Implementação - Relacionamento

Orientação a Objetos – DCC025

Prof. Edmar Welington Oliveira
edmar.oliveira@ufjf.edu.br

Universidade Federal de Juiz de Fora – UFJF
Departamento de Ciência da Computação - DCC

2011.3

Resumo – Implementação Relacionamentos

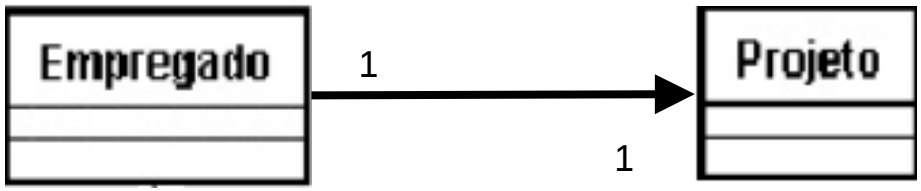
Conectividade e Multiplicidade

Conectividade	Multiplicidade em um extremo	Multiplicidade no um extremo
Um para um	**0..1** **1**	**0..1** **1**
Um para muitos	**0..1** **1**	***** **1..***
Muitos para muitos	***** **1..***	***** **1..***

A tabela acima mostra algumas possibilidades de relacionamentos entre classes. Ao longo do material, outras são apresentadas. Contudo, boa parte das implementações irão se basear me combinações envolvendo as multiplicidades acima. Algumas implementações além das explicadas neste material, podem ser obtidas a partir do que for apresentado e explicado (considerando a tabela acima).

Unidirecional Um/Um

Unidirecional Um/Um

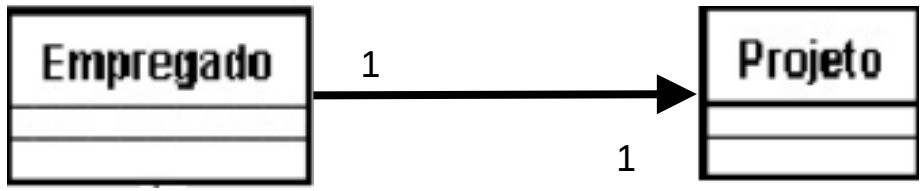


Observe que, para a classe Empregado, a multiplicidade 1 (ao lado da classe Projeto) implica em: ao se criar uma instância de Empregado, é obrigatório ligá-la, imediatamente, a uma instância de Projeto. Logo, não existe empregado sem projeto.

OBS: um projeto pode ser criado sem ter que se criar um empregado.

Como, em implementação, deve-se forçar essa obrigatoriedade?
Uso dos construtores

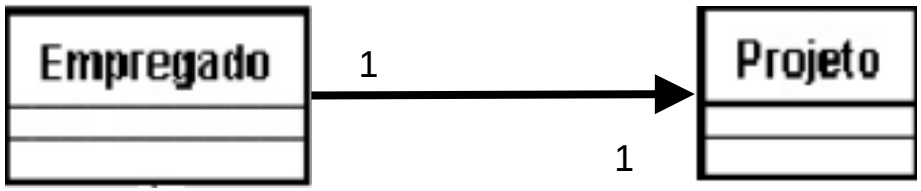
Unidirecional Um/Um



Uma multiplicidade de valor 1 significa que uma instância está associada, em um dado momento, a apenas uma instância de outra classe. Naturalmente, ao longo da vida da instância, ela poderá ser associada a muitas outras, porém somente a uma de cada vez.

Logo, uma instância de Projeto, em um dado instante, estará associada a apenas uma instância de Empregado.

Unidirecional Um/Um




```
3  public class Empregado {
4
5      private Projeto projeto;
6
7      public Empregado(Projeto p){
8          this.projeto = p;
9      }
10 }
```

Armazena a associação de Empregado para Projeto

```
3  public class Projeto {
4
5      //Atributos e métodos
6  }
```

Unidirecional Um/Um

```java
public class Empregado {

    private Projeto projeto;

    public Empregado(Projeto p){
        this.projeto = p;
    }
}
```

```java
public class Projeto {

    //Atributos e métodos
}
```

Observe que, na classe Empregado, é criado um atributo “projeto” do tipo Projeto (devido ao relacionamento unidirecional Empregado → Projeto).

Como é obrigatório que um projeto seja associado à empregado quando se cria um empregado (devido a multiplicidade 1 do lado da classe Projeto), o construtor de Empregado recebe um objeto do tipo Projeto como parâmetro, inicializando seu atributo projeto

Unidirecional Um/Um

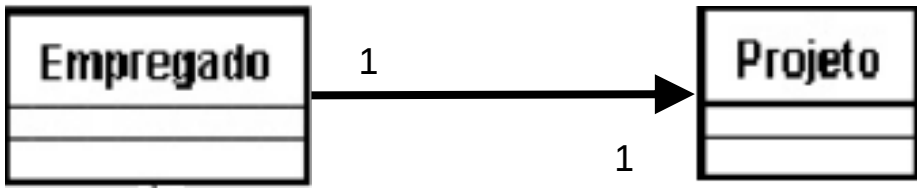
```
3  public class Empregado {
4
5      private Projeto projeto;
6
7      public Empregado(Projeto p){
8          this.projeto = p;
9      }
10 }
```

O que aconteceria se “p” fosse Null?

```
3  public class Empregado {
4
5      private Projeto projeto;
6
7      public Empregado(Projeto p){
8          if(p == null){
9              //lançar uma exceção
10         }
11         this.projeto = p;
12     }
13 }
```

Não há como garantir que o que será passado par ao construtor de Empregado é, de fato, uma instância de Projeto. Logo, é necessário fazer uma verificação – será lançada uma exceção caso seja Null, por exemplo

Unidirecional Um/Um




```
3  public class Empregado {
4
5      private Projeto projeto;
6
7      public Empregado(Projeto p){
8          this.projeto = p;
9      }
10 }
```

(1)

```
3  public class Projeto {
4
5      //Atributos e métodos
6  }
```

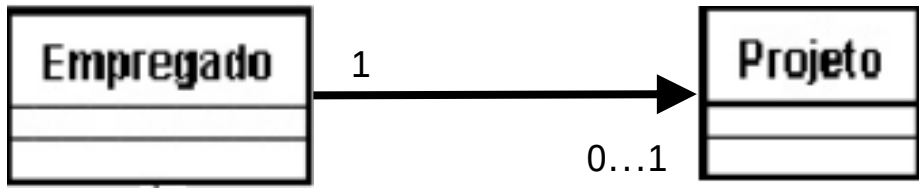
(2)

```
3  public class Empregado {
4
5      private Projeto projeto;
6
7      public Empregado(){
8          projeto = new Projeto();
9      }
10 }
```

Qual a diferença entre 1 e 2?
Qual a melhor implementação?
Alguma pode causar algum problema?

Unidirecional Um/Um

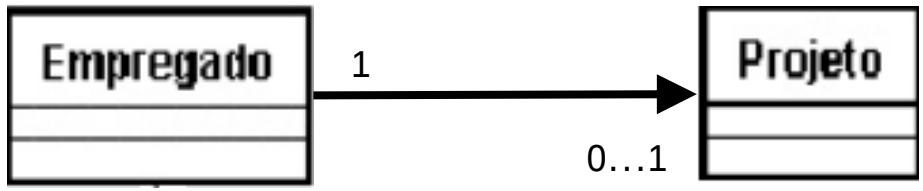
Unidirecional Um/Um



Observe que, para a classe Empregado, a multiplicidade 0...1 (ao lado da classe Projeto) não implica em ter que, ao se criar uma instância de Empregado, ligá-la a uma instância de Projeto.Logo, pode existir empregado sem projeto.

A associação 0...1 não é obrigatória, diferentemente de “1”

Unidirecional Um/Um




```
3  public class Empregado {
4
5      private Projeto projeto;
6
7      public Empregado(){
8      }
9  }
```

```
3  public class Projeto {
4
5      //Atributos e métodos
6  }
```

Unidirecional Um/Um

```
3  public class Empregado {
4
5      private Projeto projeto;
6
7      public Empregado(){
8      }
9  }
```

```
3  public class Projeto {
4
5      //Atributos e métodos
6  }
```

Observe que, na classe Empregado, é criado um atributo “projeto” do tipo Projeto (devido ao relacionamento unidirecional Empregado → Projeto).

Como não é obrigatório que um projeto seja associado à empregado quando se cria um empregado (veja a multiplicidade 0...1 do lado da classe Projeto), o construtor de Empregado não precisa receber um objeto do tipo Projeto como parâmetro

Unidirecional Um/Um

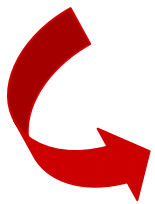
```
3  public class Empregado {
4
5      private Projeto projeto;
6
7      public Empregado(){
8      }
9  }
```

```
3  public class Projeto {
4
5      //Atributos e métodos
6  }
```

Se não existe construtor, uma forma de atribuir uma instância de Projeto ao atributo projeto de Empregado é através de um método em Empregado, que irá receber um objeto Projeto como parâmetro.

Unidirecional Um/Um

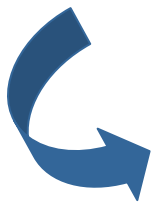
```
3  public class Empregado {
4
5      private Projeto projeto;
6
7      public Empregado(){
8      }
9  }
```



```
3  public class Empregado {
4
5      private Projeto projeto;
6
7      public Empregado(){
8      }
9
10     public void associaProjeto(Projeto p){
11         this.projeto = p;
12     }
13 }
```

Unidirecional Um/Um

```
3  public class Empregado {
4
5      private Projeto projeto;
6
7      public Empregado(){
8      }
9  }
```

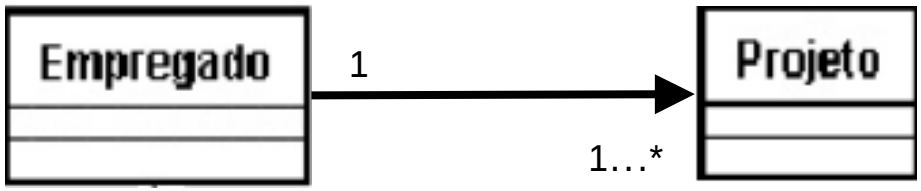


```
6  public class Empregado {
7
8      private Projeto projeto;
9
10     public Empregado(){
11         this.projeto = null;
12         //inicialização de outros atributos
13     }
14 }
```

Para inicializar o etributo empregado, neste caso, teremos que – dentro do construtor de Empregado, atribuir Null para o mesmo. Depois, utilizaríamos um método para atribuir um objeto Projeto ao atributo projeto.

Unidirecional Um/Muitos

Unidirecional Um/Muitos



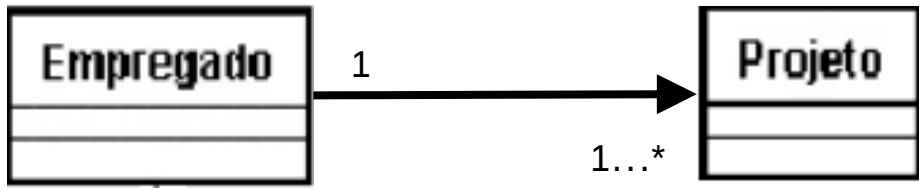
Observe que, para a classe Empregado, a multiplicidade 1...* (ao lado da classe Projeto) implica em ter que, ao se criar uma instância de Empregado, ligá-la a uma instância de Projeto. Logo, não pode existir empregado sem, ao menos um projeto.

A associação 1...* é obrigatória

Cada instância de empregado pode estar alocada para vários projetos. Contudo, DEVE estar alocada em pelo menos um projeto.

OBS: observe que uma diferença em relação a associação um/um é a necessidade de se criar, na classe, Empregado, um atributo que represente um conjunto

Unidirecional Um/Muitos




```
 6 public class Empregado {
 7
 8     private Set<Projeto> projetos = new HashSet<Projeto>();
 9
10     public Empregado(Projeto p){
11         projetos.add(p);
12     }
13
14     public Empregado(Set<Projeto> projetos){
15         //manilupação do conjunto
16     }
17 }
```

```
3 public class Projeto {
4
5     //Atributos e métodos
6 }
```

Unidirecional Um/Muitos

```
 6 public class Empregado {
 7
 8     private Set<Projeto> projetos = new HashSet<Projeto>();
 9
10     public Empregado(Projeto p){
11         projetos.add(p);
12     }
13
14     public Empregado(Set<Projeto> projetos){
15         //manilupação do conjunto
16     }
17 }
```

Observe que, no modelo, cada empregado deve, obrigatoriamente, estar associado a pelo menos um projeto. Logo, ao se criar uma instância de Empregado, a mesma deve ser associada a uma instância de projeto (podendo ser associadas a muitas).

Como um empregado pode estar associado 1 ou * projetos, construímos dois construtores: um que aceita apenas um projeto e outro que aceita vários

Unidirecional Um/Muitos

```
 6 public class Empregado {
 7
 8     private Set<Projeto> projetos = new HashSet<Projeto>();
 9
10     public Empregado(Set<Projeto> projetos){
11         //manipulação do conjunto
12     }
13 }
```

Observe que a implementação pode ficar da forma como acima (retirou-se o construtor que recebe apenas um projeto)

É fato que um cojunto de projetos é uma coleção de 0, 1 ou vários projetos. Logo, não seria necessário, ao menos a princípio, de um construtor que recebesse apenas um único projeto. Nada impede que o construtor do código acima receba um conjunto com apenas 1 projeto

OBS: tentar passar um objeto projeto causará erro. Deverá ser passado um conjunto com um objeto. Um objeto até pode ser passado, mas isso implica em aplicar conhecimentos ainda não vistos até o momento no curso

Unidirecional Um/Muitos

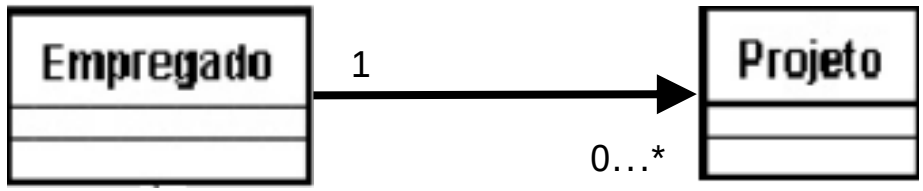
```
6  public class Empregado {
7
8      private Set<Projeto> projetos = new HashSet<Projeto>();
9
10     public Empregado(Set<Projeto> projetos){
11         //manipulação do conjunto
12     }
13 }
```

Continuando: mesmo que se deseje passar apenas uma instância de projeto para o construtor de Empregado, deverá, primeiro, ser criado um conjunto de projetos com apenas um projeto. Assim, esse conjunto seria passado como parâmetro para o construtor de Empregado.

OBS: Se houver, por alguma razão, a necessidade de se possibilitar passar apenas uma única instância de Projeto, sem considerar a construção de um conjunto, um construtor deve ser mantido para tal, com abaixo

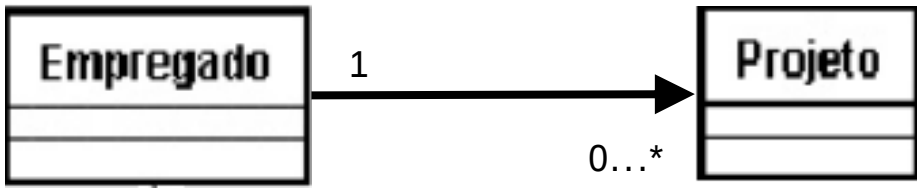
```
public Empregado(Projeto p){
    projetos.add(p);
}
```

Unidirecional Um/Muitos



Observe que, para a classe Empregado, a multiplicidade 0...* (ao lado da classe Projeto) não implica em ter que, ao se criar uma instância de Empregado, ligá-la a uma instância de Projeto. Logo, pode existir empregado sem projeto associado. Além disso, um empregado pode particpar de vários projetos

Unidirecional Um/Muitos




```
6  public class Empregado {
7
8      private Set<Projeto> projetos = new HashSet<Projeto>();
9
10     public Empregado(){
11     }
12 }
```

```
3  public class Projeto {
4
5      //Atributos e métodos
6  }
```

Unidirecional Um/Muitos

```
6  public class Empregado {
7
8      private Set<Projeto> projetos = new HashSet<Projeto>();
9
10     public Empregado(){
11     }
12 }
```

```
3  public class Projeto {
4
5      //Atributos e métodos
6  }
```

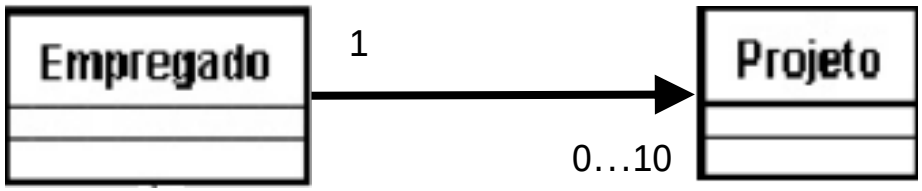
Observe que, na classe empregado, o atributo “Set projetos” representa uma coleção de objetos do tipo Projeto. Essa coleção possui 0 ou muitos objetos do tipo Projeto.

Observe, ainda, que o construtor de Empregado não precisa receber um objeto de projeto (não há a obrigatoriedade de, ao se criar uma instância de Empregado, associá-la a uma instância de Projeto).

OBS: Na classe Empregado, será necessário implementar métodos para manipula a coleção

Unidirecional Um/Intervalo

Unidirecional Um/Intervalo



Para este caso, um objeto de Empregado pode trabalhar em, no máximo, 10 projetos, podendo não trabalhar em nenhum.

Unidirecional Um/Intervalo

```
 6 public class Empregado {
 7
 8     private Set<Projeto> projetos;
 9     private static final int numeroMaxProjeto = 10;
10
11     public Empregado(){
12         projetos = new HashSet<Projeto>();
13     }
14
15     public void associarProjeto(Projeto p){
16         if(projetos.size() < Empregado.numeroMaxProjeto){
17             projetos.add(p);
18         }
19         else{
20             System.out.println("Número máximo atingido");
21         }
22     }
23 }
```

```
3 public class Projeto {
4
5     //Atributos e métodos
6 }
```

Unidirecional Um/Intervalo

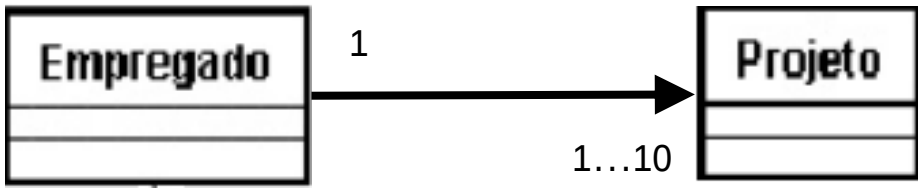
```
6  public class Empregado {
7
8      private Set<Projeto> projetos;
9      private static final int numeroMaxProjeto = 10;
10
11     public Empregado(){
12         projetos = new HashSet<Projeto>();
13     }
14
15     public void associarProjeto(Projeto p){
16         if(projetos.size() < Empregado.numeroMaxProjeto){
17             projetos.add(p);
18         }
19         else{
20             System.out.println("Número máximo atingido");
21         }
22     }
23 }
```

Definição de número máximo de projetos

Inicialização de atributo

Verificação de número máximo

Unidirecional Um/Intervalo



Se fosse uma multiplicidade 1...10, teríamos que ter o cuidado de alterar o construtor da classe Empregado, de forma a obrigar que, ao menos, um objeto de Empregado esteja associado a um objeto de Projeto

O que muda em relação ao código anterior (multiplicidade 0...10) é o construtor.

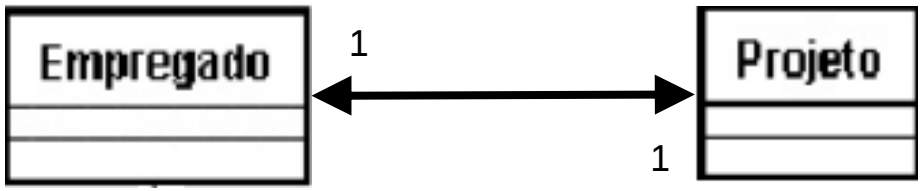
Unidirecional Um/Intervalo

```
 6 public class Empregado {
 7
 8     private Set<Projeto> projetos;
 9     private static final int numeroMaxProjeto = 10;
10
11     public Empregado(Set<Projeto> p){
12         projetos = new HashSet<Projeto>();
13
14         //manipulação do conjunto "p"
15         //Deve-se pegar o(s) objeto(s) de "p" e passar para o atributo "projetos"
16     }
17
18     public void associarProjeto(Projeto p){
19         if(projetos.size() < Empregado.numeroMaxProjeto){
20             projetos.add(p);
21         }
22         else{
23             System.out.println("Número máximo atingido");
24         }
25     }
26 }
```

Será visto ao estudarmos conjunto

Bidirecional Um/Um

Bidirecional Um/Um



Observe que, para ambas as classes, a multiplicidade 1 implica em algo "mandatório/obrigatório". Neste caso, Ao se criar uma instância de Empregado, é obrigatório ligá-lo, imediatamente, a uma instância de Projeto. E da mesma forma, ao se criar uma instância de Projeto, é obrigatório ligá-lo, imediatamente, a uma instância de Empregado

Qual o problema inerente a esta necessidade?

Bidirecional Um/Um

```
 4 public class Empregado {
 5
 6     private Projeto projeto;
 7
 8     public Empregado(){
 9         this.projeto = new Projeto(this);
10     }
11
12     public Projeto getProjeto(){
13         return this.projeto;
14     }
15 }
```

```
 3 public class Projeto {
 4
 5     private Empregado empregado;
 6
 7     public Projeto(Empregado empregado){
 8         this.empregado = empregado;
 9     }
10
11     public Empregado getEmpregado(){
12         return this.empregado;
13     }
14 }
```

Bidirecional Um/Um

```
 4 public class Empregado {
 5
 6     private Projeto projeto;
 7
 8     public Empregado(){
 9         this.projeto = new Projeto(this);
10     }
11
12     public Projeto getProjeto(){
13         return this.projeto;
14     }
15 }
```

Analisando a classe Emrpegado, observamos que o construtor da mesma cria um objeto da classe Projeto. Isto quer dizer que, quando for instanciado um objeto de Empregado, automaticamente e imedidamente, será instanciado um objeto de Projeto e os dois estarão relacionados.

A palavra reservada “this” em “new Projeto(this)” significa que para o objeto Projeto que está sendo instanciado, será passado o Empregado que abacou de ser criado. Observe o construtor da classe Projeto.

Bidirecional Um/Um

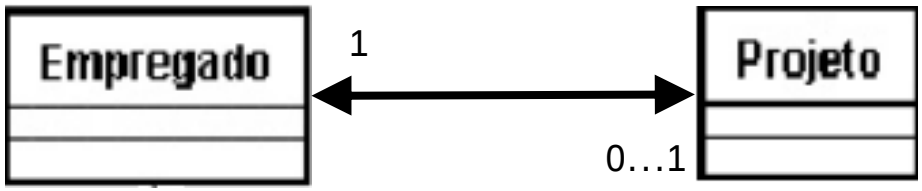
```
3  public class Projeto {
4
5      private Empregado empregado;
6
7      public Projeto(Empregado empregado){
8          this.empregado = empregado;
9      }
10
11     public Empregado getEmpregado(){
12         return this.empregado;
13     }
14 }
```

O construtor de Projeto recebe como parâmetro um objeto da classe Empregado. Novamente, ao criarmos um empregado, um projeto será criado na mesma hora e ete empregado será passado como parâmetro para o construtor de Projeto.

O que aconteceria se ambos os construtores, da classe Empregado e Projeto, recebessem um objeto da outra classe? Ex: Empregado recebendo objeto de Projeto e Projeto recebendo objeto de Empregado

Bidirecional Um/Um

Bidirecional Um/Um

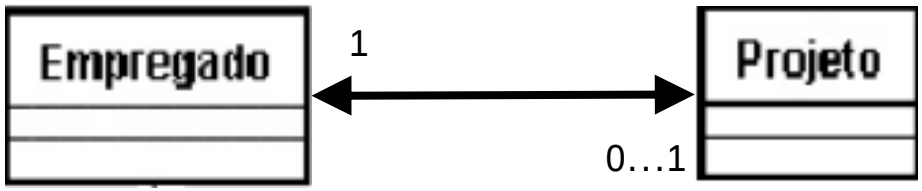


Observe que, neste exemplo, não existe a obrigatoriedade de se associar Empregado a Projeto. Isso quer dizer que podemos ter um empegado que não esteja “alocada” a algum projeto.

A implementação é semelhante a associação unidirecional. Contudo, em ambas as classes, deverá ter um atributo da outra classe

OBS: Observe que se um empregado for criado, ele não precisa ser ligado a um projeto. Contudo, se um projeto for criado, ele precisa, imediatamente, ser ligado a um empregado.

Bidirecional Um/Um

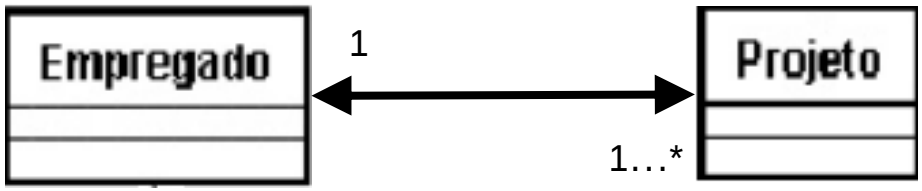



```
4  public class Empregado {
5
6      private Projeto projeto;
7
8      public Empregado(){
9      }
10 }
```

```
3  public class Projeto {
4
5      private Empregado empregado;
6
7      public Projeto(Empregado empregado){
8          this.empregado = empregado;
9      }
10 }
```

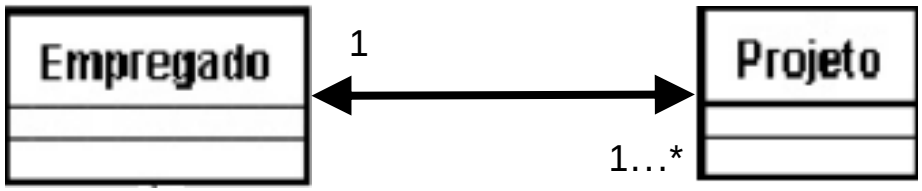
Bidirecional Um/Muitos

Bidirecional Um/Muitos



Neste caso, utiliza-se os conceitos já estudados nos casos anteriores.

Bidirecional Um/Muitos




```java
public class Empregado {

    private Set<Projeto> projetos;

    public Empregado(Set<Projeto> p){
        projetos = new HashSet<Projeto>();

        //deve-se passar os objetos em "p" para o atributo "projetos"
        //métodos devem ser implementados para isto
    }
}
```

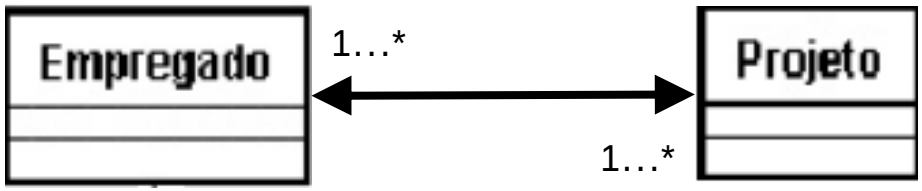
```java
public class Projeto {

    private Empregado empregado;

    public Projeto(Empregado empregado){
        this.empregado = empregado;
    }
}
```

Bidirecional Muitos/Muitos

Bidirecional Muitos/Muitos



Neste caso, cada classe deverá manter um atributo da outra classe. Além disso, observa-se que existe uma restrição de se ter, ao menos, um objeto de uma classe associada a outra (podendo ter várias).

Uma instância de empregado participa de, pelo menos, um projeto. Da mesma forma, um projeto contém, pelo menos, um empregado.

OBS: precisa ficar claro que, para o caso da bidirecionalidade, ambos os lados precisam ter conhecimento um do outro. Empregado sabe que projeto existe e projeto sabe que empregado existe.

Bidirecional Muitos/Muitos

```
 7 public class Empregado {
 8
 9     private Set<Projeto> projetos;
10
11     public Empregado(){
12         projetos = new HashSet<Projeto>();
13     }
14
15     public void adicionarProjeto(Projeto p){
16         if(!projetos.contains(p)){
17             projetos.add(p);
18             p.adicionarEmpregado(this);
19         }
20
21     }
22 }
```

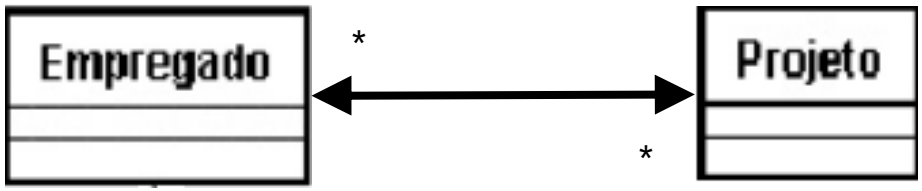
Em termos de código, observa-se que o construtor não se altera se comparado a multiplicidade 0...* (ou *). Ao se criar um objeto de Empregado, um conjunto de projetos também é alocado como atributo. No método “adicionaProjeto”, um projeto é recebido como parâmetro. Realiza-se uma verificação se o mesmo já existe no conjunto de projetos e, caso não exista, adiciona-se o mesmo. Observe que, neste momento, invoca-se o método “adicionarEmpregado”, passando como parâmetro o empregado que acabou de ser criado. Logo, quando um projeto é associado a um empregado, o mesmo empregado é associado ao projeto

Bidirecional Muitos/Muitos

```
 6 public class Projeto {
 7
 8     private Set<Empregado> empregados;
 9
10     public Projeto(){
11         empregados = new HashSet<Empregado>();
12     }
13
14     public void adicionarEmpregado(Empregado e){
15         if(!empregados.contains(e)){
16             empregados.add(e);
17             e.adicionarProjeto(this);
18         }
19     }
20 }
```

Continuando: Todo projeto para o qual um empregado trabalha precisa estar presente no conjunto de projetos de empregado (ou seja, no atributo “projetos” de Empregado). Da mesma forma, todo empregado associado a um projeto precisa estar presente no conjunto de empregados de projeto.

Bidirecional Muitos/Muitos



Funciona como no caso anterior.

Implementação - Relacionamento

Orientação a Objetos – DCC025

Prof. Edmar Welington Oliveira
edmar.oliveira@ufjf.edu.br

Universidade Federal de Juiz de Fora – UFJF
Departamento de Ciência da Computação - DCC

2011.3