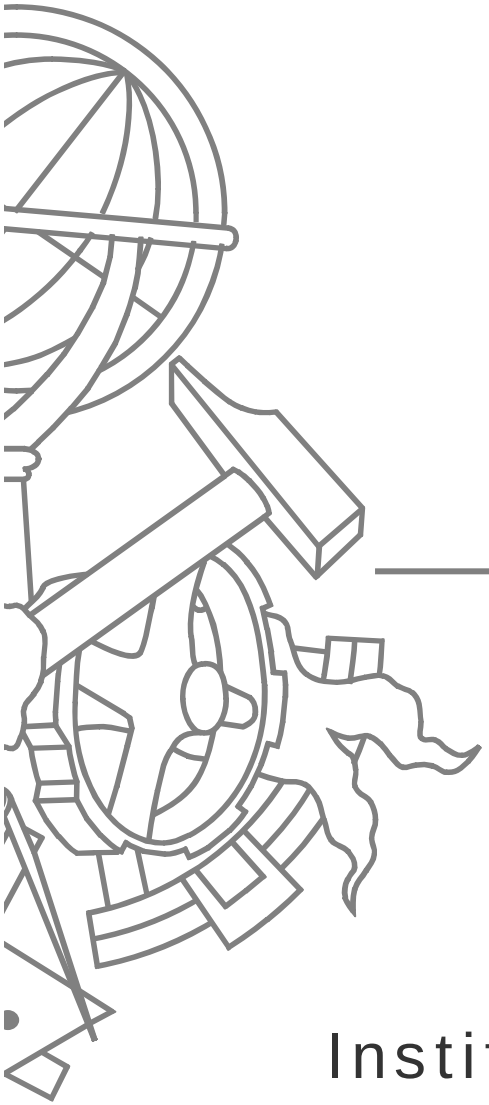


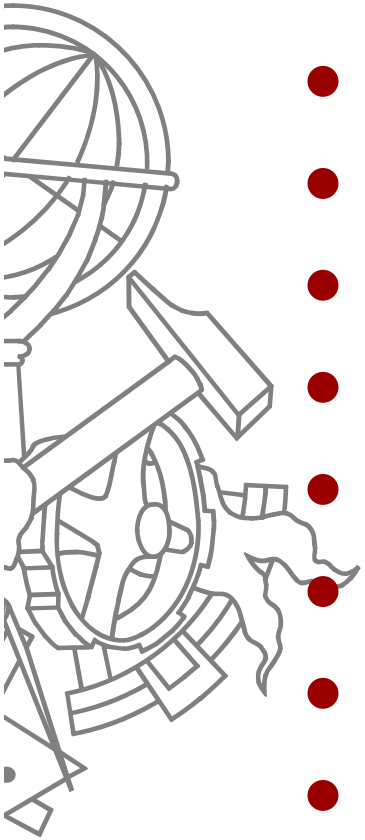
Padrões GoF

Paulo Sousa

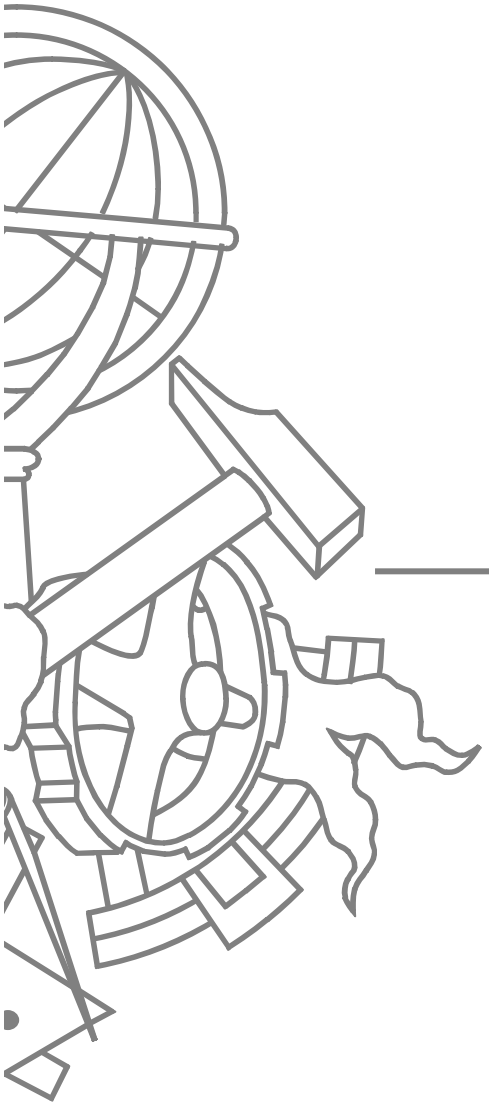
Engenharia da Informação

Instituto Superior de Engenharia do Porto

Padrões base

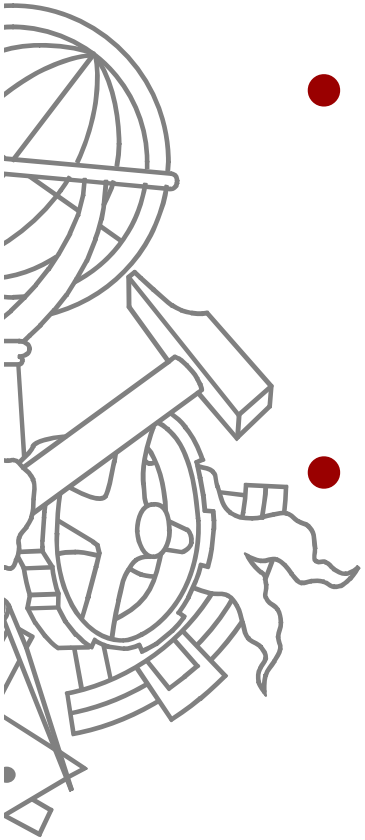


- Factory Method [GoF]
- Adapter [GoF]
- Strategy [GoF]
- Decorator [GoF]
- Singleton [GoF] [ESP]
- Monostate
- Observer [GoF]
- Memento [GoF]
- Façade [GoF]



Factory Method

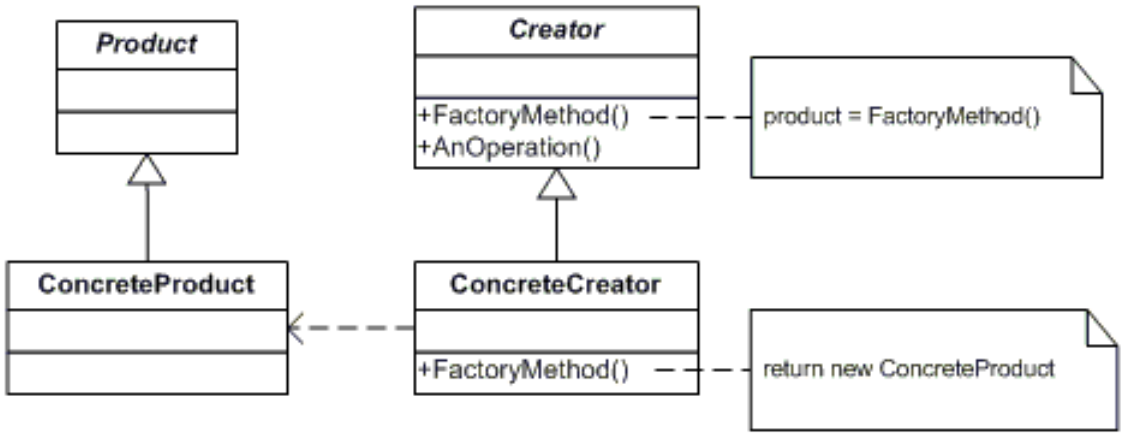
Factory Method



- **Problem:**
  - How do you simplify the manipulation of many different implementations of the same interface.
- **Solution:**
  - Hide creation within a method, have the method declare a return type that is more general than its actual return type.

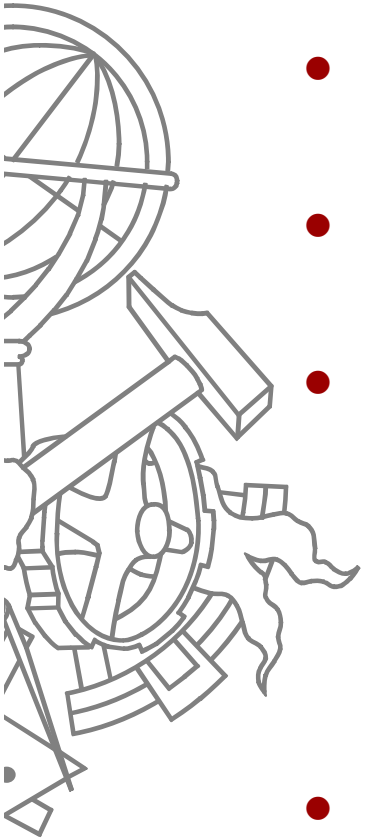
Factory Method

- A method that hides the details of creating an object



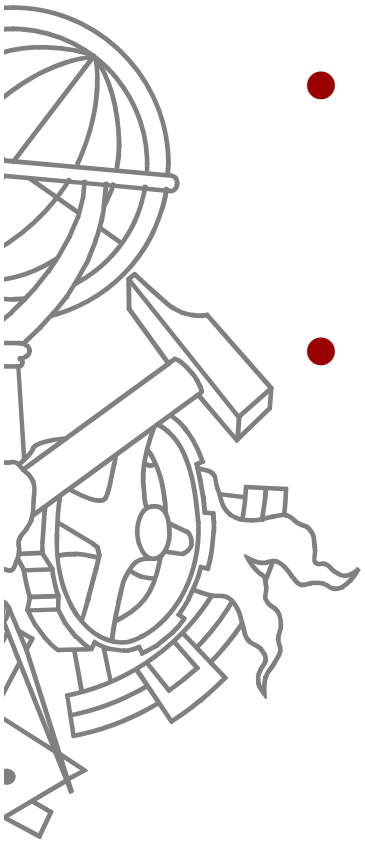
fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Participantes



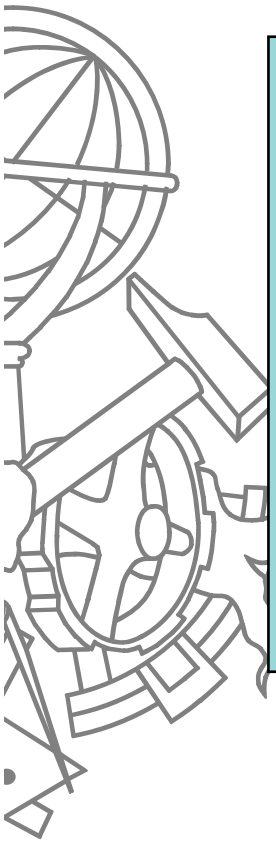
- **`Product`**
  - defines the interface of objects the factory method creates
- **`ConcreteProduct`**
  - implements the **`Product`** interface
- **`Creator`**
  - declares the factory method, which returns an object of type **`Product`**. Creator may also define a default implementation of the factory method that returns a default **`ConcreteProduct`** object.
  - may call the factory method to create a **`Product`** object.
- **`ConcreteCreator`**
  - overrides the factory method to return an instance of a **`ConcreteProduct`**.

Factory Method



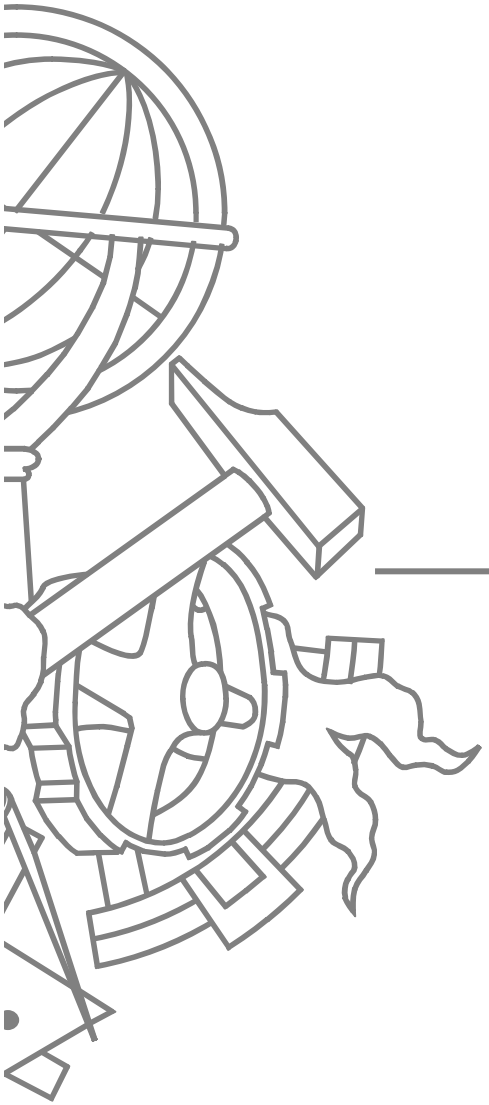
- Permite centralizar num único ponto a criação de objectos de um dado tipo (ou subclasses desse tipo)
- Isola a criação da classe da sua utilização
  - Permite controlar o processo de criação (ex., logging)
  - Permite modificar o objecto concreto que se quer criar
  - Ex., consoante o tipo de SGDB definido na configuração da aplicação, criar um componente de acesso a dados de um tipo ou outro

Exemplo (java)



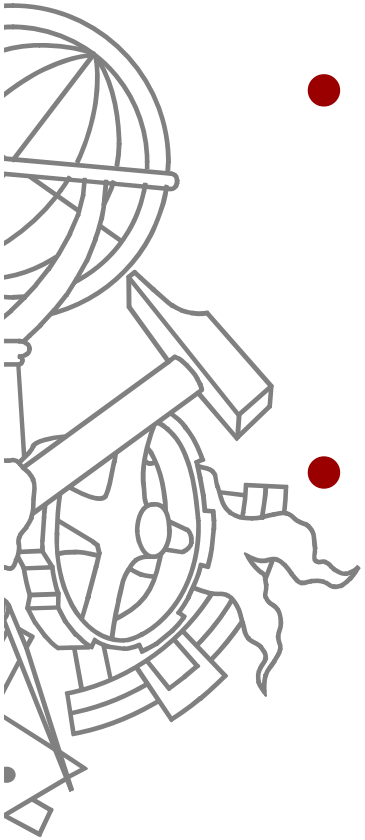
```
class SortedList {
   ...
   Enumerator elements ()
      { return new SortedListEnumerator(); }
   ...
   private class SortedListEnumerator implements
      Enumerator { ... }
}
```

- The method is the “factory”' in the name. Users don't need to know the exact type the factory returns, only the declared type.
- The factory could even return different types, depending upon circumstances.



Adapter

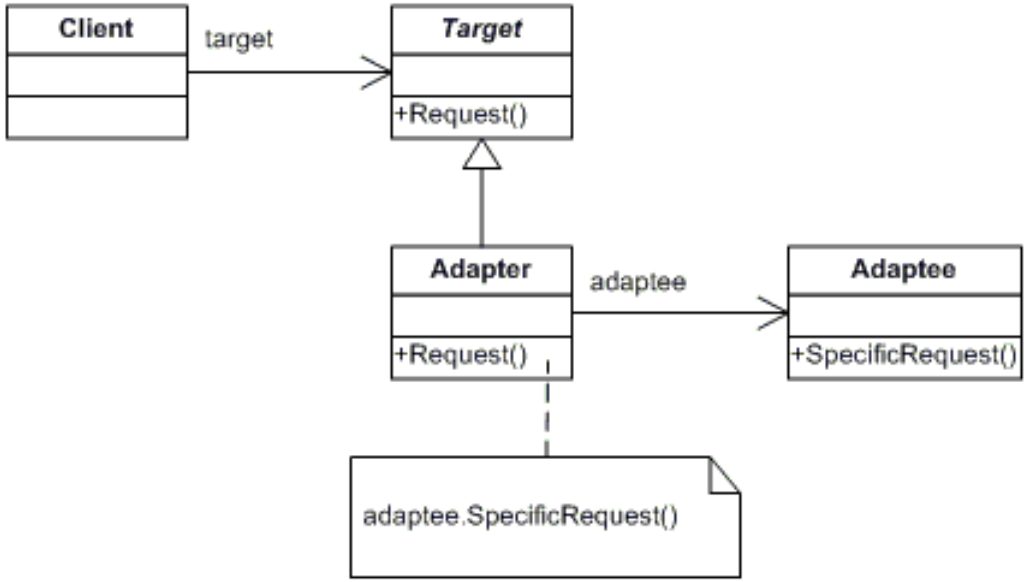
Adapter



- **Problema:**
  - Como ligar um cliente a um fornecedor de serviços quando a interface fornecida é diferente da esperada
- **Solução:**
  - Criar uma classe que forneça a interface esperada pelo cliente e que utilize a interface do fornecedor de serviço

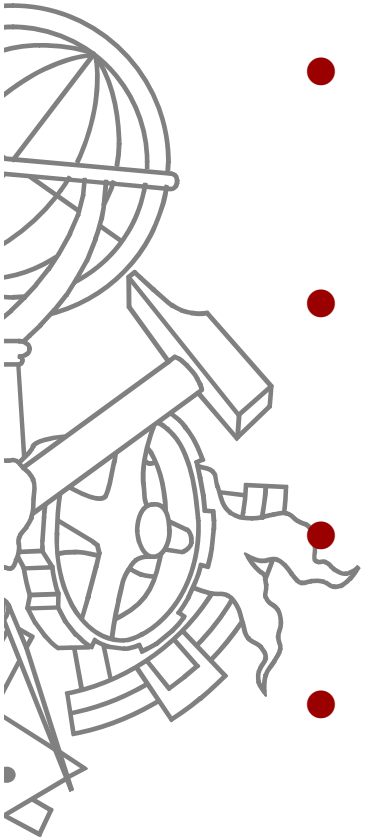
Adapter

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



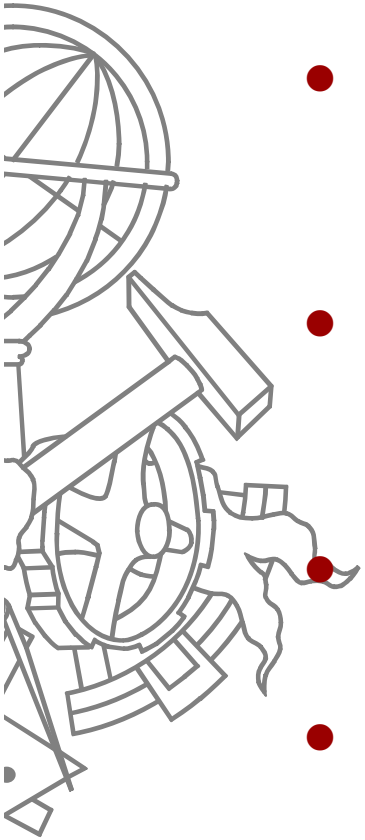
fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Participantes



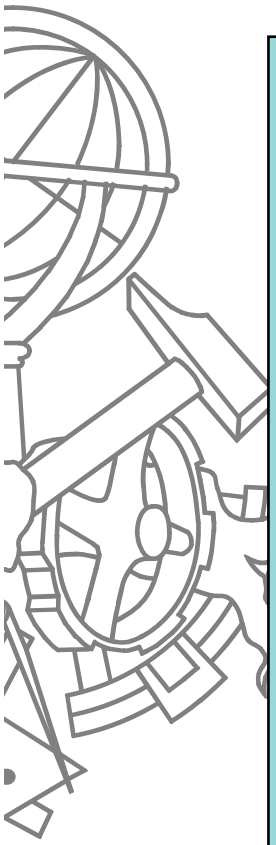
- **`Target`**
  - defines the domain-specific interface that **`Client`** uses.
- **`Adapter`**
  - adapts the interface **`Adaptee`** to the **`Target`** interface.
- **`Adaptee`**
  - defines an existing interface that needs adapting.
- **`Client`**
  - collaborates with objects conforming to the **`Target`** interface.

Adapter

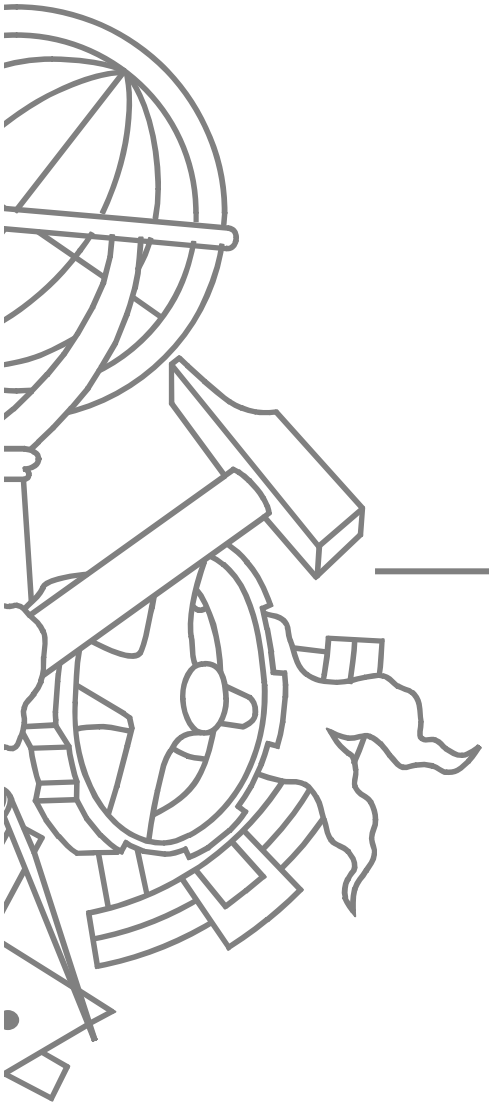


- An adapter is used to connect a client (an object that needs a service) with a server (an object that provides the service).
- The client requires a certain interface, and while the server provides the necessary functionality, it does not support the interface.
- The adapter changes the interface, without actually doing the work.
- Adapters are often needed to connect software from different vendors.

Exemplo (Java)

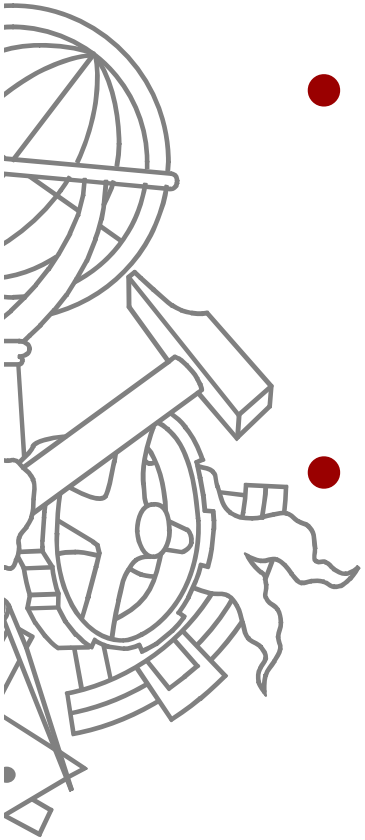


```
class MyCollection implements Collection {
   // DataBox is some collection that does not support the
      Collection interface
   private DataBox data = new DataBox();

   public boolean isEmpty () { return data.count() == 0; }

   public int size ()  { return data.count(); }

   public void addElement (Object newElement)
      {data.add(newElement); }

   public boolean containsElement (Object test)
      { return data.find(test) != null; }

   public Object findElement (Object test)
      { return data.find(test); }
}
```



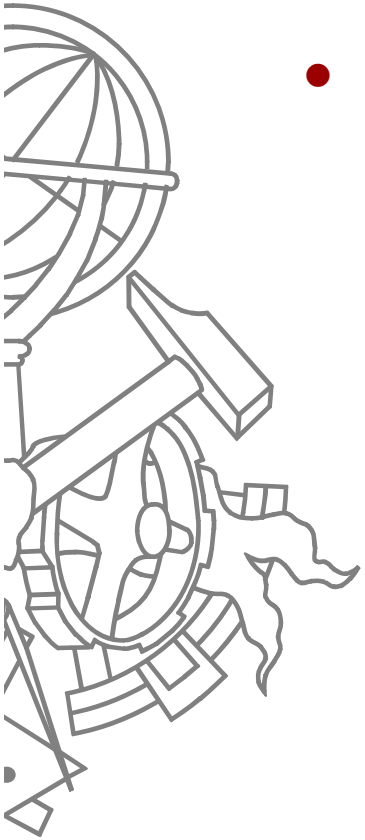
Strategy

Strategy

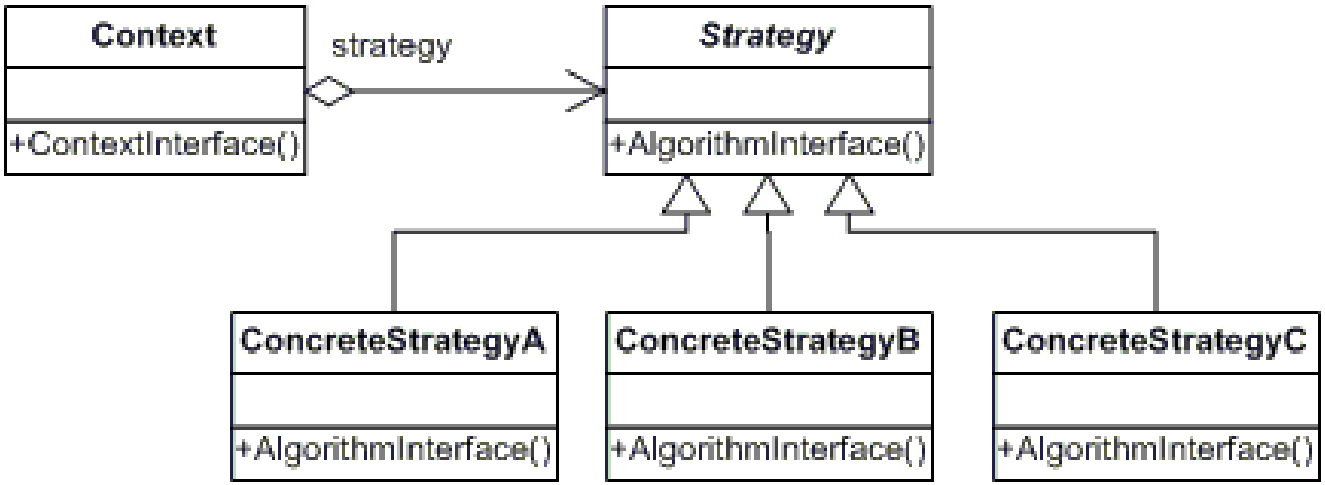


- **Problem:**
  - Allow the client the choice of many alternatives, but each is complex, and you don't want to include code for all.
- **Solution:**
  - Make many implementations of the same interface, and allow the client to select one and give it back to you.

Strategy

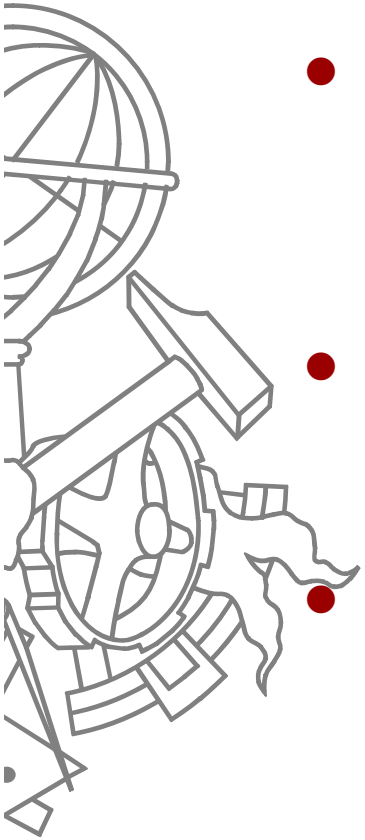


- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



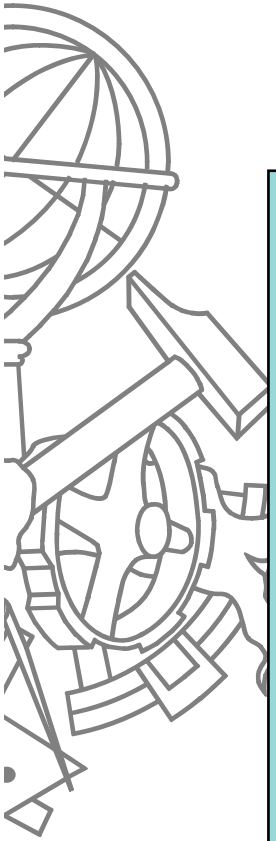
fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Participantes



- **`Strategy`**
  - declares an interface common to all supported algorithms. **`Context`** uses this interface to call the algorithm defined by a **`ConcreteStrategy`**
- **`ConcreteStrategy`**
  - implements the algorithm using the **`Strategy`** interface
- **`Context`**
  - is configured with a **`ConcreteStrategy`** object
  - maintains a reference to a **`Strategy`** object
  - may define an interface that lets **`Strategy`** access its data.

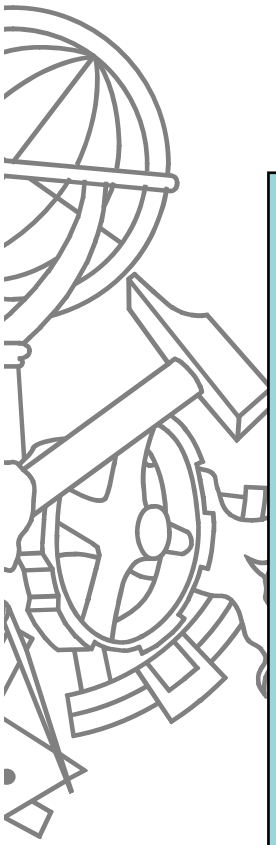
Exemplo (C#)



- Uma colecção de elementos pode implementar diversos algoritmos (estratégias) de ordenação

```
interface SortStrategy {
   void Sort(Coleccao obj);
}

class Coleccao {
   ...
   private SortStrategy theStrategy;
   public Coleccao(SortStrategy aStrategy) {
      ...
      theStrategy = aStrategy;
   }
   public void Sort() {
      theStrategy.Sort(this);
   }
}
```

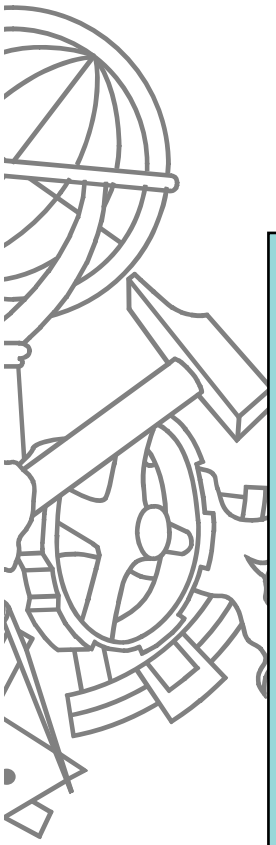
Exemplo (C#)



- Implementar cada uma das estratégias

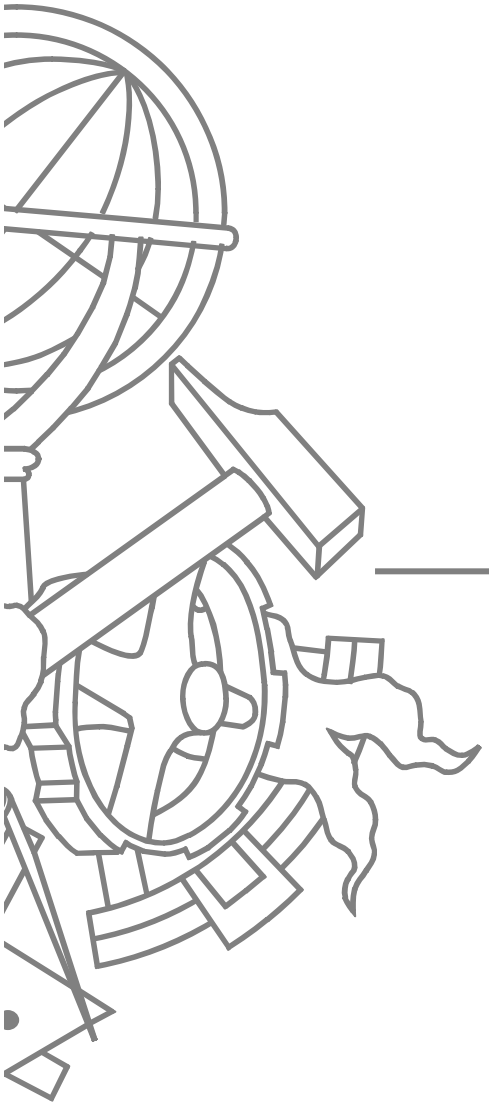
```
class QuickSort : SortStrategy
{
   public void Sort(Coleccao obj) { ... }
}

class MergeSort : SortStrategy
{
   public void Sort(Coleccao obj) { ... }
}

class ShellSort : SortStrategy
{
   public void Sort(Coleccao obj) { ... }
}
```

Exemplo (C#)

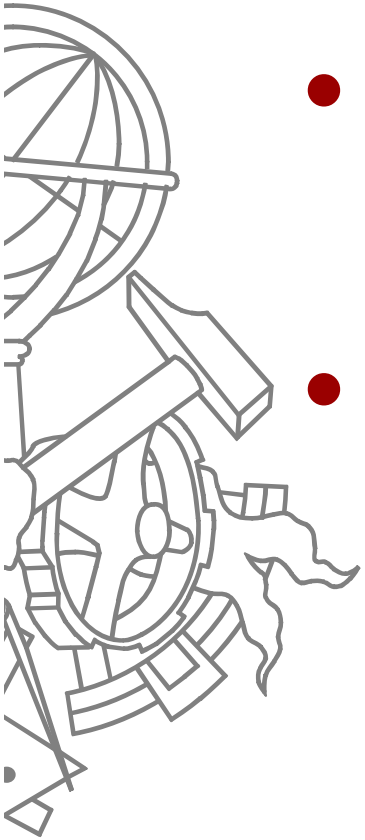


- Ao criar instâncias da colecção indicar qual a estratégia a utilizar

```
class TesteColeccao
{
   ...
   public void teste() {
      Coleccao c1 = new Coleccao(new QuickSort());
      C1.Sort();
      ...
      Coleccao c2 = new Coleccao(new MergeSort());
      C2.sort();
      ...
   }
}
```



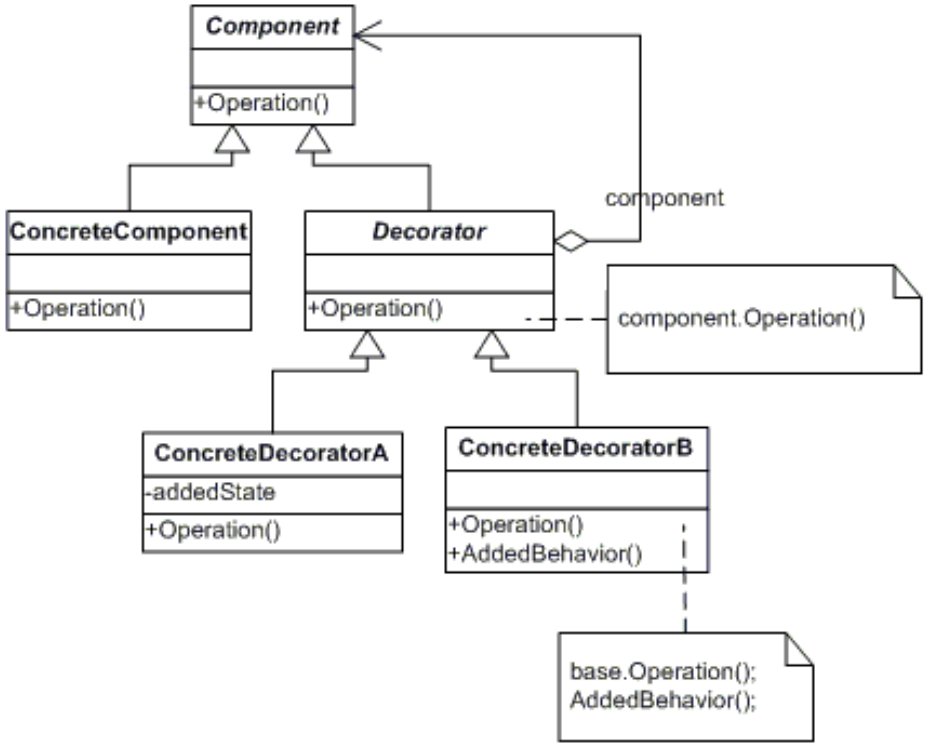
Decorator



- **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.

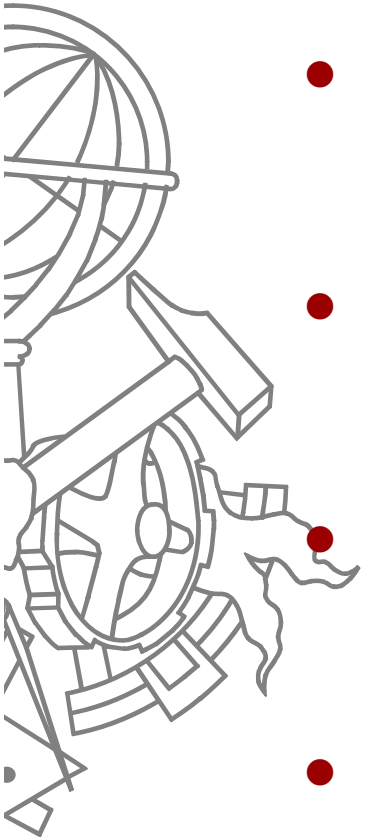
Decorator

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



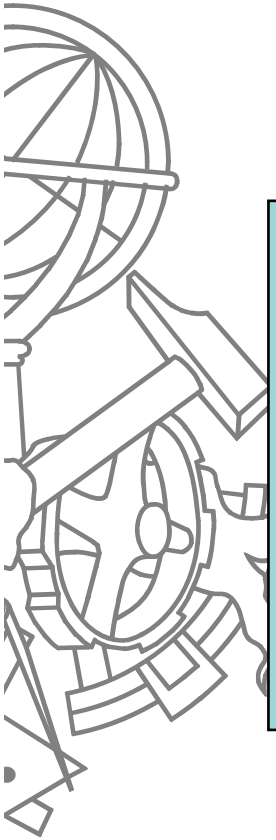
fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Participantes



- **`Component`**
  - defines the interface for objects that can have responsibilities added to them dynamically.
- **`ConcreteComponent`**
  - defines an object to which additional responsibilities can be attached.
- **`Decorator`**
  - maintains a reference to a **`Component`** object and defines an interface that conforms to **`Component`**'s interface.
- **`ConcreteDecorator`**
  - adds responsibilities to the component.

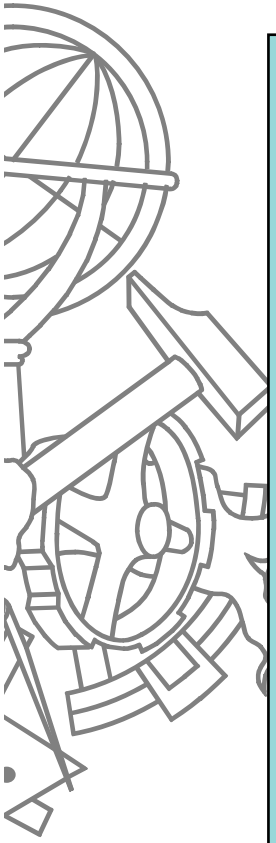
Exemplo (C#)



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

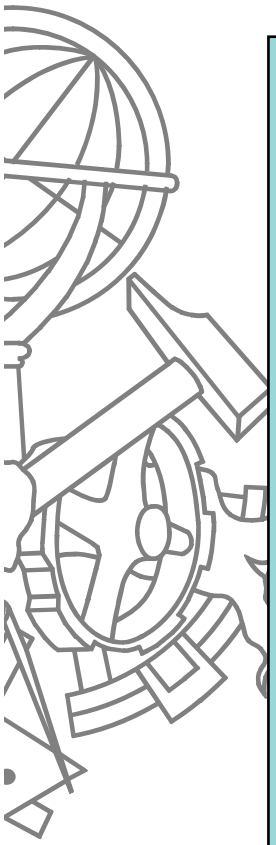
```
public interface IAcessoDados
{
     public bool Insert(object r);
     public bool Delete(object r);
     public bool Update(object r);
     public object Load(object id);
}
```

Exemplo (C#)



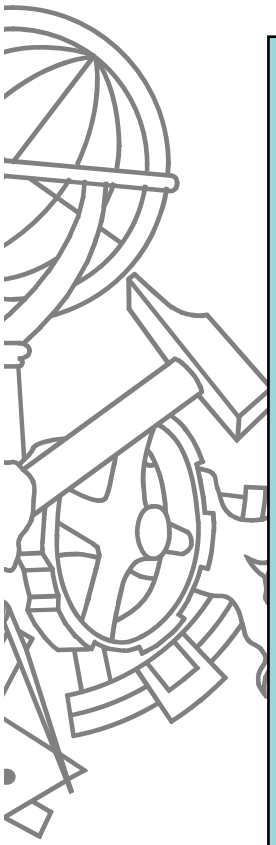
```
public class PessoaAcessoDados : IAcessoDados
{
     public PessoaAcessoDados() { ... }

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

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

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

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

Exemplo (C#)



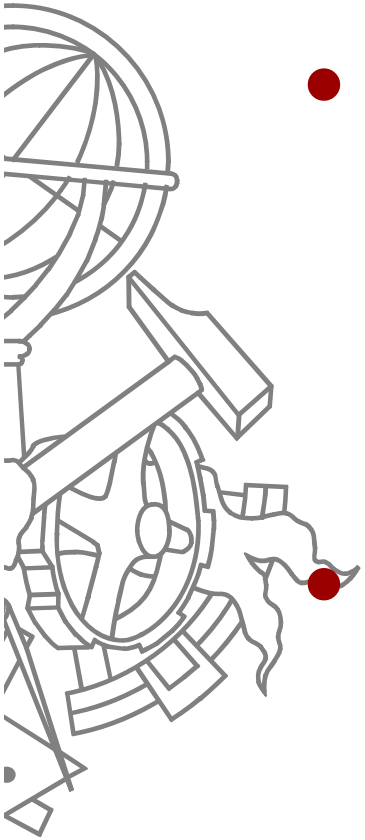
```
public class LoggingDecorator : IAcessoDados
{
      IAcessoDados componente;
      public LoggingDecorator(IAcessoDados componente) {
            this.componente = componente;
      }

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

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

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

Exemplo (C#)



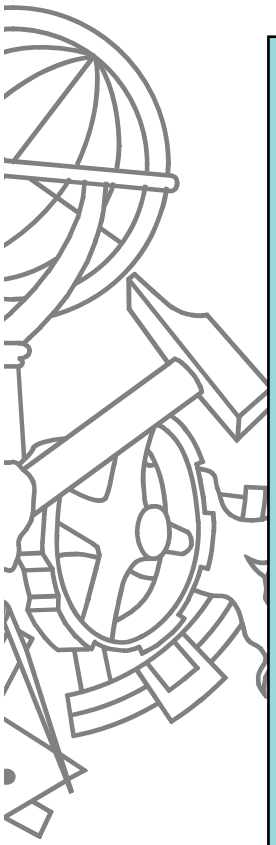
```
public class TesteDecorator
{
     public void Teste()
     {
          IAcessoDados da = new PessoaAcessoDados();
          IAcessoDados dec = new LoggingDecorator(da);

          ...

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

Decorator



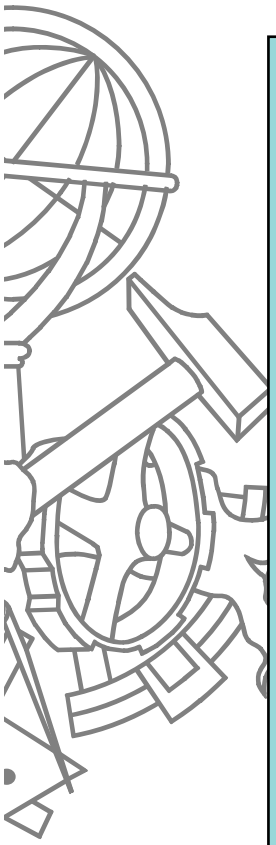
- Como a classe **`Decorator`** implementa a mesma interface do **`Component`**, pode ser usada em qualquer lugar do programa que necessite de um objecto **`Component`**
- É possível encadear **`Decorator`**s!

Exemplo (C#)

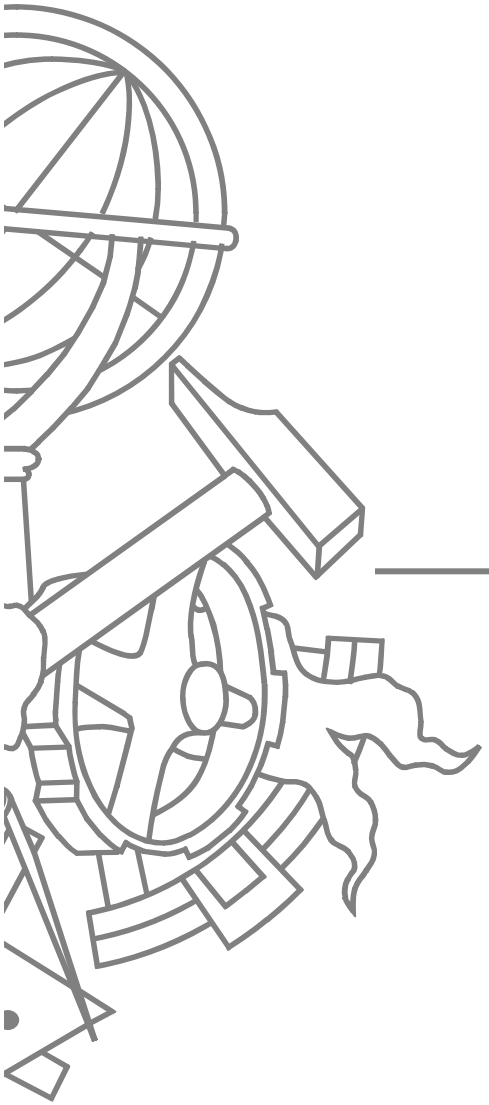


```
public class CounterDecorator : IAcessoDados
{
     int nAcessos = 0;

     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; } }
}
```

Exemplo (C#)

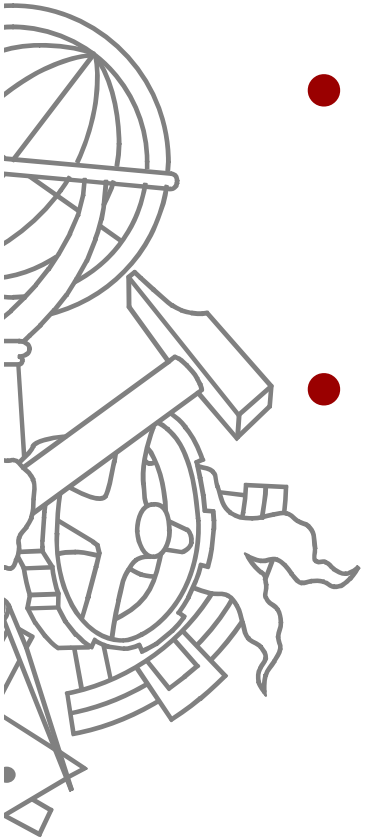


```
public class BillingDAL
{
     public void Teste()
     {
          IAcessoDados da = new PessoaAcessoDados();
          IAcessoDados dec = new LoggingDecorator(da);
          IAcessoDados bil = new CounterDecorator(dec);

          ...

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



Singleton

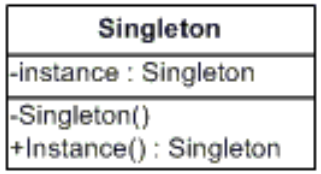
Singleton



- **Problem:**
  - You want to ensure that there is never more than one instance of a given class.
- **Solution:**
  - Make the constructor private, have a method that returns just one instance, which is held inside the class itself.

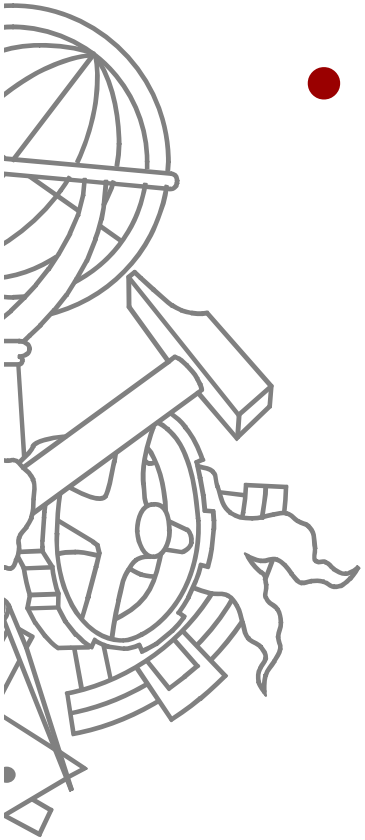
Singleton

- Ensure a class has only one instance and provide a global point of access to it.



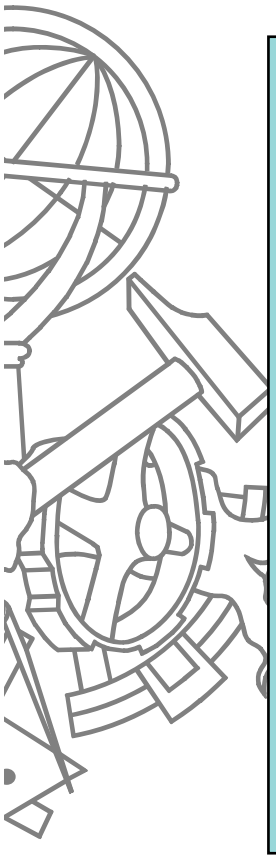
fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Participantes



- **`Singleton`**
  - defines an operation named **`Instance`** that lets clients access its unique instance. **`Instance`** is a class operation.
  - responsible for creating and maintaining its own unique instance.

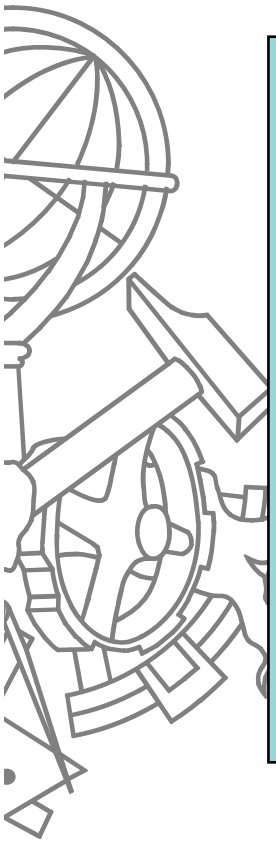
Exemplo (Java)



```
public class final Singleton {
     private static Singleton theInstance = null;

     public static Singleton instance() {
        if( theInstance == null ) {
                theInstance = new Singleton();
        }
        return theInstance;
     }

     public void func() { ... }
}
```

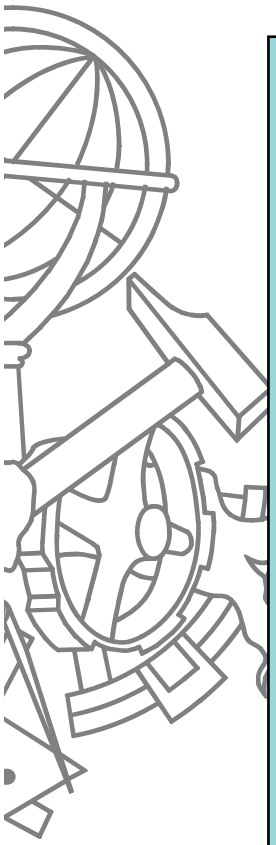
- Esta implementação **não** é *thread-safe*

Exemplo (Java)



```
class TesteSingleton {
   public void teste() {
      ...
      Singleton.instance.func();
      ...
   }
   ...
}
```

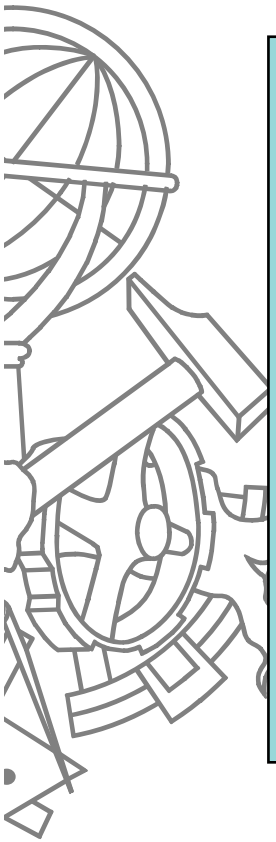
Exemplo (C#)



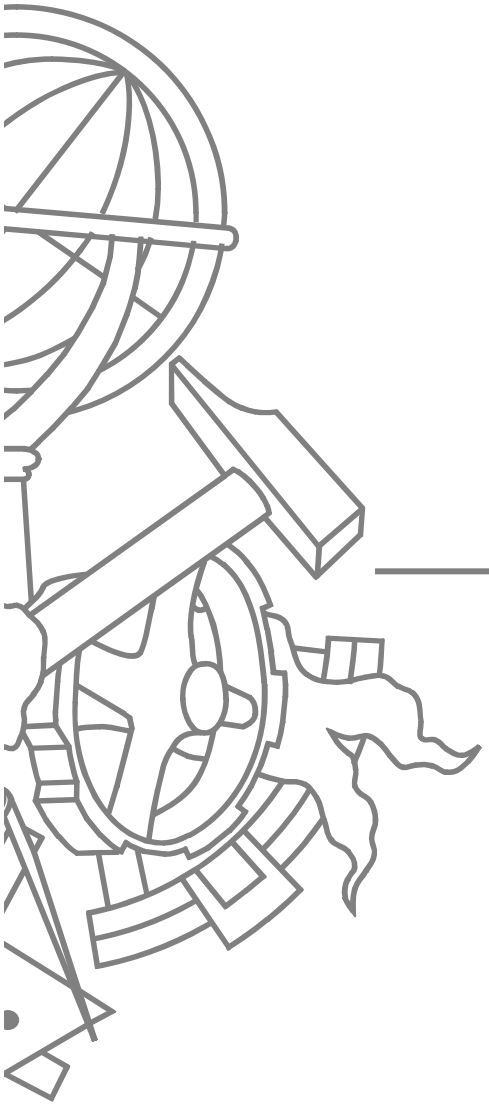
```
public class sealed Singleton
{
   private Singleton() { ... }

   private static readonly Singleton theOne = new
      Singleton();

   public static Singleton Instance
   {
      get { return theOne; }
   }

   public void Func() { ... }

   ...
}
```

- Palavra reservada **`readonly`** garante *thread-safety*

Exemplo (C#)



```
class TesteSingleton
{
   public void Teste()
   {
      ...
      Singleton.Instance.Func();
      ...
   }
   ...
}
```

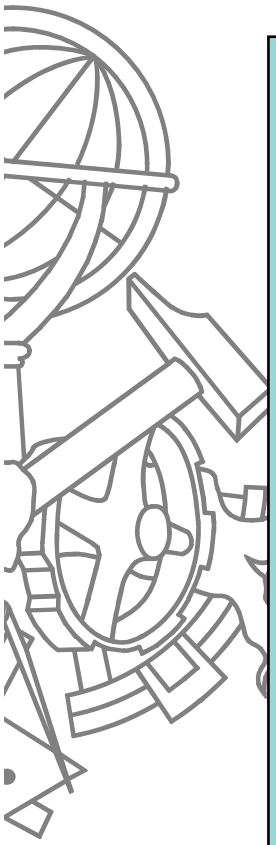


Monostate

Monostate

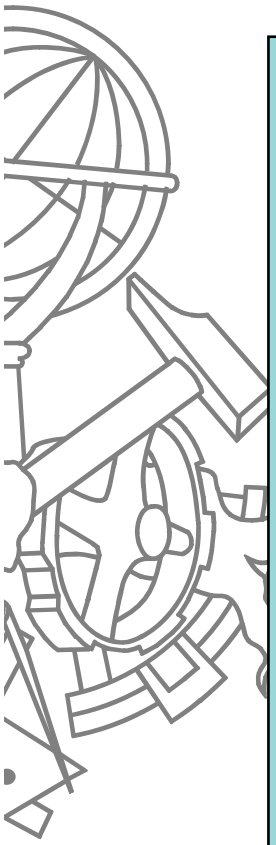
- Muito semelhante ao *Singleton*, mas ao contrário de existir apenas uma instância, todas as instâncias partilham o mesmo estado
  - Atributos da classe são todos *static*

Exemplo (C#)



```
public class MonostateClass {
   private static int Atributo1;
   private static string Atributo2;

   public MonostateClass () { ... }
   ...
   public int Attr1
   {
   get { return Atributo1; }
   set { Atributo1 = value; }
   }
}
```

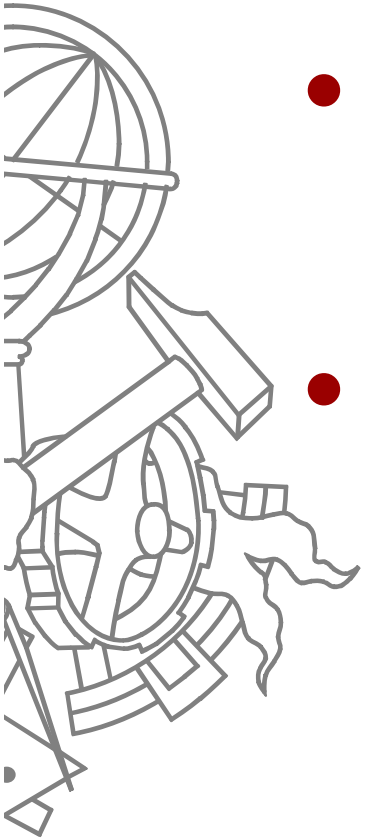
Exemplo (C#)



```
class TesteMonostate {
   public void teste() {
      ...
      MonostateClass x1 = new MonostateClass();
      MonostateClass x2 = new MonostateClass();

      x1.Attr1 = 56;
      Console.Writeln(x2.Attr1);
      // vai imprimir 56
      ...
   }
   ...
}
```

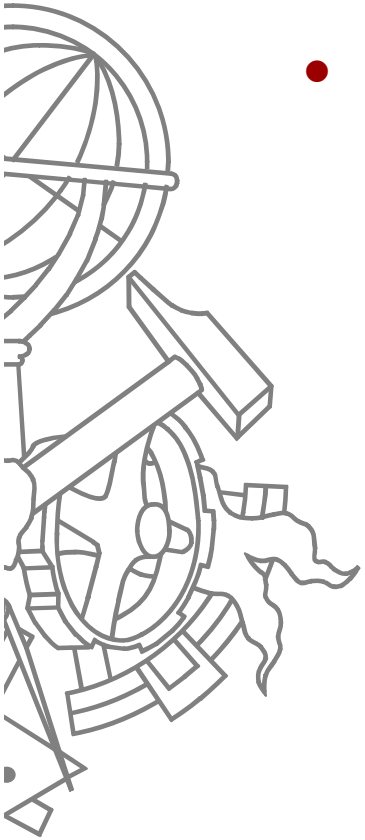
Observer (publish/subscribe)

Observer

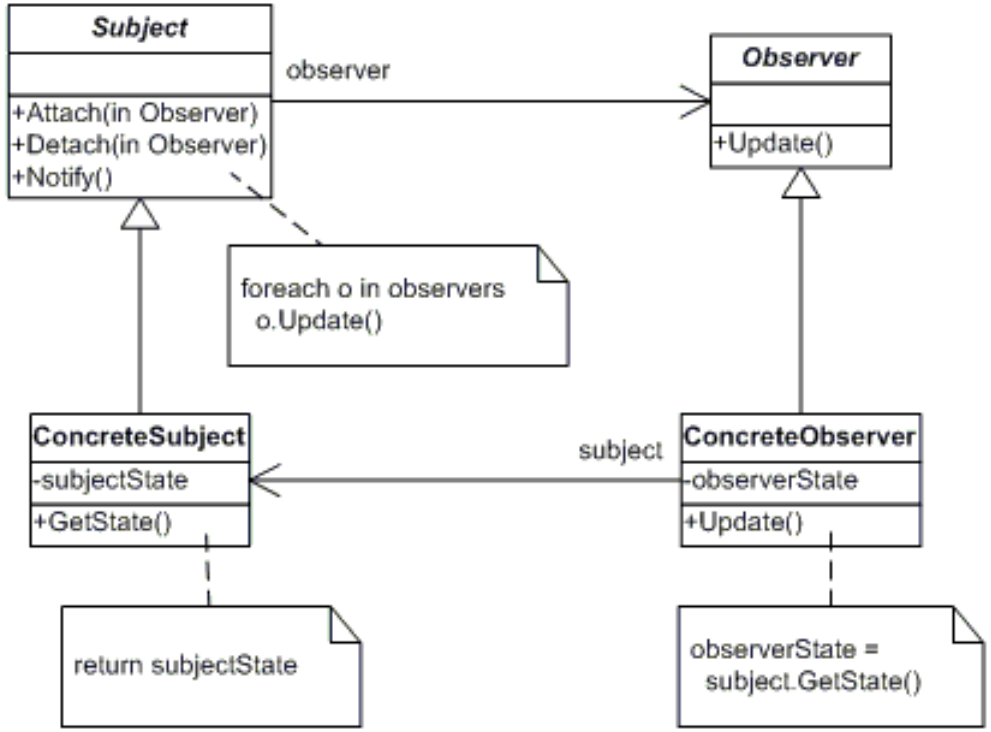


- **Problem:**
  - How do you dynamically (at run time) add and remove connections between objects.
- **Solution:**
  - Fonecer um mecanismo de subscrição de notificações para que um objecto possa ser avisado de alterações noutro

Observer

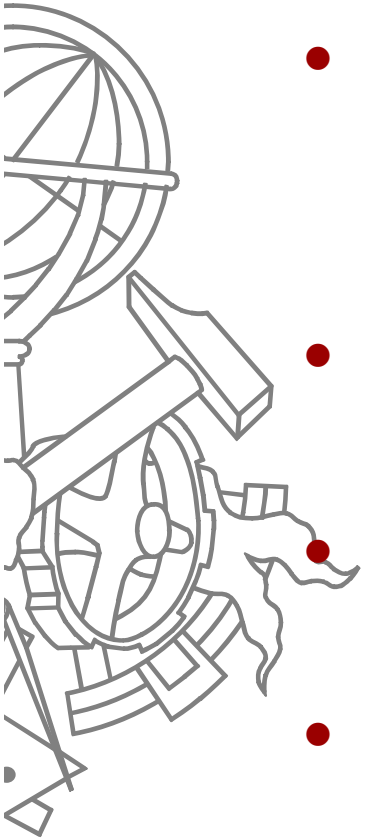


- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically



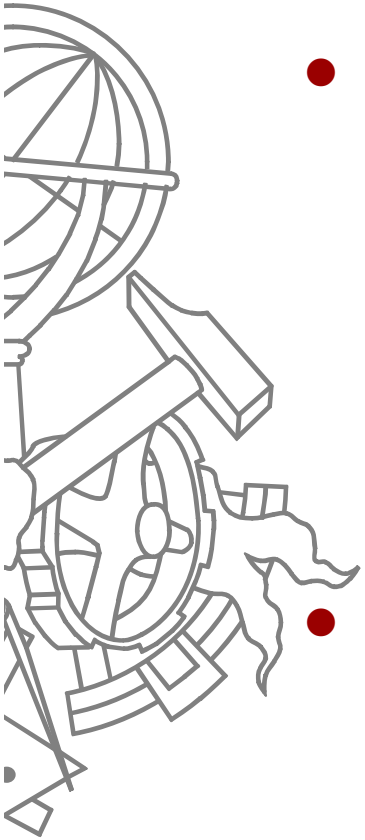
fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Participantes



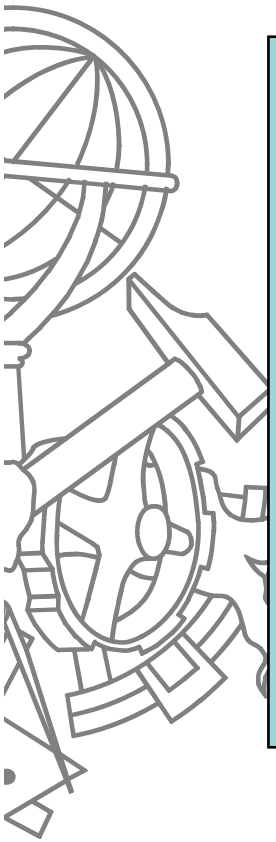
- **`Subject`**
  - knows its observers. Any number of **`Observer`** objects may observe a subject
  - provides an interface for attaching and detaching **`Observer`** objects.
- **`ConcreteSubject`**
  - stores state of interest to **`ConcreteObserver`**
  - sends a notification to its observers when its state changes
- **`Observer`**
  - defines an updating interface for objects that should be notified of changes in a subject.
- **`ConcreteObserver`**
  - maintains a reference to a **`ConcreteSubject`** object
  - stores state that should stay consistent with the subject's
  - implements the **`Observer`** updating interface to keep its state consistent with the subject's

Suporte na plataforma Java



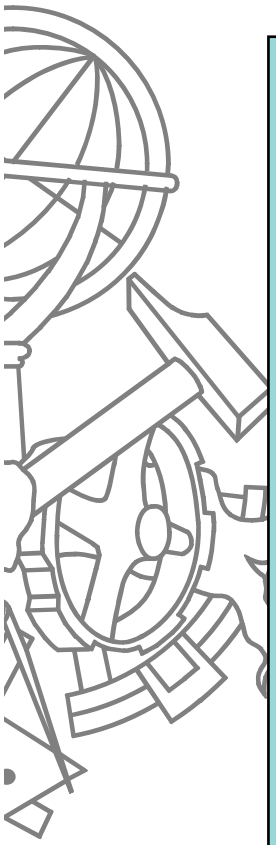
- Classe **`Observable`**
  - As classes que “publicam” devem ser derivadas desta classe que implementa o comportamento standard para adicionar vistas e notificar todos os “subscritores”
  - **`void addObserver(Observer o)`**
  - **`void setChanged()`**
  - **`void notifyObservers(Object arg)`**
- Interface **`Observer`**
  - As classes “subscritor” devem implementar esta interface contendo um método que será invocado quando o “publicador” for actualizado
  - **`void update(Observable o, Object arg)`**

Exemplo (Java)



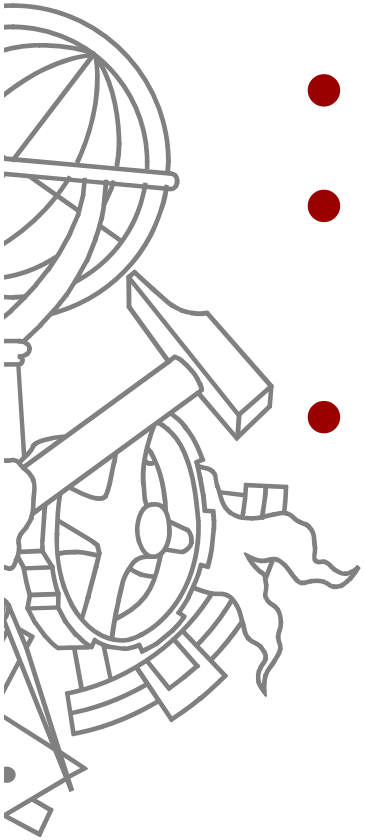
```
public class Subject extends Observable
{
     public void doSomeAction()
     {
          ...
          Integer p = new Integer(123);
          setChanged();
          notifyObservers(p);
          ...
     }
}
```

Exemplo (Java)



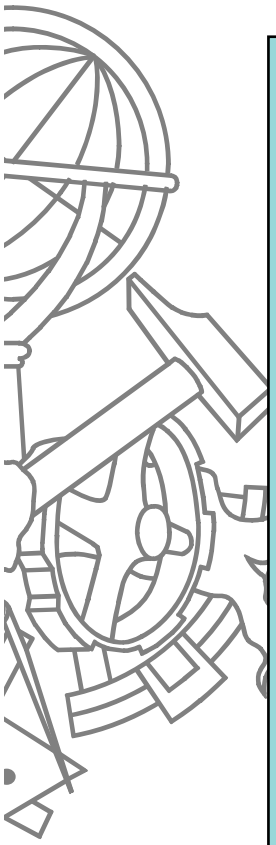
```
public class MyObserver implements Observer
{
     private void update(Observable sender, Object parm)
     {
          ...
     }

     public void doWhatever(Subject x)
     {
          ...
          // registar interesse nas notificações
          x.addObserver(this);
          ...
     }
}
```

Suporte da plataforma .Net



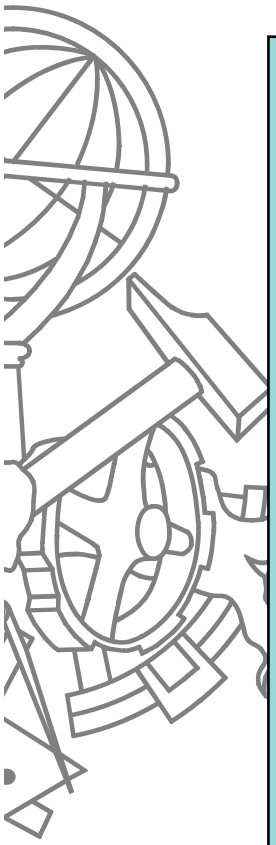
- Eventos e *delegates*
- Classes “subject” declaram um *delegate* e um evento sobre esse *delegate*
- Classes “observer” implementam um método que respeite o *delegate* e associam-se ao evento

Exemplo (.Net)

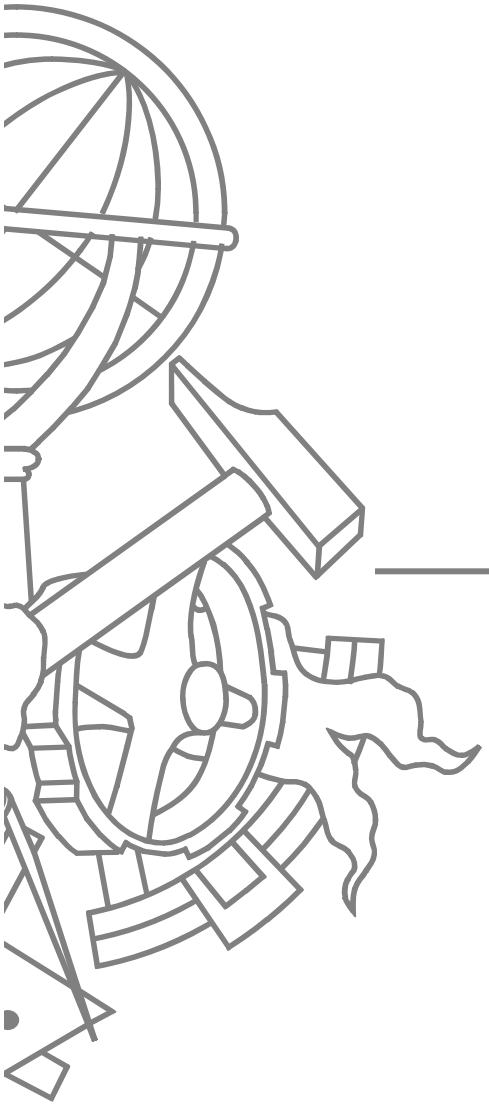


```
public class MySubject
{
     public delegate void SomeEventToObserve(int parm);

     public event SomeEventToObserve OnSomeEvent;

     public void DoSomeAction()
     {
              ...
              int p = 123;
              ...
              // notificar todos os observadores
              OnSomeEvent(p);
              ...
     }

}
```

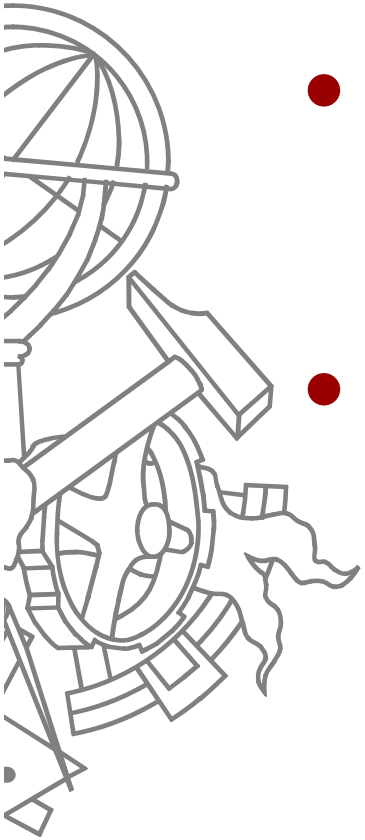
Exemplo (.Net)



```
public class MyObserver1
{
      private void SomeEventCallback(int x)
      {
            ...
      }

      public void DoWhatever(Subject x)
      {
            ...
            // registar interesse no evento
            x.OnSomeEvent += new
Subject.SomeEventToObserve(SomeEventCallback);
            ...
      }

}
```



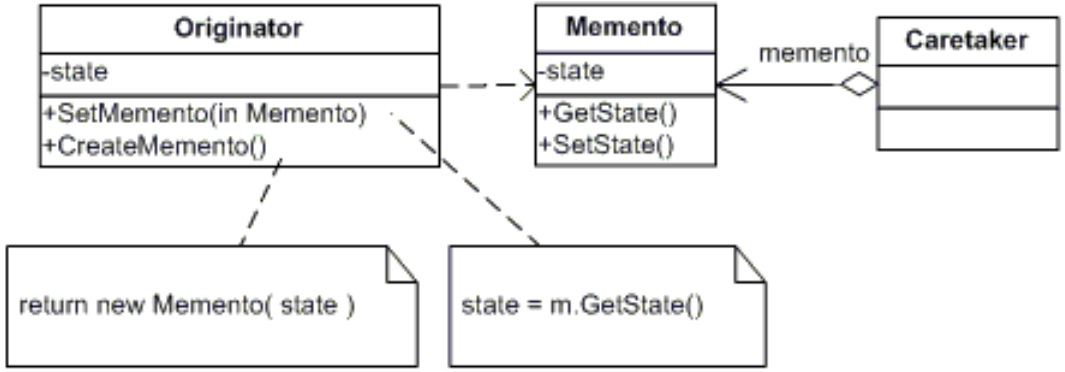
Memento



- **Problema:**
  - Como fazer um objecto “regressar” a um estado anterior
- **Solução:**
  - Fornecer um mecanismo para que o objecto possa criar representações do seu estado interno (*snapshot*) que possam mais tarde ser recuperadas pelo próprio objecto

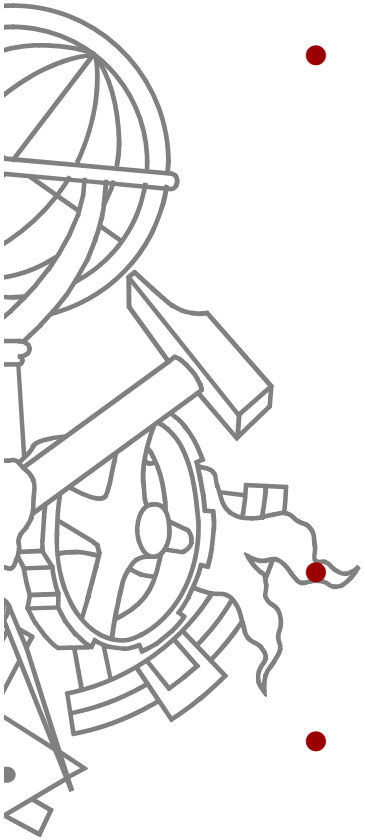
Memento

- Without violating encapsulation, capture and externalize an objects internal state so that the object can be restored to this state later



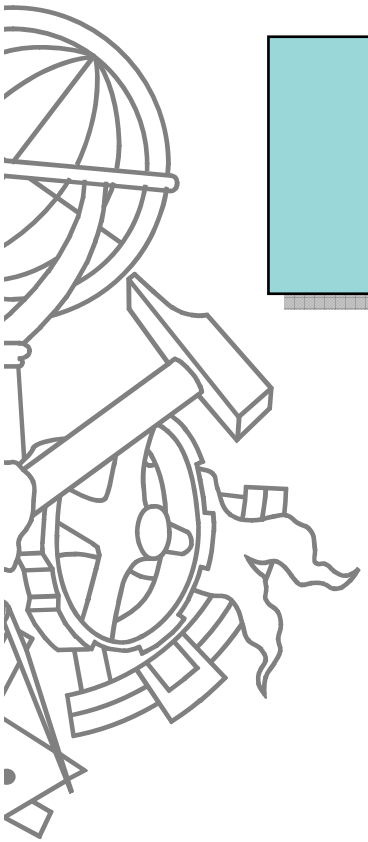
fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Participantes



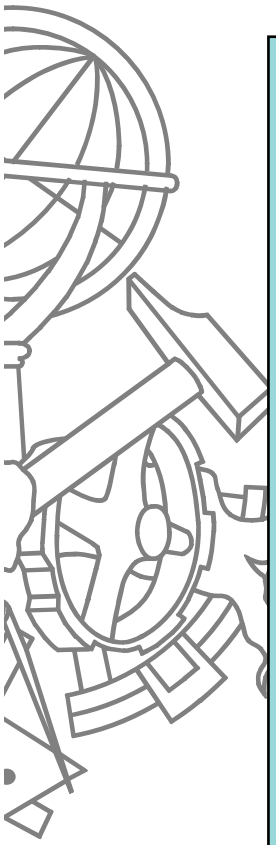
- **`Memento`**
  - stores internal state of the **`Originator`** object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - protect against access by objects of other than the originator. Mementos have effectively two interfaces. **`Caretaker`** sees a narrow interface to the Memento -- it can only pass the memento to the other objects. **`Originator`**, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produces the memento would be permitted to access the memento's internal state.
- **`Originator`**
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state
- **`Caretaker`**
  - is responsible for the memento's safekeeping
  - never operates on or examines the contents of a memento.

Exemplo



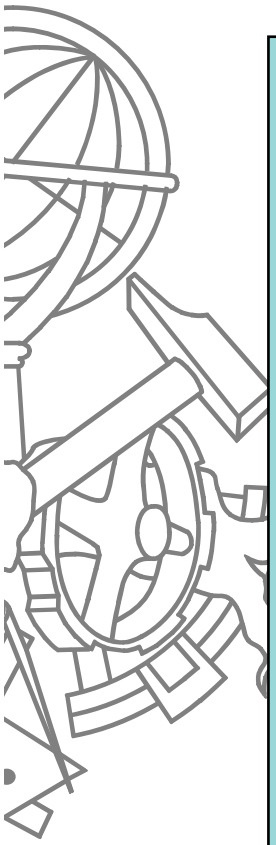
```
public interface Memento
{
}
```

Exemplo (.Net)



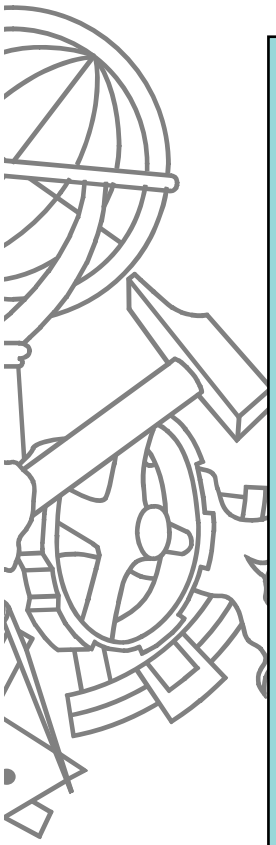
```
public class Subject
{
     private string nome;
     private int idade;

     private class SubjectMemento : Memento
     {
            public string nome;
            public int idade;

            public SubjectMemento(string n, int i)
            {
                   nome = n;
                   idade = i;
            }
     }

     // continua...
```

Exemplo (.Net)

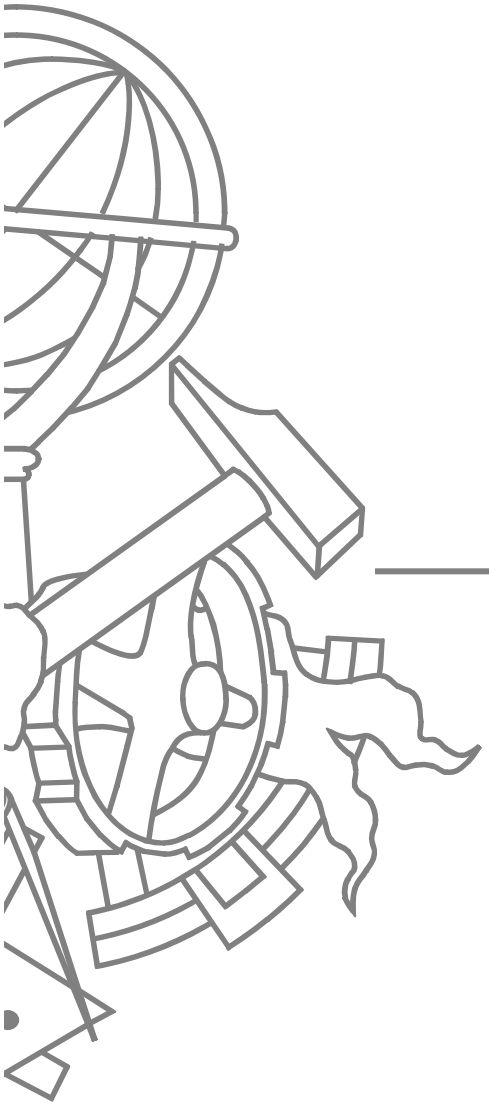


```
        // continuação...

        public Memento CreateMemento()
        {
                return new SubjectMemento(nome, idade);
        }

        public void SetMemento(Memento m)
        {
                SubjectMemento myMemento = (SubjectMemento)m;

                nome = myMemento.nome;
                idade = myMemento.idade;
        }
    }
```

Exemplo (.Net)

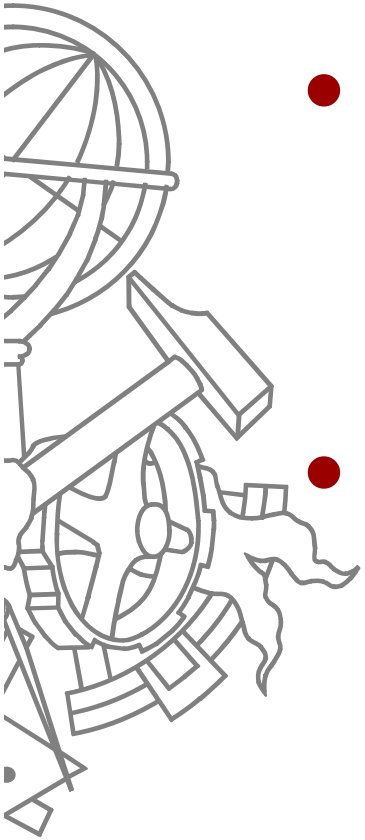


```
public class CareTaker {
    public CareTaker(Subject x) {
      theSubject = x;
    }
    IList mementos = new ArrayList();
    Subject theSubject;

    public void SetRestorePoint() {
      mementos.Insert(0, theSubject.CreateMemento());
    }

    public void Roolback() {
      Memento m = (Memento)mementos[0];
      mementos.RemoveAt(0);

      theSubject.SetMemento(m);
    }
}
```



Façade

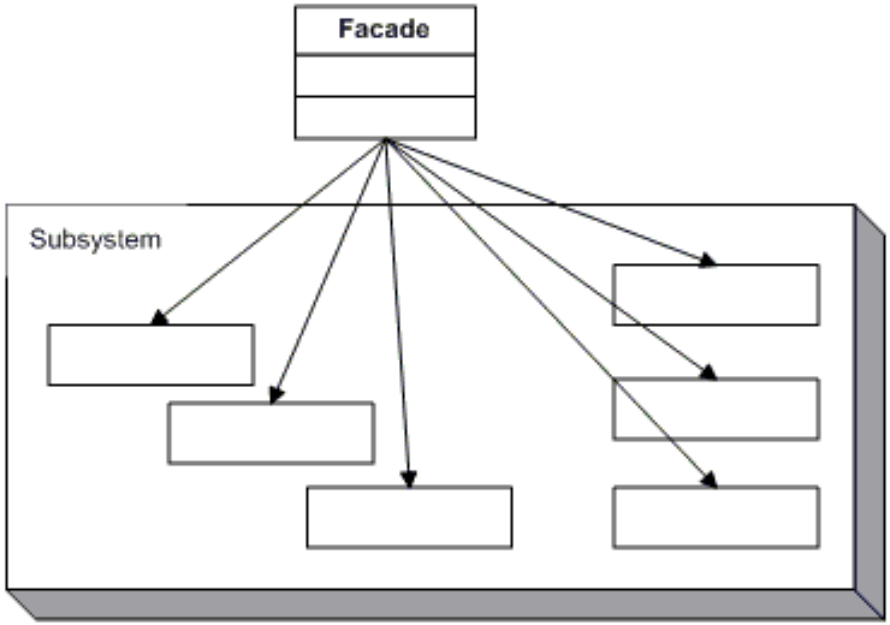
Façade



- **Problem**:
  - Actual work is performed by two or more objects, but you want to hide this level of complexity from the client.
- **Solution:**
  - Create a facade object that receives the messages, but passes commands on to the workers for completion.

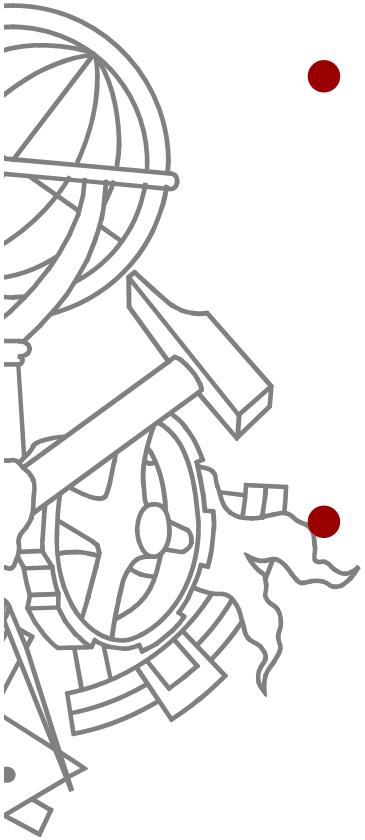
Façade

- Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

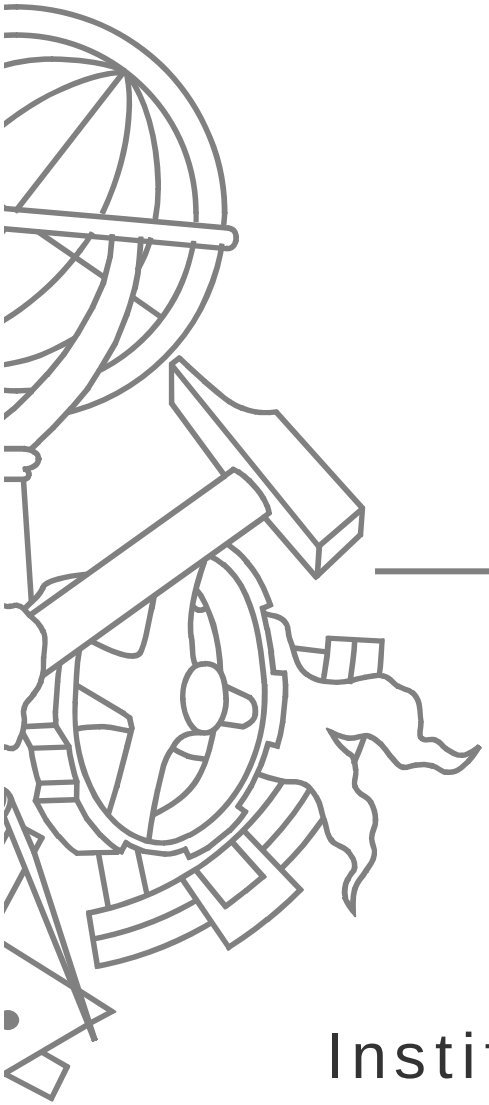


fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Participantes



- **`Facade`**
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- **Subsystem classes**
  - implement subsystem functionality.
  - handle work assigned by the **`Facade`** object.
  - have no knowledge of the facade and keep no reference to it.



Padrões GoF

Paulo Sousa

Engenharia da Informação

Instituto Superior de Engenharia do Porto