




Universidade do Porto  
Faculdade de Engenharia  
**FEUP**


# Arquitectura de Sistemas de Software

Ademar Aguiar  
[www.fe.up.pt/~aaguiar](http://www.fe.up.pt/~aaguiar)  
[ademar.aguiar@fe.up.pt](mailto:ademar.aguiar@fe.up.pt)

 **FEUP** Universidade do Porto  
Faculdade de Engenharia

Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004 1

**Abstract Factory, Builder,  
Singleton, Factory Method,  
Prototype, Adapter, Bridge,  
Composite, Decorator, Facade,  
Flyweight, Proxy, Chain of  
Responsability, Command,  
Interpreter, Iterator, Mediator,  
Memento, Observer, State,  
Strategy, Template Method,  
Visitor**

 **FEUP** Universidade do Porto  
Faculdade de Engenharia

Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004 2

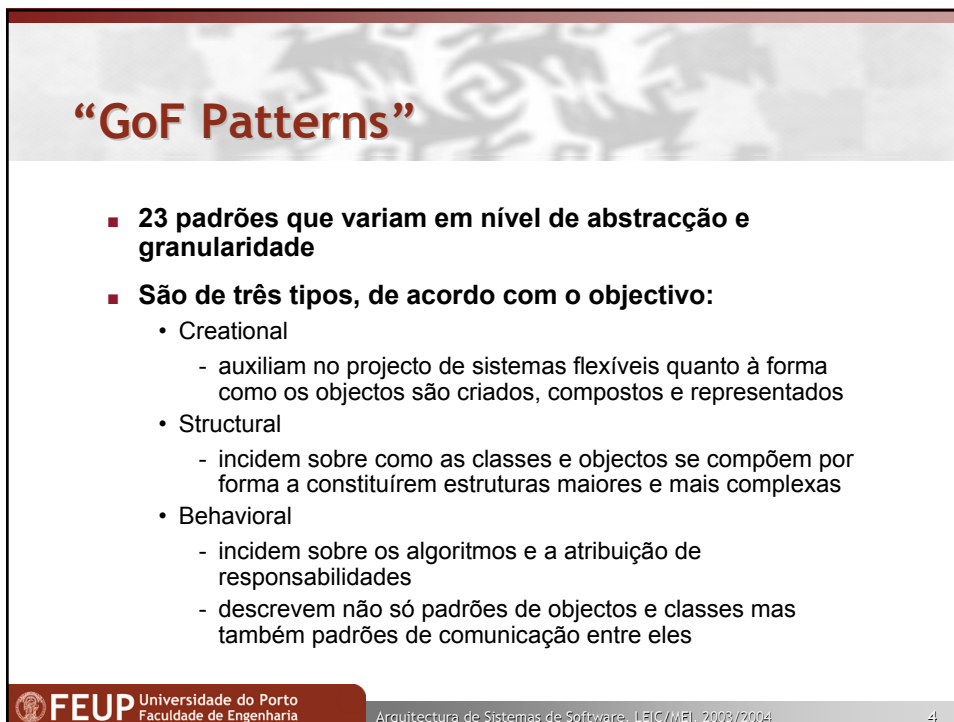
A presentation slide with a light gray background featuring a faint, repeating pattern of white birds. The title "GoF Design Patterns" is written in a bold, dark red font. Below it, in a smaller, dark red font, is the text "(GoF = Gang-of-Four)". At the bottom left is the FEUP logo and text "FEUP Universidade do Porto Faculdade de Engenharia". At the bottom right, in a small gray box, is the text "Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004" and the number "3".

## GoF Design Patterns

(GoF = Gang-of-Four)

FEUP Universidade do Porto  
Faculdade de Engenharia

Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004 3

A presentation slide with a light gray background featuring a faint, repeating pattern of white birds. The title "GoF Patterns" is written in a bold, dark red font. Below the title is a bulleted list in black text. At the bottom left is the FEUP logo and text "FEUP Universidade do Porto Faculdade de Engenharia". At the bottom right, in a small gray box, is the text "Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004" and the number "4".

## “GoF Patterns”

- 23 padrões que variam em nível de abstracção e granularidade
- São de três tipos, de acordo com o objectivo:
  - Creational
    - auxiliam no projecto de sistemas flexíveis quanto à forma como os objectos são criados, compostos e representados
  - Structural
    - incidem sobre como as classes e objectos se compõem por forma a constituírem estruturas maiores e mais complexas
  - Behavioral
    - incidem sobre os algoritmos e a atribuição de responsabilidades
    - descrevem não só padrões de objectos e classes mas também padrões de comunicação entre eles

FEUP Universidade do Porto  
Faculdade de Engenharia

Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004 4

## Nomes dos “GoF Patterns”

### Creational

Abstract Factory  
Builder  
Factory Method  
Prototype  
Singleton

### Structural

Adapter  
Bridge  
Composite  
Decorator  
Facade  
Flyweight  
•**Proxy**

### Behavioral

Chain of Responsibility  
Command  
Interpreter  
•**Iterator**  
Mediator  
Memento  
•**Observer**  
State  
Strategy  
Template Method  
Visitor



## Creational Patterns

Design Pattern	Aspecto(s) que pode(m) variar
<u>Abstract Factory (87)</u>	famílias de objectos produto
<u>Builder (97)</u>	forma de criação de objectos compostos
<u>Factory Method (107)</u>	subclasse do objecto que é criado
<u>Prototype (117)</u>	classe do objecto que é instanciado
<u>Singleton (127)</u>	única instância de uma classe



## Structural Patterns

Design Pattern	Aspecto(s) que pode(m) variar
<u>Adapter (139)</u>	interface para um objecto
<u>Bridge (151)</u>	Implementação de um objecto
<u>Composite (163)</u>	estrutura e composição de um objecto
<u>Decorator (175)</u>	responsabilidades de um objecto sem <i>subclassing</i>
<u>Facade (185)</u>	interface para um subsistema
<u>Flyweight (195)</u>	custo de armazenamento de objecto
<u>Proxy (207)</u>	forma de acesso a um objecto; a sua localização

## Behavioral Patterns

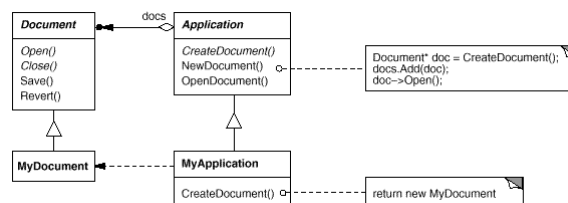
Design Pattern	Aspecto(s) que pode(m) variar
<u>Chain of Responsibility (223)</u>	objecto que pode responder a um pedido
<u>Command (233)</u>	quando e como um pedido pode ser respondido
<u>Interpreter (243)</u>	gramática e interpretação de uma linguagem
<u>Iterator (257)</u>	forma de aceder e percorrer os elementos de um objecto agregado (multi-valor)
<u>Mediator (273)</u>	como e que objectos interactuam entre si
<u>Memento (283)</u>	que informação privada é armazenada fora de um objecto, e quando
<u>Observer (293)</u>	número de objectos que dependem de outro objecto; como os objectos dependentes se mantêm actualizados
<u>State (305)</u>	estados de um objecto
<u>Strategy (315)</u>	um algoritmo
<u>Template Method (325)</u>	passos de um algoritmo
<u>Visitor (331)</u>	operações que podem ser aplicadas a objecto(s) sem alterar a(s) sua(s) classe(s)

## Uma ordem de aprendizagem

- Factory Method
- Strategy
- Decorator
- Composite
- Iterator
- Template Method
- Abstract Factory
- Builder
- Singleton
- Proxy
- Adapter
- Bridge
- Mediator
- Observer
- Chain of Responsibility
- Memento
- Command
- Prototype
- State
- Visitor
- Flyweight
- Interpreter
- Façade

## Factory Method (107)

- **Problema**
  - Definir uma interface para criar um objecto, mas delegar para subclasses a escolha da classe concreta a instanciar.
- **Motivação**



## Factory Method (107) ...

### ■ Aplicações

- Evitar ter que antecipar qual classe de objectos uma determinada classe deve criar.
- Delegar para subclasses a decisão de quais os objectos concretos a criar.
- Uma classe delega responsabilidade para várias subclasses utilitárias e pretende-se localizar o conhecimento sobre qual a subclasse é delegada.

### ■ Exemplo

- `javax.swing.BorderFactory` para criação de diversos tipos de "border's".
- `javax.xml.transform.TransformerFactory` para criação de objectos `Transformer` e `Templates`.



FEUP Universidade do Porto  
Faculdade de Engenharia

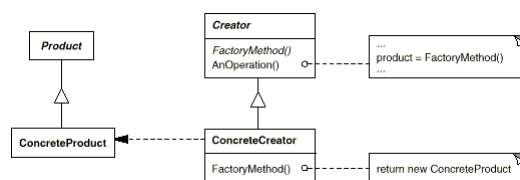
Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004

11

## Factory Method (107) ...

### ■ Solução

- Encapsular o conhecimento sobre qual o **produto concreto** (ConcreteProduct) criar numa subclasse concreta (ConcreteCreator).



[Gamma95]



FEUP Universidade do Porto  
Faculdade de Engenharia

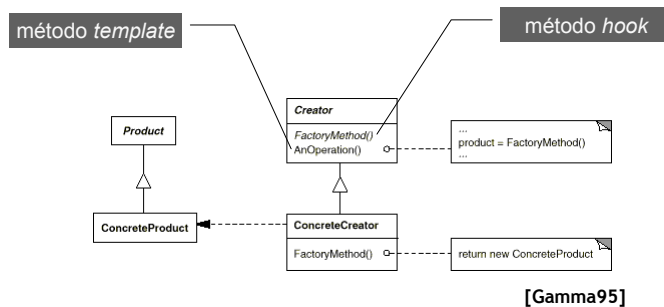
Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004

12

## Factory Method (107) ...

### ■ Solução

- Encapsular o conhecimento sobre qual o **produto concreto** (ConcreteProduct) criar numa subclasse concreta (ConcreteCreator).



## Métodos *template* e *hook*

### ■ Métodos *template*

- definem as partes “fixas” de uma funcionalidade
- implementam conhecimento de domínio genérico
- invocam métodos *hook*

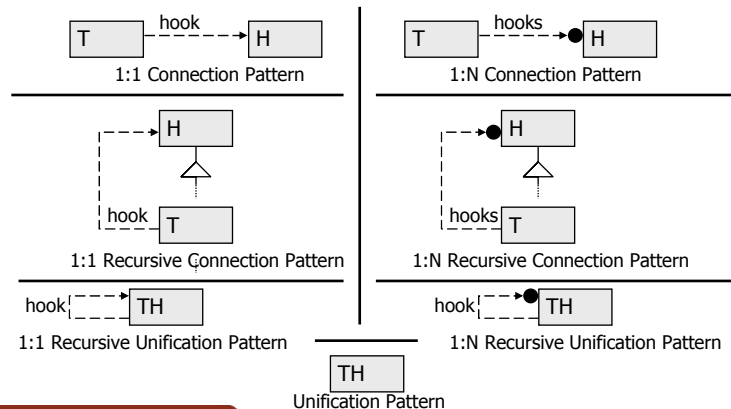
### ■ Métodos *hook*

- definem a parte flexível; a ser (re-)definida
- implementam partes específicas da aplicação
- podem ser métodos abstractos

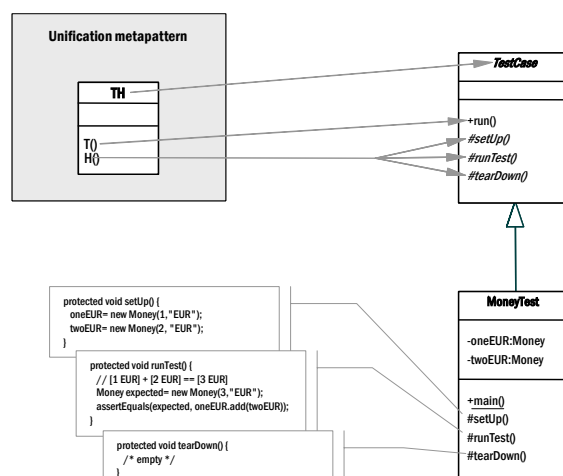
- Métodos *hook* (não-abstract) podem ser outra vez métodos *template* de outros métodos...

## Meta-patterns

- Os meta-patterns descrevem estruturas genéricas e técnicas que repetidamente se encontram em design patterns.

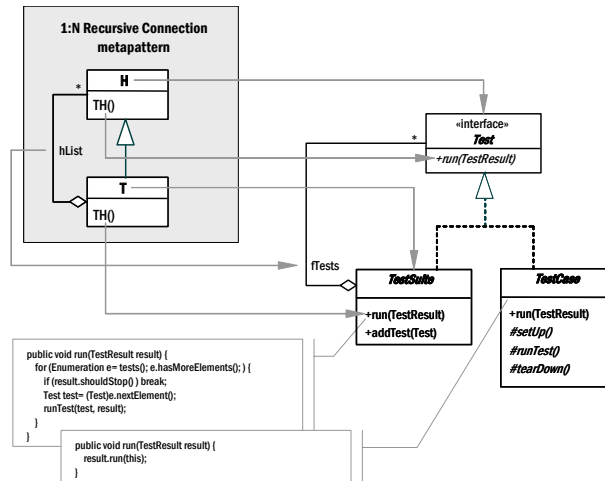


## Unification meta-pattern





## 1:N recursive connection meta-pattern

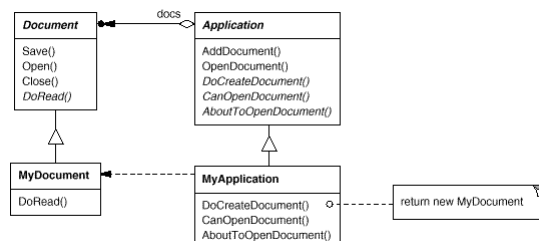


## Template Method (325)

### ■ Problema

- Alterar alguns passos de um algoritmo de uma operação sem alterar a estrutura global do algoritmo.

### ■ Motivação



## Template Method (325) ...

### ■ Aplicações

- Permitir a adaptação de alguns passos de um algoritmo, mantendo a integridade do algoritmo e a sua estrutura geral.

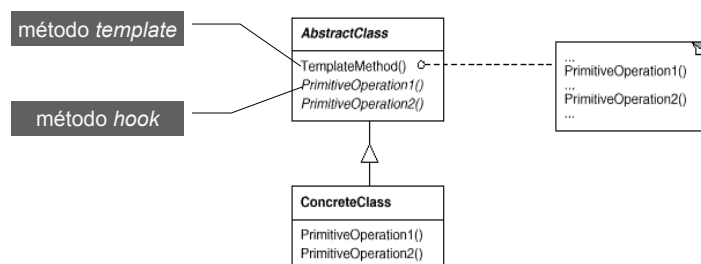
### ■ Exemplo

- Delegar a responsabilidade de implementação de alguns métodos a subclasses, por exemplo numa aplicação de desenho.



## Template Method (325) ...

### ■ Solução



## Exercícios

- **Estudar e identificar os métodos *template* e *hook* dos padrões seguintes:**
  - Strategy
  - Decorator
  - Composite
  - Iterator



(a continuar...)



## Proxy (207)

### ■ Problema

- Controlar o acesso a um objecto para, por exemplo, diferir o custo total da sua criação e inicialização até ao momento em que tal é mesmo necessário.

### ■ Aplicações

- Virtual Proxy: para a criação de objectos de recursos dispendiosos.
- Remote Proxy: para a criação de representantes locais de objectos existentes noutros espaços de endereçamento.
- Protection Proxy: para controlar o acesso a objectos partilhados.

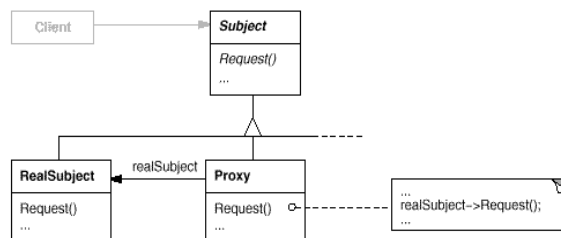
### ■ Exemplo

- Imagens num documento de um processador de texto.

## Proxy (207) ...

### ■ Solução

- Fornecer um objecto representante que permite o acesso transparente a um outro objecto



[Gamma95]

## Iterator (257)

### ■ Problema

- Aceder sequencialmente aos elementos de um objecto composto sem necessidade de expor a sua representação interna

### ■ Aplicações

- aceder sequencialmente aos elementos de um objecto composto
- permitir múltiplos tipos de visitas a objectos compostos
- fornecer uma interface comum para visitar diferentes estruturas compostas

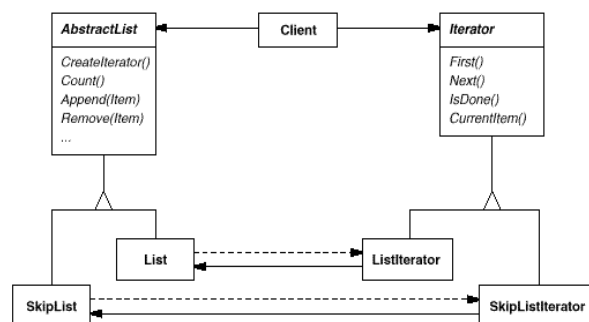
### ■ Exemplo

- Visita uniforme aos elementos de uma colecção de objectos, independentemente do seu tipo concreto: lista, árvore, vector, pilha, etc.



## Iterator (257) ...

### ■ Solução



[Gamma95]



## Comentários

- Os “GoF Design Patterns” auxiliam no projecto de sistemas de software *reutilizáveis* e facilmente *extensíveis*, embora à custa de um aumento da sofisticação dos modelos de classes e das suas interações.
- Estes padrões aplicam-se a inúmeros domínios de problemas: editores de desenho, banca, telecomunicações, CAD, CASE's, aplicações médicas, etc.
- Muitos destes padrões são considerados padrões atómicos, isto é, são padrões que não encerram em si outros padrões.
- Estes padrões, tal como quaisquer outros, não foram “inventados” a partir do nada, mas sim “descobertos” através do estudo de produtos concretos.

## Caso de Estudo: desenho da framework JUnit

## Desenho de software com padrões

### ■ Desenho com Padrões (*top-down*)

- Identificar o problema a resolver
- Seleccionar os padrões adequados
- Verificar a aplicabilidade dos padrões ao problema em questão
- Instanciar o padrão na situação concreta
- Avaliar os compromissos de desenho envolvidos

### ■ Refactoring para Padrões (*bottom-up*)

- Evoluir o desenho de código existente através da aplicação de padrões.
- *Refactoring* é uma técnica de que se popularizou bastante com o método “Extreme Programming” e consiste em aplicar sucessivas transformações de código que preservam a sua semântica.
- Exemplos: *Rename Method*, *Extract Method*, *Form Template Method*



Universidade do Porto  
Faculdade de Engenharia

Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004

29

## JUnit - framework para testes

### ■ Pressupostos

- Se um programa não possui testes automatizados, assume-se que o programa não funciona!
- Normalmente assume-se que se um programa funciona agora, funcionará sempre...será? Não!
- Assim, para além do código, os programadores têm também de escrever testes garantindo que o seu programa funciona.



Universidade do Porto  
Faculdade de Engenharia

Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004

30

## Objectivos da JUnit

- Facilitar a escrita de testes, através do uso de ferramentas fáceis de aprender e que eliminam o duplo esforço de escrita de código e testes.
- Criar testes que preservam o seu valor ao longo do tempo e que podem ser combinados com testes de outros autores sem receio de interferir com outros.
- Ser possível criar novos testes com base em existentes.



## Evolução do Desenho da JUnit

- **Processo**
  - Começar do zero.
  - Identificar o problema de desenho a ser resolvido.
  - Seleccionar o padrão que resolve o problema.
  - Instanciar o padrão no contexto do problema.
  - Repetir o processo até ter todos os problemas resolvidos.
- **Cinco problemas de desenho... Cinco design patterns**
  - Problema 1. Como representar um teste?
  - Problema 2. Onde colocar o código de teste?
  - Problema 3. Como registar os resultados de um teste?
  - Problema 4. Como uniformizar os testes?
  - Problema 5. Como agrupar vários testes?





## 1. Como representar um teste?

### ■ Forças

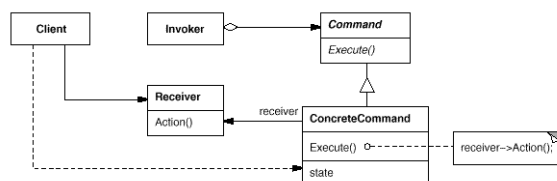
- Os testes devem ser concretizados em objectos.
- Cada teste deve poder ser facilmente manipulado.
- Os testes devem preservar o seu valor ao longo do tempo.

### ■ Padrão seleccionado: **Command** [Gamma95]

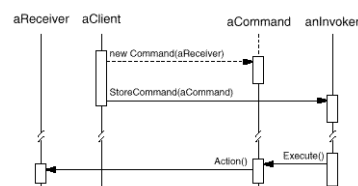
- “Encapsulate a request as an object, thereby letting you... queue or log requests...”
- O padrão sugere criar um objecto para cada operação e atribuir-lhe um método "execute".

## Command

### ■ Estrutura



### ■ Colaborações



## JUnit - 1



```

public abstract class TestCase implements Test {
    private final String fName;
    public TestCase(String name) {
        fName = name;
    }
    public abstract void run();
    ...
}
  
```

## 2. Onde colocar o código de teste?

### ■ Forças

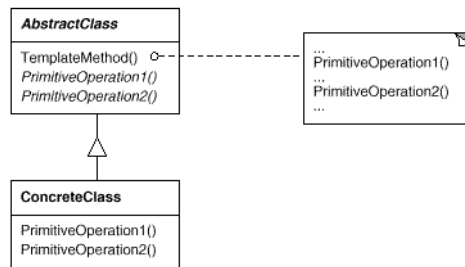
- Fornecer ao programador um local conveniente para colocar o código de teste.
- Fornecer a estrutura comum a todos os testes:
  - preparar teste
  - executar teste
  - avaliar resultados
  - e limpar teste.
- Os testes devem ser executados de forma independente.

### ■ Padrão seleccionado: **Template Method [Gamma95]**

- *"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure"*
- O padrão garante a estrutura global do algoritmo, permitindo ainda a redefinição de alguns dos seus passos.

## Template Method

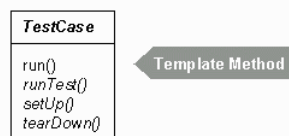
### ■ Estrutura



### ■ Colaborações

- *ConcreteClass* delega em *AbstractClass* a implementação dos passos invariantes do algoritmo.

## JUnit - 2



```

public void run() {
    setUp();
    runTest();
    tearDown();
}
protected void runTest() { }
protected void setUp() { }
protected void tearDown() { }
  
```

### 3. Como registar os resultados?

#### ■ Forças

- Pretende-se apenas registar os resultados dos testes que falham, de forma condensada.
- Pretende-se distinguir *falhas* de *erros*:
  - As falhas são situações antecipadas verificáveis por regras.
  - Os erros correspondem a problemas não previstos.

#### ■ Padrão seleccionado: **Collecting Parameter** [Kent96]

- “... when you need to collect results over several methods, you should add a parameter to the method and pass an object that will collect the results for you...”



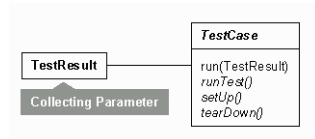
### Collecting Parameter

#### ■ Estrutura

```
void f(ParameterType param) {  
    // accumulate values on param  
    ...  
}
```



## JUnit - 3a



```

public void run(TestResult result) {
    result.startTest(this);
    setUp();
    runTest();
    tearDown();
}

public TestResult run() {
    TestResult result= new TestResult();
    run(result);
    return result;
}
  
```

## Exception Handling

### ■ Mecanismo de tratamento de excepções

- Exception
- Exception handler
- throw
- try
- catch

## JUnit - 3b

```

public void run(TestResult result) {
    result.startTest(this);
    setUp();
    try {
        runTest();
    }
    catch (AssertionFailedError e) {
        result.addFailure(this, e);
    }
    catch (Throwable e) {
        result.addError(this, e);
    }
    finally {
        tearDown();
    }
}

```



## JUnit - 3b

```

//TestCase
protected void assert(boolean condition) {
    if (!condition)
        throw new AssertionError();
}

//TestResult
public synchronized void addError(Test test, Throwable t) {
    fErrors.addElement(new TestFailure(test, t));
}

public synchronized void addFailure(Test test, Throwable t) {
    fFailures.addElement(new TestFailure(test, t));
}

//Example
public void testMoneyEquals() {
    assert(!f12CHF.equals(null));
    assertEquals(f12CHF, f12CHF);
    assertEquals(f12CHF, new Money(12, "CHF"));
    assert(!f12CHF.equals(f14CHF));
}

```



## 4. Como uniformizar os testes?

### ■ Forças

- Precisa-se de uma interface genérica para executar os testes. (De momento, os testes são implementados como diferentes métodos da mesma classe, e por isso não satisfazem uma interface única.)
- Os testes têm de parecer idênticos do ponto de vista de quem os invoca.

### ■ 1º Padrão seleccionado: *Adapter* [Gamma95]

- “Convert the interface of a class into another interface clients expect.”

### ■ 2º Padrão seleccionado: *Pluggable Selector* [Beck96]

- “use a single class which can be parameterized to perform different logic without requiring subclassing, e.g. a method selector”



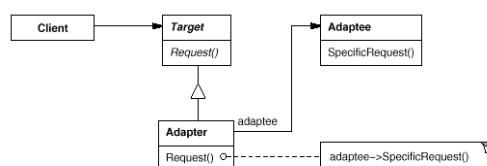
Universidade do Porto  
Faculdade de Engenharia

Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004

45

## Adapter

### ■ Estrutura



### ■ Colaborações

- Os Client invocam operações numa instância de *Adapter*.
- Por sua vez, o *adapter* invoca as operações do *Adaptee* que executam o pedido.



Universidade do Porto  
Faculdade de Engenharia

Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004

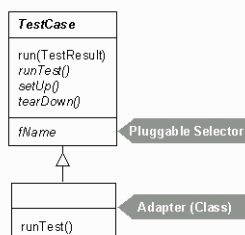
46

## Java Reflection API

### ■ Mecanismo de introspecção do Java

- Class
- Method `getMethod(String methodName, Object[] argTypes)`
- void `invoke(Object obj, Object[] args)`

## JUnit - 4



```

protected void runTest() throws Throwable {
    Method runMethod= null;
    try {
        runMethod= getClass().getMethod(fName, new Class[0]);
    } catch (NoSuchMethodException e) {
        assert("Method \"\"+fName+\"\" not found", false);
    }
    try {
        runMethod.invoke(this, new Class[0]);
    } // catch Exceptions
}
  
```



## 5. Como agrupar vários testes?

### ■ Forças

- Para obter a máxima confiança sobre o nosso sistema, pretendemos executar muitos testes, e não apenas um como até agora acontece.
- Quem invoca os testes não se deverá preocupar se está a invocar um único teste, um conjunto de testes ou mesmo um conjunto de conjuntos de testes.

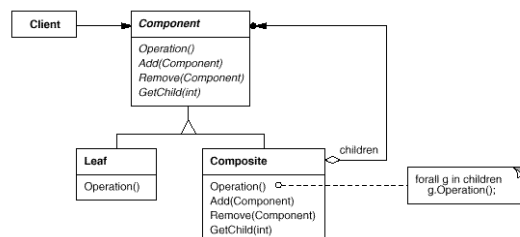
### ■ Padrão seleccionado: **Composite** [Gamma95]

- *"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly."*



## Composite

### ■ Estrutura

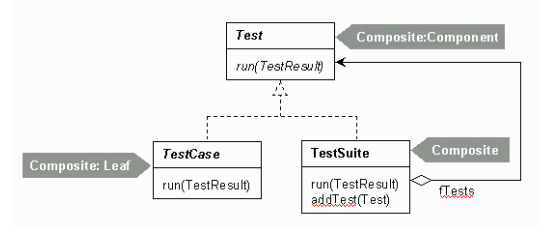


### ■ Colaborações

- Os clientes usam a interface de *Component* para interagir com os objectos na árvore de componentes. Se o componente é uma folha da árvore, o pedido é efectuado directamente, senão o pedido é redireccionado para os seus constituintes.



## JUnit - 5

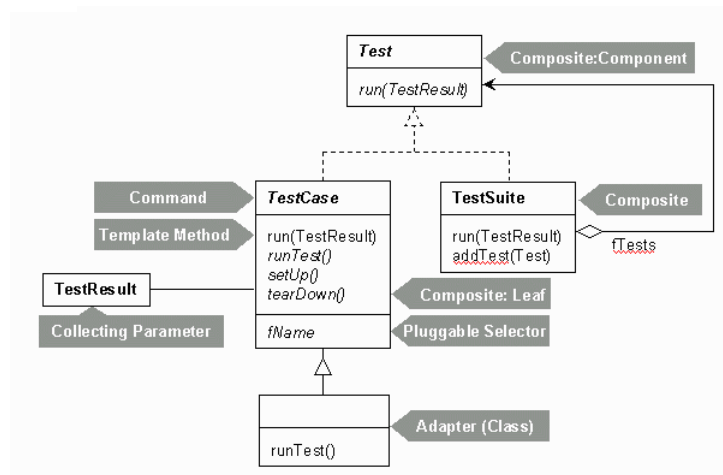


```

public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new MoneyTest("testMoneyEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
}

```

## Resultado final - JUnit



## Comentários Gerais

### ■ Padrões

- Os padrões são bastante adequados para apresentar o desenho de sistemas de software, bem como as opções envolvidas.

### ■ Densidade de Padrões

- A classe *TestCase* possui uma elevada densidade de padrões aplicados (4), uma particularidade típica de *frameworks* com um bom nível de maturidade.
- Desenhos com elevada densidade de padrões são normalmente mais fáceis de utilizar mas mais difíceis de modificar.

### ■ “Do The Simplest Thing That Could Possibly Work”

- Mais padrões poderiam ainda ser aplicados, mas isso provavelmente complicaria a *framework* sem trazer grandes vantagens adicionais aos seus utilizadores.

## Considerações Finais

- A importância dos padrões para a melhoria da produtividade e qualidade do desenvolvimento de software é hoje reconhecido por toda a comunidade de software.
- Os padrões são hoje largamente populares.
- É crucial para todos os que desenvolvem software conhecerem a existência dos padrões e como, e quanto, nos podem ajudar a melhorar os resultados do nosso trabalho.
- Para tirar o máximo benefício dos padrões **é necessário saber utilizá-los da forma mais eficaz.**
- **A instanciação de padrões requer sentido crítico** por requerer sempre adaptação ao problema concreto em mãos.
- Os padrões (de desenho) podem ser utilizados em actividades de **refinamento** (*top-down*) ou de **abstracção** (*bottom-up*).
- A melhor forma de descobrir os seus benefícios é ousá-los **aplicar aos nossos problemas concretos, se necessários...**

## Bibliografia e Referências

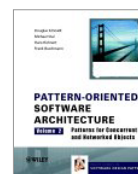
## Livros



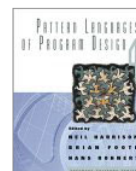
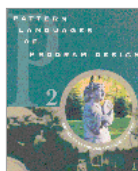
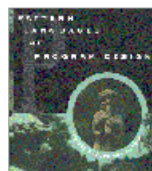
**“GoF book”**  
[Gamma95]



**“POSA”**  
[Buschmann96]



**“POSA2”**  
[Buschmann00]



**“PLOPD 1,2,3,4”**

## Web

- **Patterns Home Page**
  - <http://hillside.net/>
- **Cetus Links**
  - [http://www.cetus-links.org/oo\\_patterns.html](http://www.cetus-links.org/oo_patterns.html)
- **Outros**
  - <http://www.fe.up.pt/~aaguilar/patterns/>



Universidade do Porto  
Faculdade de Engenharia

Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004

57

## Bibliografia

- [Alexander77] C. Alexander and S. Ishikawa and M. Silverstein, A Pattern Language, Oxford University Press, 1977.
- [Alexander79] C. Alexander, A Timeless Way of Building, Oxford University Press, 1979.
- [Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- [Buschmann96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern Oriented Software Architecture - a System of Patterns, John Wiley and Sons, 1996.
- [Buschmann99] F. Buschmann, Building Software with Patterns, EuroPLoP'99 Proceedings.
- [Cope95] J. O. Coplien and D. C. Schmidt, Pattern Languages of Program Design, Addison-Wesley, 1995.
- [Vlissides96] J. M. Vlissides, J. O. Coplien, and N. L. Kerth, Pattern Languages of Program Design 2, Addison-Wesley, 1996.
- [Martin97] R. C. Martin, D. Riehle, and F. Buschmann, Pattern Languages of Program Design 3, Addison-Wesley, 1997.
- [Harrison99] N. Harrison, B. Foote, H. Rohnert, Pattern Languages of Program Design 4, Addison-Wesley, 2000.
- [Beck96] K. Beck, Smalltalk Best Practice Patterns, Prentice Hall, 1996.
- [Beck&Gamma], "JUnit Cook's Tour", em <http://www.junit.org>
- [Aguiar00], "Software Patterns: uma forma de reutilizar conhecimento", em [www.fe.up.pt/~aaguilar/patterns/](http://www.fe.up.pt/~aaguilar/patterns/), FEUP, 2000.



Universidade do Porto  
Faculdade de Engenharia

Arquitectura de Sistemas de Software, LEIC/MEI, 2003/2004

58

