throwing(java.io.IOException e): handlesIO() {
6.     System.out.println("IO Error: " + e.getMessage());
7.   }
8. }
```

Figura 31 – aspecto que imprime toda exceção ocorrida em manipulações de entrada e saída

Declarações de intertipo são capazes de definir uma superclasse para um conjunto de classes e são especificadas através da construção *declare parents*. É possível inserir atributos e métodos em classes utilizando a mesma sintaxe da linguagem Java, com a exceção de que a classe afetada deve ser informada. Também é possível incluir anotações em classes, atributos

e métodos através das construções *declare @type, declare @field* e *declare @method*, respectivamente (THE ASPECTJ..., 2009). Como exemplo de utilização destas declarações, na **Figura 32**, tem-se o aspecto *Persistence*, o qual define uma interface *Persistente* e faz todas as classes pertencentes ao pacote *org.persistente* implementá-la. Na linha 6, o aspecto insere a anotação *Entity* em *Persistente*, nas linhas 8 a 11, o atributo *id* e o método *save* são inseridos em todas as classes que implementam a interface *Persistente*.

```
1. public aspect Persistence {
2.
3.   interface Persistente {}
4.
5.   declare parents: org.persistente.* implements Persistente;
6.   declare @type: Persistente: @Entity;
7.
8.   private int Persistente.id;
9.   private void Persistente.save() {
10.    ...
11.  }
```

Figura 32 – Exemplo de declarações de intertipo

## 4.2 Implementação da persistência com AspectJ

Rashid e Chitchyan (2003) propuseram um projeto inicial de *framework* de persistência orientado a aspectos utilizando AspectJ. Eles argumentaram que, para implementar a persistência, é necessário um meio de distinguir objetos persistentes de objetos transientes e, neste trabalho, implementaram uma classe chamada *PersistentRoot* e utilizaram um aspecto com declarações de intertipo para especificar quais classes são persistentes. A conexão com o banco de dados foi implementada através de dois pontos de corte que selecionam os pontos de junção em que a conexão deve ser estabelecida e fechada, respectivamente. Os objetos são convertidos em tuplas e inseridos no banco de dados no momento em que são instanciados. Um ponto de corte foi definido para selecionar os pontos de junção em que ocorre a criação de uma instância pertencente a qualquer uma das subclasses de *PersistentRoot*. Para a atualização do banco de dados, um ponto de corte foi definido para selecionar todas as execuções de métodos cujos nomes iniciam com *set*, os quais denotam alterações no estado de objetos. Entretanto, recuperação e remoção de objetos não puderam ser implementados na íntegra com aspectos. Os autores afirmam que o termo recuperação é o ato de obter informação de um local de armazenamento e a aplicação não pode ignorar o fato de que objetos persistentes são obtidos de fontes secundárias. Apesar

disso, os autores argumentam que aspectos podem ser utilizados para modularizar partes do código relacionado à recuperação. O projeto proposto define uma interface chamada *PersistentData*, a qual deve ser a superclasse de todas as classes referentes à recuperação de objetos. Tais classes devem possuir métodos que retornam uma lista de objetos e cujos nomes iniciem com *get*, os quais existem apenas para marcar determinados pontos em que a aplicação deseja obter objetos. Um ponto de corte é definido para selecionar todas as chamadas a estes métodos e um adendo *around* é responsável por identificar critérios de seleção, através do uso de reflexão, e executar a consulta correspondente.

Conforme mencionado anteriormente, a remoção de objetos tem de ser considerada explicitamente durante o desenvolvimento, pois dados são excluídos do banco de dados mediante pedido específico da aplicação. Além disso, não é possível identificar pontos em que objetos são excluídos em Java, devido ao *garbage collector*, um mecanismo da plataforma Java responsável por excluir da memória objetos inutilizados de forma automatizada. Mesmo em linguagens onde a remoção de objetos é explícita, como em C++, não é possível identificar quando a intenção da aplicação é remover do banco de dados ou apenas da memória. Por isso, a classe *PersistentRoot* possui o método *delete*, o qual é invocado pela aplicação e um adendo é encarregado de excluir o objeto correspondente.

Camargo et al. (2003) propuseram a implementação em aspectos do padrão de projeto Camada de Persistência (*Persistence Layer*) (YODER, 1998). A aplicação utilizada foi implementada utilizando a POO e, através de diretrizes definidas pelos autores, códigos relativos ao padrão de projeto foram modularizados em aspectos. O padrão Camada de Persistência é composto por dez subpadrões dos quais cinco foram utilizados pelos autores. Na implementação, atributos e métodos relativos ao padrão e pertencentes às classes persistentes foram retirados e implementados em um aspecto chamado *AspectStructure* através de declarações de intertipo. No método construtor de cada classe persistente, haviam códigos que especificavam informações concernentes ao mapeamento objeto-relacional, como a tabela a ser mapeada e a relação entre colunas e atributos. Como este código é específico de cada classe, foi criado um aspecto para cada uma delas, o qual encapsula tal código em um adendo *after* cujos pontos de junção afetados são momentos em que instâncias das classes são criadas. *AspectStrutucture* inclui métodos para inserção, remoção e consulta de objetos dentro das classes persistentes. Estes métodos, embora modularizados com este aspecto, são explicitamente chamados pela interface da aplicação. Estes métodos utilizam a classe

*TableManager*, a qual realiza a persistência dos objetos.

A interface da aplicação foi construída com *servlets*, as quais pertencem à plataforma Java EE e são utilizadas em aplicações baseadas na *Web*. Para a realização da conexão com o banco de dados, foi criado o aspecto *AspectConnection* que define um adendo *before* para a execução de tal tarefa antes da chamada ao método *init* das *servlets*.

Os autores propuseram uma forma de retirar códigos relativos ao padrão de persistência das classes de negócio utilizando AspectJ, com o objetivo de promover o reúso tanto do padrão quanto das classes. Porém, como algumas porções do código do padrão estão relacionadas a dados específicos do banco de dados, houve a necessidade de criar aspectos específicos para cada classe persistente, resultando em aumento significativo do número de componentes da aplicação.

# CAPÍTULO 5 – IMPLEMENTAÇÃO DA PERSISTÊNCIA UTILIZANDO JPA E ASPECTJ

Este trabalho tem como estudo de caso uma proposta de implementação do interesse de persistência de uma aplicação utilizando a JPA para a realização do mapeamento objeto-relacional e AspectJ para modularizar sua utilização, de tal forma que não existam códigos relativos à JPA nas classes da aplicação. A aplicação foi desenvolvida para *desktop* e possui cadastro de funcionários e dependentes. Foram utilizados alguns dos padrões de projeto propostos em (EVANS, 2004), os quais são: *Entity*, *Aggregate*, *Repository* e *Service*.

Em uma aplicação, existem objetos que possuem uma identidade que possui uma continuidade durante todos os estados do sistema. O estado de tais objetos mudam, porém sua identidade é mantida. Por exemplo, uma pessoa, mesmo que mude de nome, permanece a mesma pessoa, pois sua identidade não é baseada em seu nome. Tais objetos são classificados por (EVANS, 2004) como *entities*. Objetos que não possuem uma identidade são chamados de *value objects*. Por exemplo, objetos que representam endereços de correio podem ser considerados *value objects,* pois se o número ou o logradouro de um endereço mudar, ele não continua a ser o mesmo endereço de antes. Na aplicação de estudo de caso, cujo diagrama de classes simplificado está na **Figura 33**, são utilizados duas classes cujos objetos são considerados *entities*, as quais são *Funcionario* e *Dependente*, e há um relacionamento um para muitos entre estas classes. Ambos possuem um atributo *codigo*, o qual representa a identidade de seus objetos.

Uma rede de objetos compõe uma aplicação orientada a objetos. Todo objeto possui um ciclo de vida, que inclui a sua criação, computação, mudança de estados e destruição. Uma aplicação gerencia os ciclos de vida de objetos e seus relacionamentos, e sua complexidade é proporcional à quantidade de relacionamentos existentes entre estes objetos. Com o padrão *Aggregate* é possível diminuir esta complexidade através do agrupamento de objetos relacionados.

Um objeto *aggregate* delimita o acesso a seus objetos internos por parte de objetos externos. Todo *aggregate* possui uma raiz, a qual deve ser uma *entity*. Um *aggregate* pode conter *value objects* e *entities*, porém, estas apenas tem sentido quando estão dentro do *aggregate* e suas identidades são ditas locais. Objetos internos podem referenciar outros objetos internos dentro de um mesmo *aggregate,* porém objetos externos podem referenciar

apenas o objeto raiz.

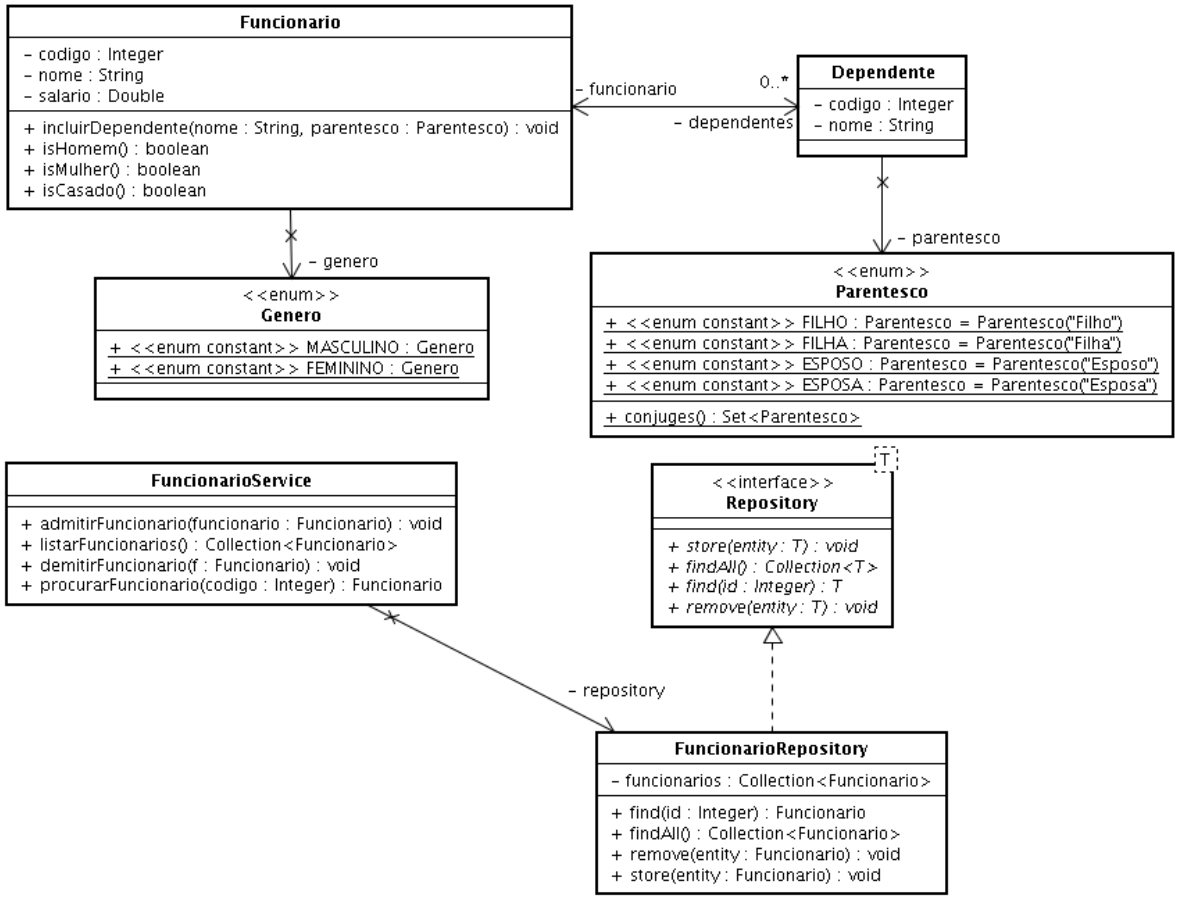


Figura 33 – Diagrama de classes simplificado da aplicação

Desta forma, um *aggregate* assegura integridade de dados e mantém as invariantes, as quais são regras que devem ser mantidas sempre que dados mudarem (AVRAM, 2007). Objetos externos não podem alterar objetos internos diretamente, apenas podem alterar a raiz ou pedir a ela que altere os internos. Na aplicação de estudo de caso, a classe *Funcionario* é um *aggregate* que possui um relacionamento um para muitos com a classe *Dependente*. Desta forma, *Funcionario* possui uma coleção de objetos da classe *Dependente* e delimita o acesso a esses objetos.

Objetos são criados para serem utilizados; para que um objeto utilize outro, deve possuir uma referência a ele. Para a obtenção de referências a objetos, pode-se utilizar o padrão *Repository*. *Repositories* são objetos que encapsulam toda a lógica necessária para a obtenção de objetos. A função de um *repository* é centralizar o acesso a *aggregates* através de métodos que tenham relação estrita com o domínio (EVANS, 2004). No estudo de caso, foi

desenvolvida uma interface Java chamada *Repository* que possui métodos para obtenção de *aggregates*. Todo *repository* possui uma ligação com um *aggregate* em particular, portanto, foi desenvolvida uma classe chamada *FuncionarioRepository*, a qual implementa a interface *Repository*. Os objetos da classe *Funcionario* são armazenados neste *repository*, o qual possui um conjunto subjacente de tipo *Set*.

Algumas operações requeridas pela aplicação não pertencem a nenhum objeto de domínio. Tais operações, se incluídas nestes objetos, adicionam mais responsabilidades a estes e isto viola o SRP. Geralmente, estas operações atuam em vários objetos e possivelmente de classes diferentes. Estas operações podem ser implementadas em objetos sem estado chamados de *Services*. Um *service* correlaciona e oferece estas operações. Na aplicação de estudo de caso, foi criado a classe *FuncionarioService*, a qual provê operações para admissão, demissão e listagem de funcionários.

Para realizar o mapeamento objeto-relacional, foi criado um aspecto para cada classe persistente, os quais são *DependenteJPA* e *FuncionarioJPA*. Cada aspecto insere as anotações necessárias para o mapeamento. Na **Figura 34**, tem-se o aspecto *DependenteJPA*, que insere as anotações *Entity, Id, GeneratedValue* e *Enumerated*, além de incluir um construtor sem argumentos, o qual é exigência da JPA. A anotação *Entity* indica *Dependente* como persistente, as anotações *Id* e *GeneratedValue* no atributo *codigo* indicam chave primária autoincrementada e a anotação *Enumerated* no atributo *parentesco* indica que a forma de persistência do mesmo é através do armazenamento dos nomes das constantes do tipo enumerado *Parentesco,* o qual pode ser visualizado na **Figura 33**.

```
1. package infrastructure;
2.
3. import javax.persistence.*;
4. import domain.model.*;
5.
6. public aspect DependenteJPA {
7.
8.   declare @type: Dependente: @Entity;
9.   declare @field: Integer Dependente.codigo: @Id;
10.  declare @field: Integer Dependente.codigo:
                    @GeneratedValue(strategy=GenerationType.IDENTITY);
11.  declare @field: Parentesco Dependente.parentesco:
                    @Enumerated(EnumType.STRING);
12.
13.  public Dependente.new() { super(); }
14.}
```

Figura 34 – aspecto *DependenteJPA* que realiza o ORM na classe *Dependente*

O armazenamento de objetos da classe *Funcionario* por *FuncionarioRepository* é

feito através de um atributo de tipo *Set*. Desta forma, a aplicação é capaz de manter referências somente em memória. Para realizar a persistência dos objetos em um banco de dados relacional, foi criado um aspecto chamado *RepositoryJPA*. Este aspecto possui um *EntityManagerFactory* e um *EntityManager*. Um ponto de corte foi criado para cada seleção de pontos de junção relativos a um determinado método da interface *Repository*. Adendos foram construídos para executarem as operações do *EntityManager* relativas aos métodos. Por exemplo, conforme **Figura 35**, o ponto de corte *salvar* seleciona todas as execuções do método *store* da interface *Repository*, bem como coleta o objeto a ser persistido, e um adendo *around* executa o método *persist* do *EntityManager*.

```
pointcut salvar(Object o): execution(void Repository.store(Object)) && args(o);

void around(Object o): salvar(o) {
  em.persist(o);
}
```

Figura 35 – ponto de corte *salvar* e adendo *around*

O método *findAll*, de *Repository*, tem como objetivo retornar todos os objetos de um determinado *aggregate*. Um adendo *around* é executado em volta das execuções deste método, selecionadas pelo ponto de corte *findAll*, e executa uma consulta JPQL para recuperar todos os objetos desejados do banco de dados. Foram utilizados recursos da plataforma Java para obter o nome da classe persistente relacionada ao *repository* coletado pelo ponto de corte. Na **Figura 36**, tem-se o ponto de corte *findAll* e o adendo *around*. O método *getRepositoryClass* pertence ao aspecto e é responsável por recuperar a classe persistente através do *repository* coletado. O adendo, por fim, constrói uma consulta JPQL com o nome da classe persistente e a executa pelo *EntityManager*.

```
pointcut findAll(Repository r): target(r) && execution(Collection<*> findAll());

Collection around(Repository r): findAll(r) {
  Class clazz = getRepositoryClass(r);

  return em.createQuery("SELECT o FROM " + clazz.getName() + " o").getResultList();
}
```

Figura 36 – ponto de corte *findAll* e adendo *around*

Como as operações *persist* e *remove* do *EntityManager* requerem uma transação, foi necessário desenvolver um ponto de corte que seleciona todos os pontos de junção em que são executadas tais operações. Os potenciais pontos de junção detectados foram todas as

chamadas aos métodos *store* e *remove* de *Repository* e todas as chamadas aos métodos de *FuncionarioService*. Como objetos *Service* são frequentemente criados para realizar operações com vários *aggregates* e isto implica em realizar várias chamadas a métodos de *repositories*, foram escolhidas todas as chamadas de métodos de objetos *Service* ao invés de métodos de *repositories*. Desta forma, se duas chamadas ao método *store* forem feitas em um método de um objeto *service*, por exemplo, ambas estarão dentro de uma única transação ao invés de serem criadas duas transações.

Na **Figura 37** tem-se o ponto de corte *transaction*, o qual seleciona todas as chamadas a qualquer método da classe *FuncionarioService*. A palavra *withincode* foi utilizada juntamente com o operador *!* para impedir que uma transação fosse iniciada dentro de outra, pois métodos de objetos *Service* podem utilizar de seus outros métodos para realizar uma determinada função. Pode-se ler este ponto de corte da seguinte forma: selecione todas as chamadas feitas a qualquer um dos métodos da classe *FuncionarioService* desde que tais chamadas não estejam dentro destes métodos. Desta forma, se um método *a* possui uma chamada a um método *b*, ambos pertencentes a objetos *services*, apenas uma transação será iniciada.

```
pointcut transaction(): call(void FuncionarioService.*(..)) &&
  !withincode(void FuncionarioService.*(..));

before(): transaction() {
  tx = em.getTransaction();
  tx.begin();
}

after() returning(): transaction() {
  tx.commit();
  em.clear();
}

after() throwing(): transaction() {
  tx.rollback();
  em.clear();
}
```

Figura 37 – ponto de corte *transaction* e adendos *before* e *after*

Nesta figura, tem-se três adendos. O adendo *before* é encarregado de iniciar uma transação e isto é feito através do método *begin* do objeto *EntityTransaction*, o qual é recuperado através do método *getTransaction* do *EntityManager*. A variável *tx* pertence ao aspecto *RepositoryJPA* e mantém a transação corrente. O adendo *after returning* finaliza a transação depois da execução normal do método e o adendo *after throwing* a desfaz caso o

método lance alguma exceção.

## 5.1 Considerações sobre a implementação

Segundo Rashid e Chitchyan (2003), uma aplicação não pode ignorar o fato de que objetos persistentes são obtidos de fontes externas. O fato de que bancos de dados relacionais utilizam uma linguagem declarativa, a SQL, que baseia-se em predicados e condições para a obtenção de dados, contribui para a disparidade entre relacional e orientado a objeto, uma vez que, no último, recupera-se dados através de travessias de objetos.

Os autores definem que a remoção de objetos é requisitada explicitamente pela aplicação e, por isso, não é possível deixar de considerá-la durante o desenvolvimento. Levando estas questões em consideração, neste trabalho foi desenvolvido um meio de centralizar o acesso a dados através do padrão *Repository*. Um *repository* é criado para que, desta forma, toda requisição da aplicação relacionada ao armazenamento e recuperação de objetos seja enviada a ele. Consultas mais complexas podem ser criadas através de objetos que encapsulam critérios de seleção. Tais critérios podem ser aplicados a todos os objetos persistentes com o objetivo de coletar aqueles que os satisfizeram. O aspecto *RepositoryJPA* pode conter um adendo *around* capaz de interpretar tais objetos e transcrevê-los em consultas JPQL. Isto pode ser feito através da correlação entre nomes das classes de critério e *named queries*, do JPA. Por exemplo, pode existir a classe *ProcuraFuncionarioPorNome* e uma *named query* correspondente de mesmo nome. Os parâmetros podem ser os atributos da classe de critério e estes podem ser obtidos através de recursos de reflexão da linguagem Java. Entretanto, a utilização dos recursos de reflexão pode diminuir demasiadamente a performance da aplicação (FORMAN et al., 2004).

A JPA fornece a opção de carregamento de atributos sob demanda. Isto é útil quando um objeto que possui uma coleção de objetos ou um atributo de conteúdo binário, como, por exemplo, uma imagem, é recuperado e tais atributos nem sempre são desejados para processamento. Entretanto, para que um objeto seja carregado sob demanda, o *EntityManager* deve estar aberto no momento de sua inicialização. Como não é possível identificar se um objeto ainda não foi carregado, optou-se, neste trabalho, por deixar o *EntityManager* aberto durante toda a execução da aplicação. Porém, à medida em que objetos são manipulados pelo *EntityManager*, ocorre um aumento na utilização da memória principal, pois tais objetos

ficam armazenados no *PersistenceContext*. A fim de amenizar tal situação, o *PersistenceContext* é limpo a cada transação, através do método *clear* do *EntityManager*, conforme mostrado na **Figura 37**. Entretanto, se durante a execução da aplicação, nenhuma transação for executada, não haverá a limpeza do *PersistenceContext* e é possível que todos objetos da aplicação fiquem em memória principal. Uma possível solução é não utilizar o carregamento sob demanda, porém, a aplicação corre o risco de subutilizar dados recuperados do banco de dados.

Apesar destas implicações, foi possível implementar o interesse de persistência desta aplicação com JPA e AspectJ, de forma que todo código relativo ao *framework* fosse retirado das classes da aplicação e modularizado em aspectos. Isto contribui para a reutilização das classes e dos aspectos de persistência, exceto aqueles responsáveis por incluir as anotações relativas ao mapeamento objeto-relacional por possuírem códigos específicos relacionados às classes.

# CAPÍTULO 6 – CONCLUSÃO

O mapeamento objeto-relacional pode ser feito com o auxílio de *frameworks*, como a JPA. Entretanto, códigos relativos ao *framework* ficam misturados e espalhados por todo o código da aplicação, devido à falta de abstrações adequadas por parte da POO para a modularização destes códigos. Neste contexto, surgiu a POA e AspectJ para suprir a falta de tais abstrações.

Este trabalho propõe uma possível implementação orientada a aspectos para o interesse de persistência implementado com JPA. A implementação proposta mostrou-se eficaz para a modularização da utilização do *framework*; porém, foram encontrados alguns problemas que afetam a performance da aplicação. Tais problemas são a necessidade de utilização de reflexão para a realização de consultas no banco de dados e a necessidade de manter o *EntityManager* aberto, devido ao carregamento sob demanda, ocasionando ocupação demasiada da memória principal. Entretanto, as classes da aplicação ficam desprovidas das anotações da JPA e de outros códigos relativos ao *framework*, evitando o entrelaçamento e espalhamento de código. Desta forma, as classes da aplicação e o aspecto de persistência se tornam mais reutilizáveis e com manutenção facilitada.

Espera-se que esta proposta contribua para a utilização de *frameworks* de persistência juntamente com a programação orientada a aspectos, e que colabore para a reutilização de códigos de interesses-base e de interesses transversais.

# REFERÊNCIAS

ALUR, D.; CRUPI, J.; MALKS, D. **Core J2EE patterns**: best practices and design strategies. Upper Sadle River, Estados Unidos da América: Prentice Hall, 2003.

AMBLER, S. W. **The object-relational impedance mismatch**. Disponível em: <http://www.agiledata.org/essays/impedanceMismatch.html>. Acesso em: 6 nov. 2009.

ASPECTJ. Disponível em: <http://www.eclipse.org/aspectj>. Acesso em: 17 nov. 2009.

AVRAM, A. **Domain-driven design quickly**. [S.l.]: Lulu.com, 2007.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML:** Guia do Usuário. Rio de Janeiro: Elsevier, 2007.

BRAY, T.; PAOLI, J.; SPERBERG-MCQUEEN, C. M.; MALER, E.; YERGEAU, F. **Extensible markup language (XML) 1.0**. 5ed. [S.l.: s.n.], 2008. Disponível em: <http://www.w3.org/TR/2008/REC-xml-20081126/>. Acesso em: 13 nov. 2009.

CACHÉ. Disponível em: <http://www.intersystems.com/cache/>. Acesso em: 24 out. 2009.

CAMARGO, V. V. **Frameworks transversais**: definições, classificações, arquitetura e utilização em um processo de desenvolvimento de software. 2006. 256 p. Tese (Doutorado em Ciência da Computação e Matemática Computacional) – Universidade de São Paulo, São Carlos, 2006.

CAMARGO, V. V.; RAMOS, R. A.; PENTEADO, R. A. D.; MASIERO, P. C. Projeto baseado em aspectos do padrão camada de persistência. In: Simpósio Brasileiro de Engenharia de Software – SBES, 2003, Manaus-AM.

CHEN, P. P. The entity-relationship model – toward a unified view of data. **ACM transactions on database systems (TODS)**, [s.l.], v. 1, n. 1, p. 9-36, mar. 1976.

CLASSLOADER. Disponível em: <http://java.sun.com/javase/6/docs/api/java/lang/ClassLoader.html>. Acesso em: 17 nov. 2009.

CODD, E. F. A relational model of data for large shared data banks. **Communcations of the ACM**, [S.l.], v. 16, n. 6, p. 377-387, jun. 1970.

DEITEL, H. M.; DEITEL, P. J. **Java:** Como Programar. Porto Alegre: Bookman, 2003.

DEMICHIEL, L.; KEITH, M. (Líderes). **JSR 220: Enterprise JavaBeans, version 3.0** - Java Persistence API. Final Release. California, Estados Unidos da América: [s.n.], 2006. Disponível em: <http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html>. Acesso em: 24 mai. 2009.

DIJKSTRA, E. W. **A discipline of programming**. Englewood Cliffs, Estados Unidos da América: Prentice-Hall, 1976.

ECLIPSELINK JPA. Disponível em: <http://www.eclipse.org/eclipselink/jpa.php>. Acesso em: 15 nov. 2009.

ELMASRI, R.; NAVATHE, S. B. **Sistemas de bancos de dados**. 4ed. São Paulo: Pearson Addison Wesley, 2005.

EVANS, E. **Domain-driven design**: tackling complexity in the heart of software. Boston, Estados Unidos da América: Addison-Wesley, 2004.

FAKURA, D. **Object-oriented Software Design and Construction with Java**. Upper Saddle River, Estados Unidos da América: Prentice-Hall, 2000.

FEHILY, C. **Python**. Berkeley, Estados Unidos da América: Peachpit, 2002.

FLANAGAN, D. **Java foundation classes in a nutshell**. [S.l.]: O'Reilly, 1999.

FLANAGAN, D.; MATSUMOTO, Y. **Ruby Programming Language**. Sebastopol, Estados Unidos da América: O'Reilly, 2008.

FORMAN, R.; FORMAN, N.; IBM, J. V. **Java reflection in action**. [S.l.]: Manning Publications, 2004.

FOWLER, M. **Patterns of enterprise application architecture**. Boston, Estados Unidos da América: Addison-Wesley, 2003.

GAMMA, E.; et al. **Design patterns:** elements of reusable object-oriented software. Reading, Estados Unidos da América: Addison Wesley, 1995.

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G. Annotations. In: _____. **The Java Language Specification**. 3. ed. Upper Saddle River, Estados Unidos da América: Addison-Wesley, 2005. p. 281-286.

HIBERNATE. Disponível em: <http://www.hibernate.org/>. Acesso em: 15 nov. 2009.

HILSDALE, E.; HUGUNIN, J. In: Proceedings of the 3rd international conference on Aspect-oriented software development. n. 3. Lancaster, Reino Unido: ACM, 2004. p. 26-35.

HORSTMAN, C. S.; CORNELL, G. **Core Java 2:** Fundamentos. Tradução: João Eduardo Nóbrega Tortello. São Paulo: Makron Books, 2001.

JACOBSON, I. **Object-Oriented Software Engineering:** A Use Case Driven Approach. Nova York, Estados Unidos da América: ACM Press, 1992.

JAVA PLATFORM SE 6. Documentação da plataforma Java SE. Disponível em:<http://java.sun.com/javase/6/docs/api/>. Acesso em: 18 out. 2009.

JIA, X. **Object-oriented Software Development in Java:** principles, patterns and frameworks. Reading, Estados Unidos da América: Addison Wesley, 2000.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, G. Getting started with AspectJ. In: Communications of ACM. n. 10. New York, Estados Unidos da América: ACM, 2001. vol. 44, p. 59-65.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.; IRWIN, J. Aspect oriented programming. In: Proceedings of 11 ECOOP. p. 220-242, 1997.

LADDAD, R. **AspectJ in action**: enterprise AOP with Spring. 2. ed. [S.l.]: Manning Publications, 2009.

MEYER, B. **Object-Oriented Software Construction**. 2nd ed. Hertfordshire, Inglaterra: Prentice Hall International, 1997.

MICROSOFT SQL Server 2008. Disponível em: <http://www.microsoft.com/brasil/servidores/sql/default.mspx>. Acesso em: 19 out. 2009.

MYSQL Developer Zone. Disponível em: <http://dev.mysql.com/>. Acesso em: 19 out. 2009.

OBJECTSTORE. Disponível em: <http://www.objectstore.com/>. Acesso em: 24 out. 2009.

OMG - OBJECT MANAGEMENT GROUP. **Unified Modeling Language, Infrastructure: version 2.2**. [S.l. : s.n.], 2009. Disponível em: <http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/>. Acesso em: 09 out. 2009.

OPENJPA. Disponível em: <http://openjpa.apache.org/>. Acesso em: 15 nov. 2009.

ORACLE. Disponível em: <http://www.oracle.com/>. Acesso em: 19 out. 2009.

POSTGRESQL. Disponível em: <http://www.postgresql.org/>. Acesso em: 24 out. 2009.

PREVAYLER. [S.l.: s.n., s.d.]. Disponível em: <http://www.prevayler.org/>. Acesso em: 19 out. 2009.

RASHID, A.; CHITCHYAN, R. **Persistence as an aspect**. In: PROCEDINGS OF 2ND INTERNATIONAL CONFERENCE ON ASPECT ORIENTED SOFTWARE DEVELOPMENT – AOSD, 2, 2003. Boston, Estados Unidos da América: ACM, 2003. 128p. p. 120-129.

RUMBAUGH, J.; BLAHA, M. **Modelagem e Projetos Baseados em Objetos com UML 2**. Rio de Janeiro: Elsevier, 2006.

RUMBAUGH, J.; BLAHA, M.; PREMERLANI, W. **Modelagem e Projetos Baseados em Objetos**. Rio de Janeiro: Elsevier, 1993.

SCHACH, S. **Object-oriented and Classical Software Engineering**. Boston, Estados Unidos da América: McGraw-Hill, 2005.

SEBESTA, R. W. **Concepts of programming languages**. 9 ed. [S.l.]: Addison-Wesley, 2009.

SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Sistema de Banco de Dados**. São Paulo: Makron Books, 1999.

SOARES, S.; LAUREANO, E.; BORBA, P. Implementing distribution and persistence aspects with AspectJ. In: Proceedings of the 17th ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 17, Washington, Estados Unidos da América: ACM, 2002. p. 174-190.

SOMMERVILLE, I. **Engenharia de Software**. 8 ed. São Paulo: Addison Wesley, 2007.

SRP: the single responsibility principle. Disponível em: <http://www.objectmentor.com/resources/articles/srp.pdf >. Acesso em: 17 nov. 2009.

STROUSTRUP, B. **C++ Programming Language**. Reading, Estados Unidos da América: Addison Wesley, 1997.

SUBSONIC. Disponível em: <http://subsonicproject.com/>. Acesso em: 21 nov. 2009.

SUN MICROSYSTEMS. **Developer Resources for Java Technology**. [S.l.: s.n., s.d.]. Disponível em: <http://java.sun.com>. Acesso em: 14 out. 2009.

THE ASPECTJ Programming Guide. Disponível em: <http://www.eclipse.org/aspectj/doc/released/progguide/index.html>. Acesso em: 18 nov. 2009.

THE JAVA Community Process (SM) Program. Disponível em: <http://jcp.org/>. Acesso em: 15 nov. 2009.

TUPLE. In: HOWE, Dennis. **The free on-line dictionary of computing**. Disponível em: <http://dictionary.reference.com/browse/tuple>. Acesso em: 22 out. 2009.

VERSANT. Disponível em: <http://www.versant.com/>. Acesso em: 24 out. 2009.

WITTHAWASKUL, W.; JOHNSON, R. **Specifying Persistence in Platform Independent Models**. In: PROCEDINGS OF THE WORKSHOP IN SOFTWARE MODEL ENGINEERING HELD IN CONJUNCTION WITH THE UML 2003 – THE UNIFIED MODELING LANGUAGE, 6, 2003. **Model Languages and Applications**. San Francisco, Estados Unidos da América: 2003.

YODER, J. W.; JOHNSON, R.; WILSON, Q. Connection business objects to relational databases. In: Conferece on Patterns Languages of Programs (PloP), 5th, 1998, Monticello, Estados Unidos da América.

YOURDON, E.; COAD, P. **Object-oriented analysis**. Englewood Cliffs, Estados Unidos da América: Yourdon Press, 1990.