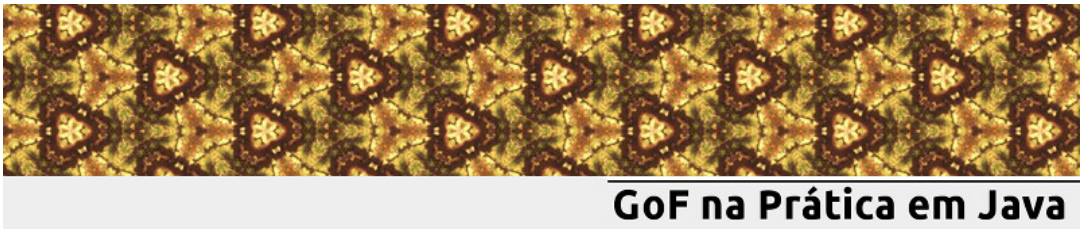
Iterator

Fernando Anselmo
GoF na Prática em Java

Função deste Padrão

Prover um modo de acesso a elementos de uma coleção de objetos, sequencialmente, sem a exposição de suas estruturas internas.



1 Ficha do Padrão

Tipo : Comportamental, especificamente voltados para a comunicação entre objetos.

Conhecimentos : Generics, Enumeração, Interface e Classes Concretas.

Consequências : Suportar variações no percurso de um objeto agregado. Mais que um caminho pode estar pendente em um agregado.

É usado quando : Acessar o conteúdo de um objeto sem expor sua representação interna com suporte a múltiplas varreduras de objetos agregados. Prover uma interface uniforme para varrer diferentes estruturas agregadas.

2 Problema

Padronizar o acesso de uma classe que suporta a lista de Dependentes de um determinado Funcionário, evitando erros do tipo: acessar um Dependente que não existe.

3 Prévia Estrutura de Classes

Lista de enumeração com os possíveis graus de parentesco dos dependentes:

Listagem 1: *Enumeração GrauParentesco*

```
1 enum GrauParentesco {
2   CONJUGE, FILHO, PAI
3 }
```

Classe de implementação do dependente:

Listagem 2: *Classe Dependente*

```
1 class Dependente {
2   private String nome;
3   private GrauParentesco grauParentesco;
4
```

```
  public Dependente(String nome, GrauParentesco grauParentesco) {
    this.nome = nome;
    this.grauParentesco = grauParentesco;
  }
  public String getNome() {
    return nome;
  }
  public String getGrauParentesco() {
    switch (grauParentesco) {
      case CONJUGE: return "Conjugê";
      case FILHO: return "Filha/o";
      case PAI: return "Pai/Mãe";
    }
    return null;
  }
  public String toString() {
    return this.getNome() + " " + this.getGrauParentesco();
  }
}
```

Classe das ações realizadas com a lista de dependentes:

Listagem 3: *Classe AcoesParaDependente*

```
public class AcoesParaDependente {
  private Dependente [] lstDependente;

  public void adicionar(Dependente dependente) {
    byte pos = 0;

    if (lstDependente == null) {
      lstDependente = new Dependente[1];
    } else {
      Dependente [] backup = lstDependente;
      lstDependente = new Dependente[lstDependente.length + 1];
      for (byte i = 0; i < backup.length; i++)
        lstDependente[i] = backup[i];
      pos = (byte)backup.length;
    }
    lstDependente[pos] = dependente;
  }
  public void remover(byte posicao) {
    if (lstDependente.length == 1)
      lstDependente = null;
    else {
      Dependente [] backup = lstDependente;
      lstDependente = new Dependente[lstDependente.length - 1];
      byte j = 0;
      for (byte i = 0; i < backup.length; i++)
        if (i != posicao)
          lstDependente[j++] = backup[i];
    }
  }
  public void listar() {
    if (lstDependente != null) {
      System.out.println("Dependentes de Funcionário:");
      for (byte i = 0; i < lstDependente.length; i++)
        System.out.println("- " + lstDependente[i]);
    }
  }
```

```
}
```

4 Aplicação do Padrão

Interface para padronização do padrão:

Listagem 4: *Interface Iterator*

```
interface Iterator {
  public void first();
  public void next();
  public boolean isDone();
  public Object currentItem();
  public int getIndex();
  public byte getLength();
}
```

Classe genérica do padrão para qualquer outra classe:

Listagem 5: *Classe IteratorImpl*

```
class IteratorImpl <T> implements Iterator {
  private T [] items;
  private int index;

  public IteratorImpl(T [] items) {
    this.items = items;
  }
  public void first() {
    index = 0;
  }
  public void next() {
    index++;
  }
  public boolean isDone() {
    return index == items.length;
  }
  public T currentItem() {
    return items[index];
  }
  public int getIndex() {
    return index;
  }
  public byte getLength() {
    return (byte)items.length;
  }
}
```

Nova classe das ações realizadas com a lista de dependentes:

Listagem 6: *Classe AcoesParaDependente*

```
public class AcoesParaDependente {
  private Dependente [] lstDependente;

  public void adicionar(Dependente dependente) {
    byte pos = 0;
    if (lstDependente == null) {
      lstDependente = new Dependente[1];
    } else {
```

```
      IteratorImpl<Dependente> backup = new IteratorImpl<Dependente>(lstDependente);
      lstDependente = new Dependente[lstDependente.length + 1];
      for (backup.first(); !backup.isDone(); backup.next())
        lstDependente[backup.getIndex()] = backup.currentItem();
      pos = backup.getLength();
    }
    lstDependente[pos] = dependente;
  }
  public void remover(byte posicao) {
    if (lstDependente.length == 1) {
      lstDependente = null;
    } else {
      IteratorImpl<Dependente> backup = new IteratorImpl<Dependente>(lstDependente);
      lstDependente = new Dependente[lstDependente.length - 1];
      byte j = 0;
      for (backup.first(); !backup.isDone(); backup.next())
        if (backup.getIndex() != posicao)
      lstDependente[j++] = backup.currentItem();
    }
  }
  public void listar() {
    if (lstDependente != null) {
      System.out.println("Dependentes de Funcionário:");
      Dependente dependente;
      IteratorImpl<Dependente> backup = new IteratorImpl<Dependente>(lstDependente);
      for (backup.first(); !backup.isDone(); backup.next()) {
        System.out.println("- " + backup.currentItem());
      }
    }
  }
}
```

Referências

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Design Patterns. Elements of Reusable Object-Oriented Software 1 ed.* Estados Unidos, Addison-Wesley, 1995, ISBN 0-201-63361-2.