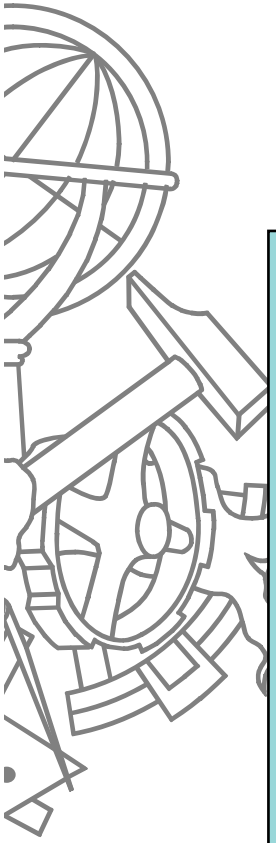
Extensibilidade/flexibilidade

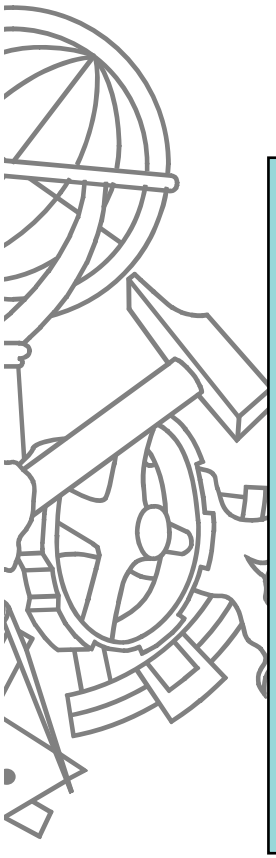
Problema



- Acrescentar capacidades de *logging* a uma classe de acesso a dados já existente

```
public class PessoaAcessoDados
{
    public PessoaAcessoDados() { ... }

    public bool Insert(object r) { ... }

    public bool Delete(object r) { ... }

    public bool Update(object r) { ... }

    public object Load(object id) { ... }
}
```

Problema



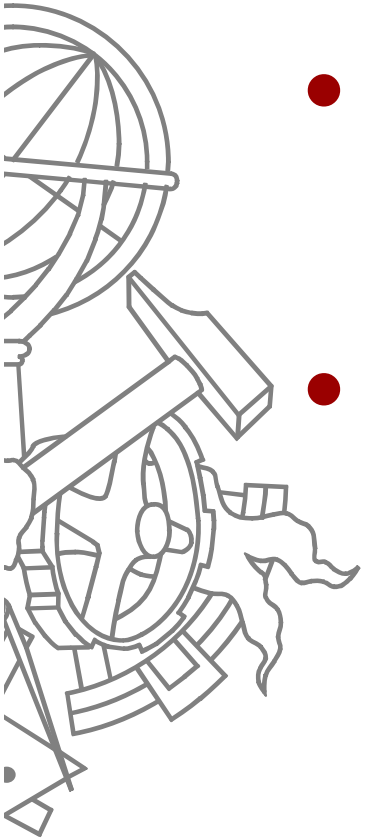
- Tipicamente, criariam subclasse com comportamento de *logging*. (ou modificariam a implementação dos métodos)

```
public class PessoaLogAcessoDados : PessoaAcessoDados
{
      public PessoaLogAcessoDados() { ... }

      public bool Insert(object r) { ... }

      public bool Delete(object r) { ... }

      public bool Update(object r) { ... }

      public object Load(object id) { ... }
}
```

- E se também fosse necessário contar o número de acesso para efeitos de *billing*? E se em certas situações fosse necessário o Log mas não o billing ou vice—versa?

Pattern GoF

Decorator

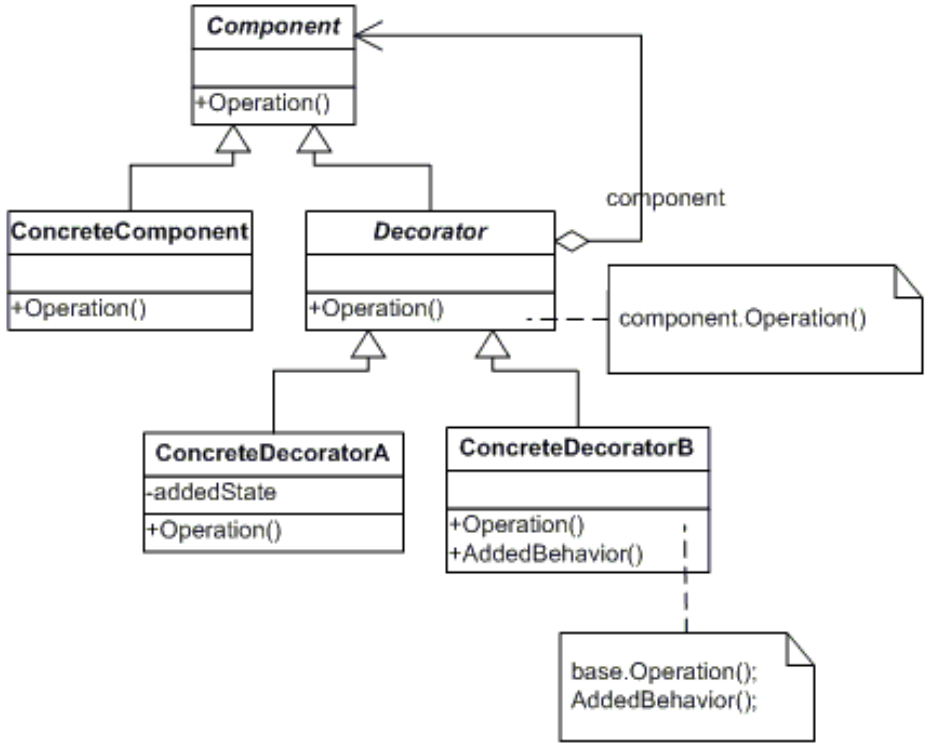


- **Problem:**
  - Allow functionally to be layered around an abstraction, but still dynamically changeable.
- **Solution:**
  - Combine inheritance and composition. By making an object that both subclasses from anther class and holds an instance of the class, can add new behavior while referring all other behavior to the original class.

Pattern

Decorator

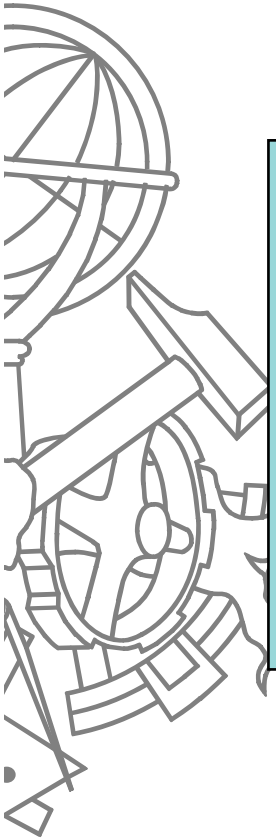
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Pattern

Exemplo (C#)

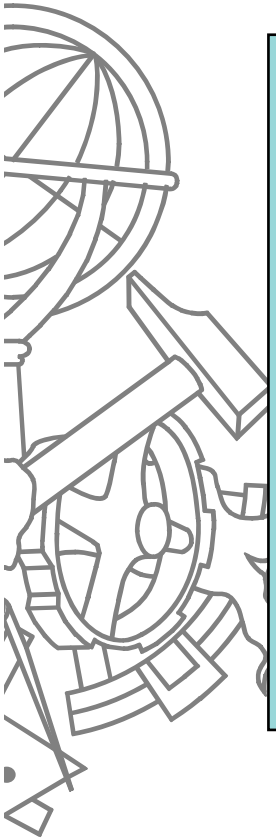


- Começar por extrair interface

```
public interface IAcessoDados
{
     public bool Insert(object r);
     public bool Delete(object r);
     public bool Update(object r);
     public object Load(object id);
}
```

Pattern GoF

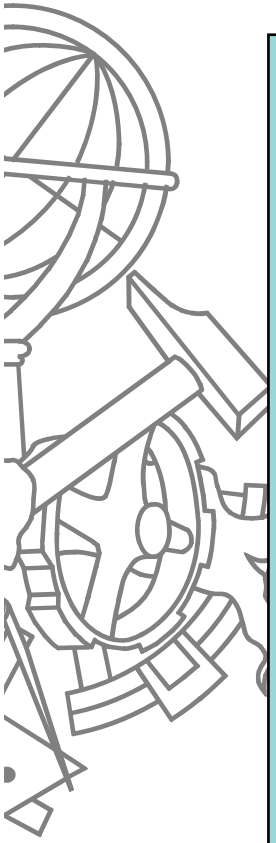
Exemplo (C#)



```
public class PessoaAcessoDados : IAcessoDados
{
    public PessoaAcessoDados() { ... }

    public bool Insert(object r) { ... }

    public bool Delete(object r) { ... }

    public bool Update(object r) { ... }

    public object Load(object id) { ... }
}
```

Pattern GoF

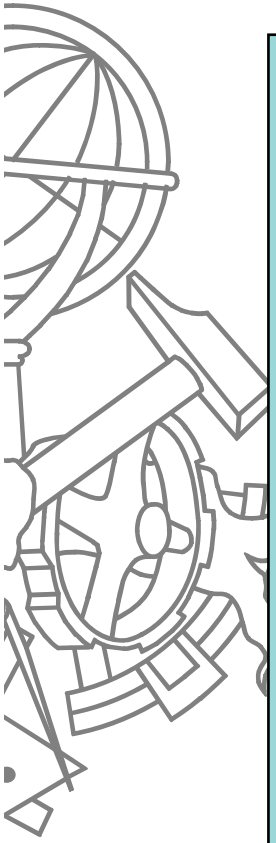
Exemplo (C#)



```
public class LoggingDecorator : IAcessoDados
{
      IAcessoDados componente;
      public LoggingDecorator(IAcessoDados componente) {
            this.componente = componente;
      }

      public bool Insert(object r) {
            WriteLog("Insert", r);
            return componente.Insert(r);
      }

      public bool Delete(object r) { ... }
      public bool Update(object r) { ... }
      public object Load(object id) { ... }

      private void WriteLog(string op, object parms)
      { ... }
}
```

Pattern GoF

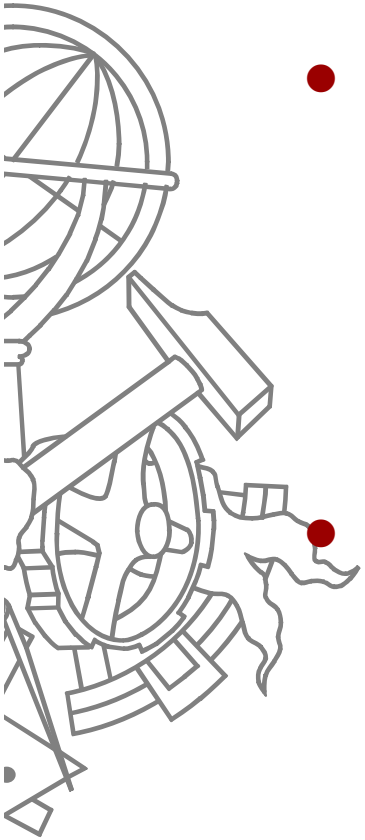
Exemplo (C#)



```
public class TesteDecorator
{
     public void Teste()
     {
          IAcessoDados da = new PessoaAcessoDados();
          IAcessoDados dec = new LoggingDecorator(da);

          ...

          dec.Insert(...);
          ...
     }
}
```

Pattern GoF

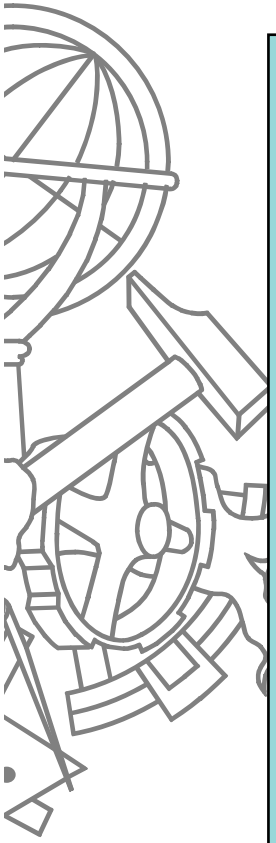
Decorator



- Como a classe **`Decorator`** implementa a mesma interface do **`Component`**, pode ser usada em qualquer lugar do programa que necessite de um objecto **`Component`**
- Se usássemos herança não conseguiríamos resolver cenários em que necessitássemos apenas de *Logging* ou apenas de contagem ou de ambos
  - Mas é possível encadear **`Decorator`**s!

Pattern GoF

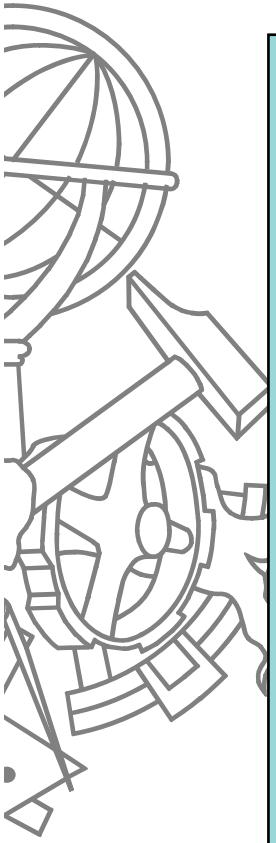
Exemplo (C#)



```
public class CounterDecorator : IAcessoDados
{
      int nAcessos = 0;

      IAcessoDados componente;
      public CounterDecorator(IAcessoDados componente) {
            this.componente = componente;
      }
      public bool Insert(object r) {
            nAcessos++;
            return componente.Insert(r);
      }

      public bool Delete(object r) { ... }
      public bool Update(object r) { ... }
      public object Load(object id) { ... }

      public int NumAcessos { get { return nAcessos; } }
}
```

Pattern GoF

Exemplo (C#)



```
public class BillingDAL
{
     public void Teste()
     {
          IAcessoDados da = new PessoaAcessoDados();
          IAcessoDados dec = new LoggingDecorator(da);
          IAcessoDados cd = new CounterDecorator(dec);

          ...

          cd.Insert(...);

          ...

          CounterDecorator bil = (CounterDecorator)cd;
          float custo = bil.NumAcessos * PRICE_PER_OP;
          ...
     }
}
```