4 Associações 2
Objetivos 3
Inicializações em Associações 4
Multiplicidade de Associações 4
Multiplicidade Um para Um 5
Relacionamento Bidirecional 8
Multiplicidade Um para Muitos 10
Relacionamento Unidirecional 10
@javax.persistence.OneToMany e JoinTable 10
Relacionamento Bidirecional 14
Multiplicidade Muitos para Um 15
Relacionamento Unidirecional 15
Relacionamento Bidirecional 16
Multiplicidade Muitos para Muitos 19
Relacionamento Bidirecional 19
Relacionamento Unidirecional 21
Mapeando Relacionamentos Baseados em Coleções 23
Relacionamento Baseado em Lista Ordenada 23
Relacionamento Baseado em Mapas 24
Continuação do Exemplo do Curso 26
Exercícios 27

Associações

Objetivos

- Compreender os mecanismos de associações;
- Exercitar os diferentes mapeamentos entre as associações;
- Tratar questões de cascateamento de operações em associações;
- Tratar questões de inicialização de associações.

Inicializações em Associações

Direcionamento

A direção de um relacionamento pode ser bidirecional ou unidirecional.

Em um relacionamento bidirecional, cada entity bean tem um campo do relacionamento que consulta o outro bean. Através do campo de relacionamento, o código de um entity bean pode alcançar seu objeto relacionado. Por exemplo, se **`Curso`** souber que instâncias de **`Turma`** ele possui e se **`Turma`** souber a qual **`Curso`** ela pertence, eles possuem um relacionamento do tipo bidirecional.

Em um relacionamento unidirecional, somente um dos entity beans tem um campo de relacionamento que consulta o outro. Por exemplo, **`Instrutor`** teria um campo de relacionamento que identificasse **`UsuarioAcesso`** ou **`Endereco`**, mas **`UsuarioAcesso`** ou **`Endereco`** não teria um campo de relacionamento para **`Instrutor`**. Ou seja, **`Instrutor`** sabe sobre **`Endereco`**, mas **`Endereco`** não sabe que instâncias de **`Instrutor`** lhe referenciam.

As queries JPQL são realizadas com base nos relacionamentos. O sentido de um relacionamento determina se uma query pode acessar a partir de um bean, o outro. Por exemplo, uma query pode navegar de **`Instrutor`** para **`Endereco`**, mas não pode navegar no sentido oposto. Para **`Curso`** e **`Turma`**, uma query poderia navegar em ambos os sentidos, desde que estes dois beans tenham um relacionamento bidirecional.

Multiplicidade de Associações

Existem quatro tipos de multiplicidades:

One-to-one: Cada instância de um entity bean é relacionada a uma única outra instância de um entity bean.

One-to-many: Uma instância de um entity bean pode ser relacionada as outras múltiplas instâncias de um entity bean. Por exemplo, um curso pode ter várias turmas abertas. Neste caso, o relacionamento seria de um curso para muitas turmas.

Many-to-one: Múltiplas instâncias de um entity bean serão relacionadas a uma única instância de outro entity bean. Esta multiplicidade é o oposto a one-to-many.

Many-to-many: Múltiplas instâncias de um entity bean serão relacionadas a outras múltiplas instâncias de um entity bean. Por exemplo, uma empresa de treinamento possui vários cursos, e estes possuem vários alunos. Conseqüentemente, Curso e Aluno teriam um relacionamento many-to-many.

Multiplicidade Um para Um

@javax.persistence.OneToOne e JoinColumn

Ao realizar um mapeamento OneToOne, por exemplo, é necessário indicar a qual coluna da tabela do banco de dados que o atributo pertence/referencia. No exemplo abaixo a anotação `@javax.persistence.JoinColumn` indica qual coluna da tabela de instrutores irá referenciar a pk da tabela de endereços. Por default, será referenciada a pk de enderecos. Para alterar isto pode-se usar o atributo `referencedColumnName`. O atributo `referencedColumnName` só poderá referenciar colunas "`unique`". Para chaves compostas deve-se utilizar `JoinColumns`.

OneToOne é um relacionamento onde uma entidade referencia somente uma outra. Nos exemplos que temos mostrado pode-se ver que a entidade Instrutor irá referenciar a entidade Endereço. Tipicamente, cada pessoa (instrutor) possuirá o seu endereço associado a ele. O Endereço não possui referência para o instrutor, o que faz deste relacionamento um relacionamento 1:1 ou OneToOne unidirecional. Abaixo, mostramos um exemplo completo deste relacionamento. Observe que na anotação OneToOne é adicionada à informação de que deve ser persistido o objeto referenciado pelo instrutor quando um instrutor for persistido. Isto permitirá que o usuário chame o método persist para persistir um instrutor e o objeto (Endereco) sejam, de uma só vez, persistidos.

Abaixo, o código completo para isto.

```
import java.io.Serializable;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "Instrutores")
public class Instrutor implements Serializable {

    @Id
    @Column(name = "matricula", nullable = false)
    private Integer matricula;

    @Column(name = "nome")
    private String nome;

    @Column(name = "email")
    private String email;

    @Column(name = "telefone")
    private String telefone;

    @JoinColumn(name = "endereco", referencedColumnName = "codigo")
    @OneToOne(cascade={CascadeType.PERSIST})
    private Endereco endereco;

    public Instrutor(Integer matricula) {
        this.matricula = matricula;
    }
    ...
}
```

Código 4-1: Relacionamento um pra um

Veja agora o código para testarmos à associação na camada de persistência:

```
   public class TestePersistenceInstrutorEndereco {

       public static void main(String args[]) {
           Endereco endereco = new Endereco();
           endereco.setRua( "São Francisco da Califórnia" );
           endereco.setNumero( 23 );
           endereco.setBairro( "Higienópolis" );
           endereco.setCidade( "Porto Alegre" );
           endereco.setEstado( "Rio Grande do Sul" );

           Instrutor inst = new Instrutor();
           inst.setNome("Fábio Paulo Basso");
           inst.setEmail("fabiopbasso@gmail.com");
           inst.setMatricula( 2089 );
           inst.setTelefone("51 9735 3454");

           inst.setEndereco( endereco );

       EntityManager em =
EntityManagerLoader.getInstance().getEntityManager();
       /* SALVAR */
       try {
          em.getTransaction().begin();
          em.persist( inst );
          em.getTransaction().commit();
          System.out.println("Objetos salvos com sucesso!");
       } catch (RuntimeException e) {
          e.printStackTrace();
          em.getTransaction().rollback();
       }finally{
          em.close();
       }
       }
   }
```

Código 4-2: Código cliente

Relacionamento Bidirecional

O relacionamento um-pra-um ocorre quando duas entidades possuem uma única referência entre elas em um mesmo contexto. Por exemplo, podemos para cada cliente (classe Cliente) buscar o seu cartão de crédito (classe CartaoCredito). Também podemos a partir do número do cartão de crédito informado pelo cliente, buscar seus dados pessoais especificados na classe Cliente. Assim, temos um relacionamento bidirecional, em que ambos os lados da associação permitem buscarmos a associação via as propriedades especificadas nas classes.

Estrutura da Base de Dados Relacional

Para representar esse tipo de relacionamento, a tabela CLIENTE deve conter uma coluna que serve como uma chave estrangeira para a tabela CARTAO_CREDITO:

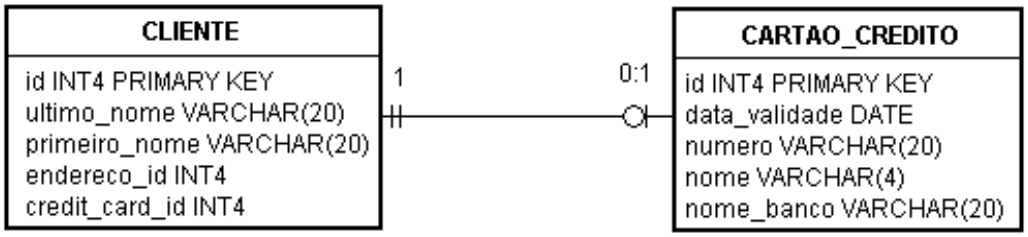


Figura 4.1 Relacionamento um-pra-um bidirecional na Base de Dados

Modelo de Programação

Para modelar o relacionamento entre Cliente e CartaoCredito, a classe Cliente deve conter um atributo de relacionamento do tipo CartaoCredito, e esta deve conter um atributo Cliente. Para ligar em código as duas propriedades na associação devemos utilizar em um dos lados, na anotação OneToOne, a propriedade **mappedBy**. Nesta propriedade devemos colocar o nome da propriedade do tipo corrente especificado na classe associada. Por exemplo, se colocarmos o mappedBy no OneToOne de Cliente, então devemos colocar como valor o nome da propriedade do tipo Cliente especificada em CartaoCredito. Do contrário, especificamos em CartaoCredito o nome da propriedade deste tipo definido Cliente

```
   @Entity
   public class Cliente implements java.io.Serializable {

      @OneToOne
      private CartaoCredito cartCred;

      public CartaoCredito getCartCred() { return cartCred; }
      public void setCartCred(CartaoCredito c) { this.cartCred = c;
}
   }
```

Código 4-3: Entidade Cliente em associação com CartaoCredito

Do outro lado da associação, na classe CartaoCredito deve-se adicionar também um atributo de relacionamento para referenciar o Cliente:

```
public class CartaoCredito implements java.io.Serializable {

  @OneToOne(mappedBy="cartCred")
  private Cliente cliente;
}
```

Código 4-4: Entidade CartaoCredito mapeando relacionamento bidirecional

Multiplicidade Um para Muitos

Relacionamento Unidirecional

@javax.persistence.OneToMany e JoinTable

Uma entidade pode manter relacionamentos que envolvem mais de uma referência para outra entidade. Ou seja, uma entidade pode agregar ou conter várias referências de outra. Por exemplo, um cliente pode ter vários telefones. Relacionamentos um para muitos e muitos para muitos requerem que o desenvolvedor trabalhe com uma coleção de referências ao invés de uma única.

Estrutura da Base de Dados Relacional

Para exemplificar o relacionamento um para muitos unidirecional, serão usadas as entidades Cliente e Telefone, cujas tabelas são definidas abaixo.

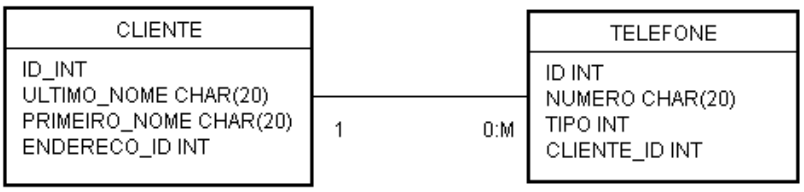


Figura 4.2 Relacionamento um para muitos usando uma chave estrangeira

O relacionamento um para muitos unidirecional entre Cliente e Telefone poderia ser implementado de várias formas. Nesse exemplo, optou-se por adicionar uma chave estrangeira para a tabela Cliente dentro da tabela Telefone. Na prática, um relacionamento desse tipo geralmente é mapeado com uma tabela de associação.

Assim, nesse exemplo, a tabela Telefone manterá uma chave estrangeira para a tabela Cliente, de forma que um ou mais registros de Telefone podem conter chaves estrangeiras para o mesmo registro Cliente. Em outras palavras, no banco de dados, os registros de Telefone apontam para os registros de Cliente. Já no modelo de programação, no entanto, ocorre o contrário: é a entidade Cliente que aponta para muitos Telefones. Esse esquema funciona da seguinte forma: quando você solicita que uma entidade Cliente retorne sua coleção de Telefones (chamando o método getTelefones()), ela consulta a tabela Telefone buscando por todos os registros que possuam uma chave estrangeira igual à chave primária da entidade Cliente. O uso de ponteiros contrários nesse tipo de relacionamento é mostrado na Figura acima.

Esse esquema de banco de dados mostra que a estrutura e os relacionamentos do banco de dados podem diferir dos relacionamentos do modelo de programação. Quando você está trabalhando com bases de dados que foram criadas antes da modelagem do sistema, cenários divergentes como o apresentado nesse exemplo são comuns.

Modelo de Programação

Você declara relacionamentos um para muitos usando a anotação @javax.persistence.OneToMany:

```
public @interface OneToMany {
  Class targetEntity() default void.class;
  CascadeType[] cascade() default {};
  FetchType fetch() default LAZY;
  String mappedBy() default "";
}
```

Código 4-5: Interface OneToMany

No modelo de programação, a multiplicidade é representada definindo uma propriedade que pode apontar para muitos objetos de entidade e anotando-a com @OneToMany. Para armazenar esse tipo de dados, serão usadas estruturas de dados do pacote java.util. Por exemplo, um Cliente pode ter relacionamentos com vários números de telefones (de casa, do trabalho, celular, fax, etc.), cada um destes representando um objeto da entidade Telefone. Logo, a entidade Cliente mantém todos os Telefones em uma estrutura `Collection`:

```
@Entity
public class Cliente implements java.io.Serializable {
  ...
  private Collection<Telefone> telefones = new
ArrayList<Telefone>();
  ...
  @OneToMany(cascade={CascadeType.ALL})
  @JoinColumn(name="CLIENTE_ID")
  public Collection<Telefone> getTelefones() {
       return telefones;
  }
  public void setTelefones(Collection<Telefone> fones) {
       this.telefones = fones;
  }
}
```

Código 4-6: Exemplo de anotação OneToMany

A anotação @JoinColumn referencia a coluna CLIENTE_ID na tabela Telefone. Note também que foi usado um template na definição da coleção de telefones (`Collection<Telefone>`). Isto indica o tipo concreto da coleção e permite ao gerenciador de persistência saber exatamente qual entidade você está relacionando com o Cliente. Se não fosse usada uma Collection "templeitizada", você teria que especificar o atributo @OneToMany.targetEntity(). O atributo mappedBy() deve ser usado apenas para relacionamentos bidirecionais.

A seguir é mostrado o código da classe Telefone. Note que a mesma não possui uma propriedade do tipo Cliente, pois como se trata de um relacionamento unidirecional, apenas a entidade Cliente mantém um relacionamento com seus Telefones.

```
import javax.persistence.*;
@Entity
public class Telefone implements java.io.Serializable {
  private int id;
  private String numero;
  private int tipo;

  public Telefone() {}
  public Telefone(String numero, int tipo) {
       this.numero = numero;
       this.tipo = tipo;
  }
  @Id @GeneratedValue
  public int getId() { return id; }
  public void setId(int id) { this.id = id; }
  ...
}
```

Código 4-7: Entidade Telefone

Para mostrar como uma entidade usa um relacionamento baseado em coleções, vamos olhar o seguinte código que interage com EntityManager:

```
  Cliente cliente = entityManager.find(Cliente.class, pk);
  Telefone fone = new Telefone("3332-9457", 5);
  cliente.getTelefones().add(fone);
```

Código 4-8: Cliente

O novo Telefone será automaticamente criado na base de dados porque o gerenciador de persistência verá que sua chave primária é 0, então gerará um novo ID (GeneratorType é AUTO) e inserirá o Telefone na base de dados.

Depois, se você precisa remover um Telefone do relacionamento, você precisa removê-lo da coleção dentro de Cliente e também da base de dados:

```
  cliente.getTelefones().remove(fone);
  entityManager.remove(fone);
```

Código 4-9: Cliente

Mapeamento com uma Tabela de Associação

Uma outra forma de mapear na base de dados um relacionamento um-para-muitos unidirecional é a criação de uma tabela de associação. No exemplo com as entidades Cliente e Telefone, a tabela de associação manteria duas colunas com chaves estrangeiras que apontam para registros da tabela Cliente e da tabela Telefone. Além disso, poderia ser adicionada uma restrição na coluna da chave estrangeira de Telefone na tabela de associação, para assegurar que ela contenha apenas entradas únicas. Já a coluna de chaves estrangeiras de Cliente poderia conter registros duplicados. A vantagem da tabela de associação é que o relacionamento entre os registros de Cliente e Telefone não precisa ser representado dentro destas tabelas. Esta é a forma de mapeamento mais usada para relacionamentos unidirecionais.

```
create table CLIENTE_TELEFONE (
     CLIENTE_ID int not null,
     TELEFONE_ID int not null unique
)
```

Código 4-10: Criação da tabela

Para ter esse tipo de mapeamento, é preciso mudar a anotação @JoinColumn na classe Cliente para a anotação @javax.persistence.JoinTable:

```
public @interface JoinTable {
     String name() default "";
     String catalog()default "";
     String schema() default "";
     JoinColumn[] joinColumns() default {};
     JoinColumn[] inverseJoinColumns() default {};
     UniqueConstraint[] uniqueConstraint() default {};
}
```

Código 4-11: Interface JoinTable

A anotação @JoinTable é muito parecida com a @Table, exceto pelos atributos adicionais: joinColumns() e inverseColumns(). O primeiro atributo define uma chave estrangeira na tabela de associação para a chave primária da entidade proprietária do relacionamento. Já o atributo inverseJoinColumns() mapeia a chave estrangeira do lado não-proprietário. Se um dos lados do relacionamento tivesse uma chave primária composta, apenas deveria ser adicionado mais anotações @JoinColumn no array:

```
@Entity
public class Cliente implements java.io.Serializable {
   ...
   private Collection<Telefone> telefones;
   ...
   @OneToMany(cascade={CascadeType.ALL})
   @JoinTable(name="CLIENTE_TELEFONE",
        joinColumns={@JoinColumn(name="CLIENTE_ID")},
        inverseJoinColumns={@JoinColumn(name="TELEFONE_ID")})
   public Collection<Telefone> getTelefones() {
        return telefones;
   }
   public void setTelefones(Collection<Telefone> fones) {
        this.telefones = fones;
   }
}
```

Código 4-12: Entidade Cliente

Com esta definição, a chave primária de Cliente é mapeada para a coluna de junção CLIENTE_ID na tabela CLIENTE_TELEFONE e a chave primária de Telefone é mapeada para a coluna de junção TELEFONE_ID.

Relacionamento Bidirecional

Os relacionamentos um-pra-muitos e muitos-pra-um bidirecionais são iguais. Veja a descrição de Relacionamento Bidirecional na próxima seção (Multiplicidade Muitos para Um).

Multiplicidade Muitos para Um

Relacionamento Unidirecional

@javax.persistence.ManyToOne

Um relacionamento unidirecional muitos-pra-um ocorre quando muitas entidades referenciam uma única outra entidade, mas esta última não sabe da existência do relacionamento. Por exemplo, no contexto de um sistema de gerenciamento de treinamentos, cada entidade Turma mantém um relacionamento muitos-pra-um com uma entidade Sala. O relacionamento é unidirecional: a Turma mantém uma referência para a Sala onde a aula será ministrada, mas a Sala, que pode ser alocada para várias turmas, não mantém referências para estas últimas

Estrutura da Base de Dados Relacional

O relacionamento muitos-pra-um entre Turma e Sala requer que a tabela Turma mantenha uma coluna de chave estrangeira para a tabela Sala (SALA_ID), com cada linha na tabela Turma apontando para uma linha na tabela Sala.

Figura 4.3 – Relacionamento muitos-pra-um na Base de Dados

Modelo de Programação

Relacionamentos muitos-para-um são descritos com a anotação @javax.persistence.ManyToOne:

```
public @interface ManyToOne {
     Class targetEntity() default void.class;
     CascadeType[] cascade() default {};
     FetchType fetch() default EAGER;
     boolean optional() default true;
}
```

Código 4-13: Interface ManyToOne

O modelo de programação para esse tipo de relacionamento é bastante simples: deve-se adicionar uma propriedade Sala dentro da classe Turma e sinalizá-la com a anotação @ManyToOne:

```
@Entity
public class Turma implements java.io.Serializable {
  private int id;
  private String nome;
  private Sala sala;

  // construtor default requerido por Java Persistence
  public Turma() {}
  public Turma(String nome, Sala sala) {
       this.nome = nome;
       this.sala = sala;
  }
  @Id @GeneratedValue
  public int getId() { return id; }
  public void setId(int id) { this.id = id; }
  ....
  @ManyToOne
  @JoinColumn(name="SALA_ID")
  public Sala getSala() { return sala; }
  public void setSala(Sala sala) { this.sala = sala; }
}
```

Código 4-14: Entidade Turma

A anotação @JoinColumn especifica que a tabela Turma tem uma coluna adicional chamada SALA_ID, que é uma chave estrangeira para a tabela Sala.

Como o relacionamento entre essas entidades é unidirecional, a definição da classe Sala não contém nenhuma propriedade que a relacione com a classe Turma.

Relacionamento Bidirecional

Os relacionamentos um-pra-muitos e muitos-pra-um bidirecionais são iguais. Eles ocorrem quando uma entidade mantém uma coleção de referências para uma outra entidade e cada entidade referenciada na coleção mantém uma única referência para aquela que a está agregando. Por exemplo, cada classe Aluno mantém uma coleção de referências para todos os interesses de alunos por determinado cursos (classe InteresseCurso). Cada InteresseCurso é particular de cada aluno. Este relacionamento é um-pra-muitos a partir da perspectiva da Aluno e é muitos-pra-um a partir da perspectiva da InteresseCurso.

Estrutura das Classes

Para representar esse tipo de relacionamento, a tabela INTERESSE_CURSO deve conter uma coluna que serve como uma chave estrangeira para a tabela ALUNO, mas a tabela ALUNO não mantém uma chave estrangeira para INTERESSE_CURSO:

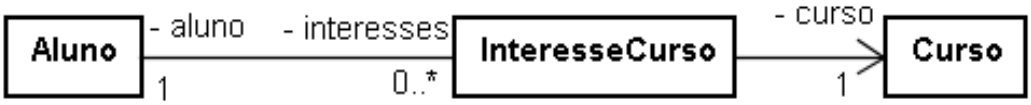


Figura 4.4 Relacionamento um-pra-muitos bidirecional em classes

O gerenciador de persistência pode determinar o relacionamento entre essas entidades consultando a tabela INTERESSE_CURSO, dispensando, assim, a presença de ponteiros para InteresseCurso dentro de Aluno.

Modelo de Programação

Para modelar o relacionamento entre Aluno e InteresseCurso, a classe InteresseCurso deve conter um atributo de relacionamento do tipo Aluno:

```
@Entity
public class InteresseCurso implements java.io.Serializable {
   @Id
   @GeneratedValue
   private int id;

   @OneToOne
   private Curso curso;

   @Temporal(TemporalType.DATE)
   private Date dataPreferencial;

   @Temporal(TemporalType.DATE)
   private Date dataLimite;

   private Periodo periodoPreferencial;

   @ManyToOne()
   private Aluno aluno;
}
```

Código 4-15: Entidade Reserva

Por outro lado, na classe Aluno deve-se adicionar também um atributo de relacionamento, mas este será uma coleção, que pode referenciar todas os interesses do aluno por um curso:

```
public class Aluno implements java.io.Serializable {
   @OneToMany(mappedBy="aluno")
   private ArrayList<InteresseCurso> interesses;
}
```

Código 4-16: Entidade Turma

Em um relacionamento um-pra-muitos bidirecional deve haver um lado proprietário no relacionamento. Atualmente, o Java Persistence requer que o lado muitos-pra-um seja sempre o proprietário - no exemplo considerado, é a classe InteresseAluno.

No exemplo considerado aqui, a entidade InteresseCurso nunca muda seu aluno. Se um aluno quer fazer uma pré-matricula em um curso, então precisará adicionar um novo objeto do tipo InteresseCurso, que por sua vez linka o aluno ao curso. Então, para removermos o interesse do aluno devemos passar como argumento o objeto do tipo InteresseCurso. Devemos portanto sempre passar o lado da coleção para remover o dado da coleção, ao invés de buscarmos todos os interesses do aluno, removermos da coleção o interesseCurso, e efetuarmos a mudança no base de dados. O seguinte exemplo mostra o correto e o que seria o desnecessário.

```
//esta correto
entityManager.remove(interesseCurso);
/*
aluno.getInsteresses().remove(interesseCurso);
entityManager.merge(aluno);
*/
```

Como a classe InteresseCurso é a proprietária do relacionamento, o atributo interesses de Aluno é atualizado com a remoção na próxima vez que ele for carregado do banco de dados.

Multiplicidade Muitos para Muitos

Relacionamento Bidirecional

@javax.persistence.ManyToMany

O relacionamento muitos-pra-muitos ocorre quando um objeto de uma classe possui uma coleção de objetos de uma outra classe e cada objeto referenciado nessa coleção mantém também uma coleção que referencia de volta os primeiros objetos. Por exemplo, ainda no contexto de um sistema de gerenciamento de treinamentos, uma Reserva pode referenciar-se a muitos Clientes (uma empresa pode fazer uma única reserva para vários de seus funcionários) e cada Cliente pode ter várias Reservas (uma pessoa pode querer fazer vários cursos).

Estrutura da Base de Dados Relacional

A estrutura das tabelas Reserva e Cliente já foi apresentada nas seções anteriores. Agora, para estabelecer um relacionamento muitos-pra-muitos, é necessária a criação de uma tabela de junção (ou de associação). Esta tabela, chamada de Reserva_Cliente, deve manter duas colunas de chaves estrangeiras: uma para a tabela Reserva e outra para a tabela Cliente. Relacionamentos muitos-pra-muitos sempre requerem uma tabela de associação em uma base de dados relacional normalizada.

Figura 4.5 Relacionamento muitos-pra-muitos bidirecional na Base de Dados

Modelo de Programação

Relacionamentos muitos-pra-muitos são definidos usando a anotação @javax.persistence.ManyToMany:

```
public @interface ManyToMany {
     Class targetEntity() default void.class;
     CascadeType[] cascade() default {};
     FetchType fetch() default LAZY;
     String mappedBy() default "";
}
```

Código 4-17: Interface ManyToMany

Para representar o relacionamento muitos-pra-muitos entre as classes Reserva e Cliente, deve-se adicionar um atributo coleção, que representa o relacionamento, em ambas as classes:

```
@Entity
public class Reserva implements java.io.Serializable {
   ...
   private Set<Cliente> clientes = new HashSet<Cliente>();
   ...
   @ManyToMany
   @JoinTable(name="RESERVA_CLIENTE",
        joinColumns={@JoinColumn(name="RESERVA_ID")},
         inverseJoinColumns={@JoinColumn(name="CLIENTE_ID")})
   public Set<Cliente> getClientes() { return clientes; }
   public void setClientes(Set clientes){
        this.clientes = clientes;
   }
   ...
}
```

Código 4-18: Entidade Reserva

O atributo de relacionamento clientes é declarado como um java.util.Set. O tipo Set é usado para expressar a restrição de que apenas Clientes únicos podem estar associados com uma Reserva, ou seja, não pode haver duplicação. No entanto, a efetividade da coleção Set depende de restrições de integridade referencial estabelecidas na base de dados.

Como ocorre em todos os relacionamentos bidirecionais, nesse também é necessário se ter um lado que representa a entidade proprietária do relacionamento. Neste exemplo, é a entidade Reserva. Logo, é essa classe que deve definir o mapeamento @JoinTable. O atributo joinColumns() identifica a coluna de chave estrangeira na tabela Reserva_Cliente que referencia a tabela Reserva e o atributo inverseJoinColumns() identifica a chave estrangeira que referencia a tabela Cliente.

A classe Cliente também deve ser modificada para conter um atributo coleção que mantém todas as suas Reservas:

```
@Entity
public class Cliente implements java.io.Serializable {
   ...
   private Collection<Reserva> reservas = new
ArrayList<Reserva>();
   ...
   @ManyToMany(mappedBy="clientes")
   public Collection<Reserva> getReservas() { return reservas;}
   public void setReservas(Collection<Reserva> res) {
        this.reservas = res;
   }
   ...
}
```

Código 4-19: Entidade Cliente

O atributo mappedBy() identifica a propriedade na classe Reserva que define o relacionamento. Ele também identifica a classe Cliente como sendo o lado inverso do relacionamento.

Quando for necessário modificar ou remover objetos do relacionamento, isto deve ser feito sempre a partir do lado proprietário do mesmo. Para remover um cliente de uma reserva, por exemplo, deve-se fazer:

```
Reserva reserva = em.find(Reserva.class, id);
reserva.getClientes().remove(cliente);
```

Como Reserva é o lado proprietário, você deve remover o Cliente do atributo clientes da classe Reserva. Se for feito o contrário, remover a Reserva do atributo reservas da classe Cliente, esta modificação não seria atualizada na base de dados.

Relacionamento Unidirecional

O relacionamento muitos-pra-muitos unidirecional ocorre quando um objeto de uma classe possui uma coleção de objetos de uma outra classe e cada objeto referenciado nessa coleção não mantém uma coleção que referencia de volta os primeiros objetos. Por exemplo, no sistema de gerenciamento de treinamentos, um Curso pode ser ministrado por muitos Instrutores e ainda um Instrutor pode ministrar também muitos Cursos. Enquanto que a classe Curso precisa saber quem são os Instrutores que o ministram, o Instrutor não necessariamente precisa manter referências para os Cursos que ele apenas ministra (ou seja, não é responsável pela manutenção do curso e seus materiais didáticos). Assim, a classe Curso mantém uma coleção de Instrutores, mas a classe Instrutor não mantém referências para os Cursos ministrados.

Estrutura da Base de Dados Relacional

Para representar o relacionamento muitos-pra-muitos unidirecional entre Curso e Instrutor, precisa-se de uma tabela de associação, chamada Instrutor_Curso:

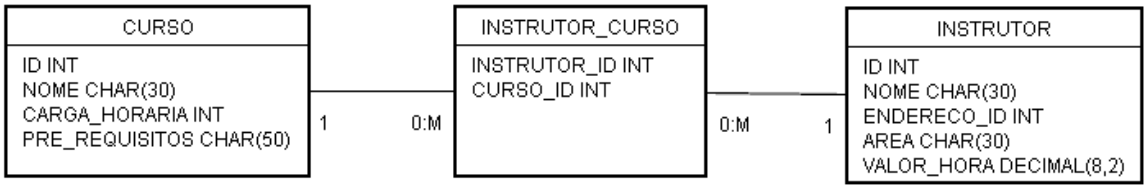


Figura 4.5 – Relacionamento muitos-pra-muitos unidirecional na Base de Dados

Modelo de Programação

Para modelar esse relacionamento entre as classes Curso e Instrutor, é preciso adicionar um atributo coleção que referencia Instrutores dentro da classe Curso:

```
   @Entity
   public class Curso implements java.io.Serializable {
      ...
      private Set<Instrutor> instrutores = new
HashSet<Instrutor>();
      ...
      @ManyToMany
      @JoinTable(name="INSTRUTOR_CURSO",
            joinColumns={@JoinColumn(name="INSTRUTOR_ID")},
             inverseJoinColumns={@JoinColumn(name="CURSO_ID")})
      public Set<Instrutor> getInstrutores() { return instrutores;
}
      public void setInstrutores(Set instrutores){
            this.instrutores = instrutores;
      }
      ...
   }
```

Código 4-20: Entidade Curso

Já a classe Instrutor não mantém um atributo que referencia Cursos, o que indica que o relacionamento é unidirecional.

Mapeando Relacionamentos Baseados em Coleções

Os exemplos de relacionamentos um-pra-muitos e muitos-pra-muitos abordados nas seções anteriores usaram os tipos `java.util.Collection` e `java.util.Set`. Mas, a especificação JPA permite também que um relacionamento seja representado com um `java.util.List` ou `java.util.Map`.

Relacionamento Baseado em Lista Ordenada

@javax.persistence.OrderBy

O tipo `java.util.List` pode expressar um relacionamento baseado em coleção, permitindo ordenar o relacionamento retornado com base em um conjunto específico de critérios. Isto requer um meta-dado adicional que é fornecido pela anotação @javax.persistence.OrderBy:

```
package javax.persistence;
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OrderBy {
      String value() default "";
}
```

Código 4-21: Interface OrderBy

O atributo value() permite declarar um JPQL parcial que especifica como você quer que o relacionamento seja ordenado quando ele é recuperado da base de dados. Se o atributo estiver em branco, o `List` é ordenado de forma ascendente com base no valor da chave primária.

Considerando como exemplo o relacionamento muitos-pra-muitos bidirecional entre Reserva e Cliente, e tendo o atributo clientes de Reserva como um `List` ordenado alfabeticamente pelo último nome da classe Cliente temos:

```
@Entity
public class Reserva implements java.io.Serializable {
   ...
   private List<Cliente> clientes = new ArrayList<Cliente>();
   ...
   @ManyToMany
   @OrderBy("ultimoNome ASC")
   @JoinTable(name="RESERVA_CLIENTE"),
              joinColumns={@JoinColumn(name="RESERVA_ID")},
              inverseJoinColumns={@JoinColumn(name="CLIENTE_ID")})
   public List<Cliente> getClientes() { return clientes; }
   ...
}
```

Código 4-22: Uso da anotação @OrderBy

"ultimoNome ASC" diz ao gerenciador de persistência para ordenar o ultimoNome de Cliente de forma ascendente. DESC é a forma descendente. É possível também especificar restrições adicionais, como `@OrderBy("ultimoNome asc, primeiroNome asc")`. Neste caso, a lista seria ordenada pelo último nome e em caso de valores duplicados, seria ordenada pelo primeiro nome.

Relacionamento Baseado em Mapas

@javax.persistence.MapKey

A interface `java.util.Map` também pode ser usada para expressar relacionamentos baseados em coleção. Neste caso, o gerenciador de persistência cria um mapa onde a chave é uma propriedade específica da entidade relacionada e o valor é a própria entidade. Quando se usa um `java.util.Map`, a anotação @javax.persistence.MapKey deve ser usada:

```
package javax.persistence;
@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface MapKey {
     String name() default "";
}
```

Código 4-23: Interface MapKey

O atributo name() é o nome da propriedade que você quer que represente a chave do objeto `Map`. Se o mesmo ficar em branco, é assumido como chave do `Map` a chave primária da entidade relacionada.

Como exemplo, será usado abaixo um `Map` para representar o relacionamento um-pra-muitos unidirecional entre Instrutor e Telefone:

```
@Entity
public class Instrutor implements java.io.Serializable {
   ...
   private Map<String,Telefone> telefones = new
   HashMap<String,Telefone>();
   ...
   @OneToMany(cascade={CascadeType.ALL})
   @JoinColumn(name="INSTRUTOR_ID")
   @MapKey(name="numero")
   public Map<String,Telefone> getTelefones() {
        return telefones;
   }
   public void setTelefones(Map<String,Telefone> fones) {
        this.telefones = fones;
   }
}
```

Código 4-24: Uso da anotação @MapKey

Porém, como você deve ter percebido se analisar o modelo de tabelas gerado no banco, a tabela Telefone possui chave estrangeira de Cliente e também chave estrangeira de Instrutor. Eis a explicação do por que muitas associações um para muitos ou muitos para um tornarem-se em tabela relacionamentos muitos para muitos. A melhor solução no caso de telefone seria termos uma tabela intermediária entre Telefone e Cliente e outra tabela intermediária entre Telefone e Instrutor. Outra solução para mantermos o relacionamento um para muitos é definirmos uma superclasse de Cliente e Instrutor Pessoa por exemplo, passado o atributo telefones para a classe mãe.

Continuação do Exemplo do Curso

Nesta seção o nosso objetivo é exercitar as associações entre as classes. Assim, a visibilidade dos atributos foi removida para darmos mais enfoque para as relações.

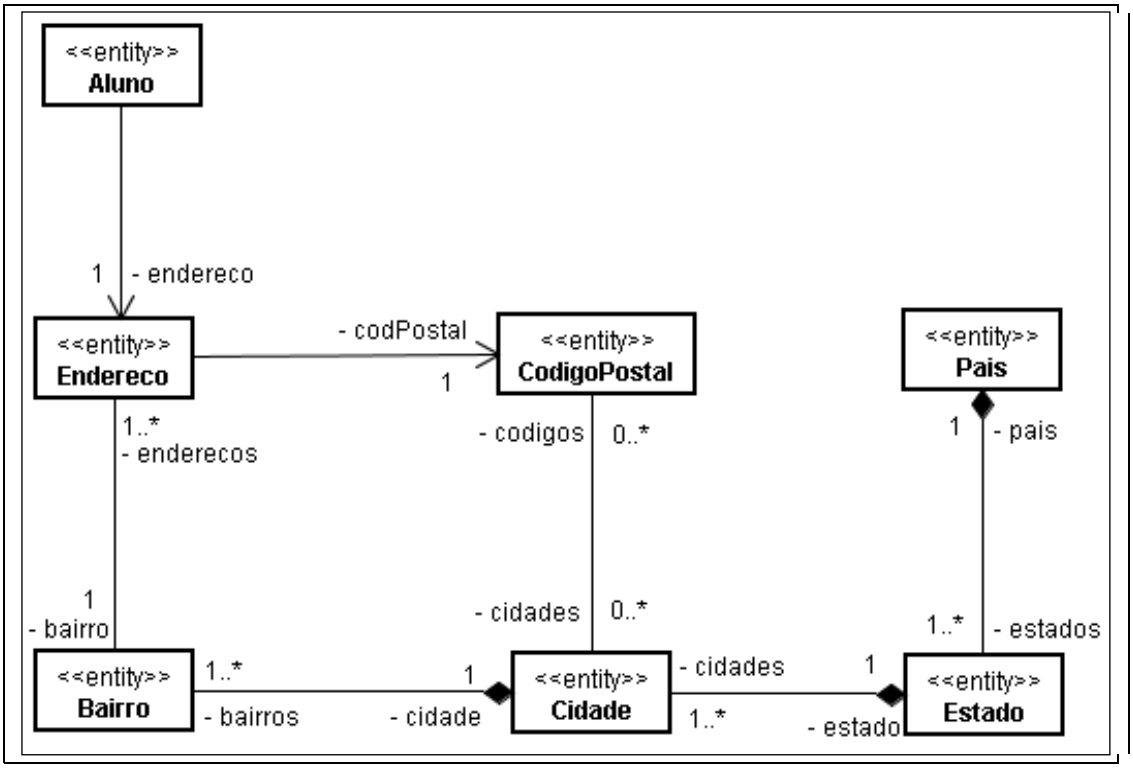
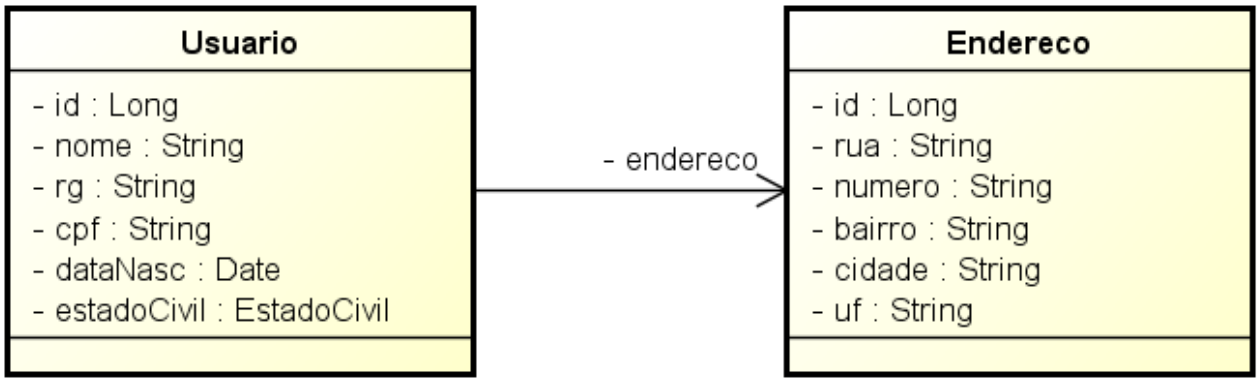


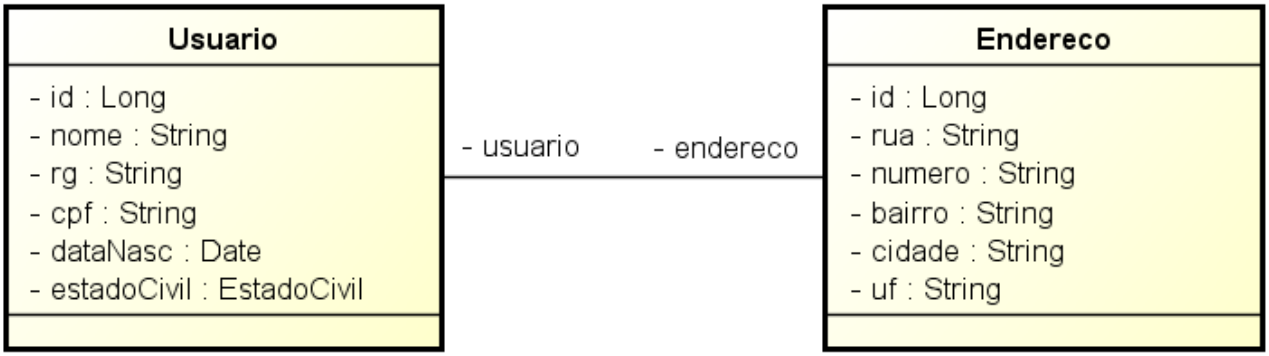
Figura 4.6 Desenvolvimento das Associações

Exercícios

1) Seguindo o diagrama abaixo, especifique a associação unidirecional entre Usuario e Endereco. Em seguida teste a geração da base de dados.



2) Agora mude a associação dentre Usuario e Endereco, tornando-a bidirecional. Observe e aplique as mudanças necessárias nos mapeamentos JPA que deverão ser acrescidas à entidade Endereço.



3) Agora, opte por uma associação unidirecional com multiplicidade Um-Para-Muitos, onde cada entidade Usuario poderá ter 1 ou mais Enderecos associados a si. Faça uso de uma @javax.persistence.JoinTable. Abaixo um diagrama que ilustra esse relacionamento: