Desenho com Padrões:

Exemplo de desenho da framework de testes JUnit

Ademar Aguiar
www.fe.up.pt/~aaguiar
ademar.aguiar@fe.up.pt

Universidade do Porto
Faculdade de Engenharia
FEUP

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 prática que se popularizou bastante com o método "Extreme Programming" e consiste em transformar código, preservando a sua semântica.
- **Caso de Estudo**
  - Desenho da framework de testes JUnit [Beck&Gamma]

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.

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

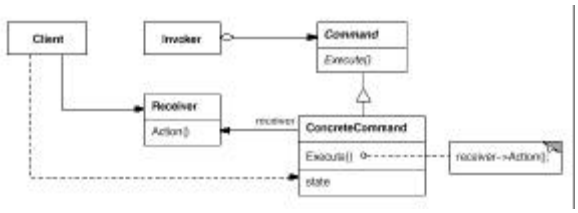
- **Processo**
  - Começar do zero.
  - Identificar o problema a ser resolvido.
  - Apresentar o padrão que resolve o problema.
  - Aplicar o padrão.
  - Repetir o processo até resolver todos os problemas.
- **5 problemas**
  - 1. Como representar um teste?
  - 2. Onde colocar o código de teste?
  - 3. Como registar os resultados?
  - 4. Como uniformizar os testes?
  - 5. Como agrupar vários testes?

1. Como representar um teste?

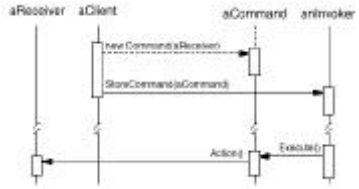
- **Forças**
  - As ideias de testes devem ser concretizadas em objectos.
  - Os testes devem ser manipulados facilmente.
  - Os testes devem preservar o seu valor ao longo do tempo.
- **Padrão seleccionado: *Command* [Gamma95]**
  - "Encapsulate a request as an object, thereby letting you… queue or log requests…"
  - O padrão diz para criar um objecto para uma operação e atribuir-lhe um método "execute".

Command

- **Estrutura**



- **Colaborações**



JUnit - 1

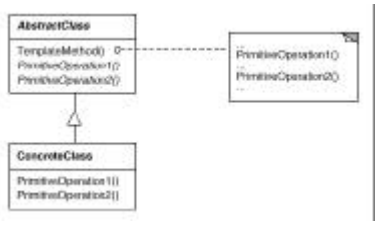

```
public abstract class TestCase implements Test {
      private final String fName;
public TestCase(String name) {
          fName= name;
      }
    public abstract void run();
          …
}
```

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.

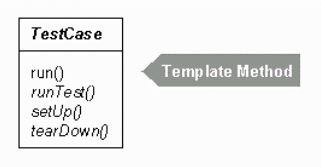
Template Method

- **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?

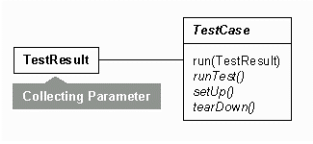
- **Forças**
  - Pretende-se apenas registar os resultados dos testes que falham, de forma condensada.
  - Pretende-se distinguir *falhas* de *erros*:
    - As falhas são situações antecipadas verificáveis por regras.
    - Os erros correspondem a problemas não previstos.
- **Padrão seleccionado: *Collecting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

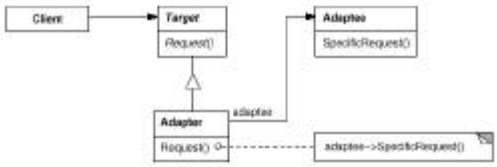
```
//TestCase
protected void assert(boolean condition) {
       if (!condition)
           throw new AssertionFailedError();
   }
//TestResult
public synchronized void addError(Test test, Throwable t) {
       fErrors.addElement(new TestFailure(test, t));
   }
public synchronized void addFailure(Test 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"

Adapter

- **Estrutura**

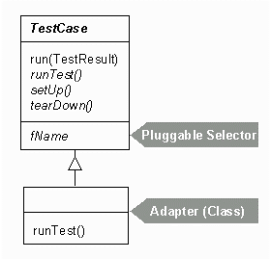


- **Colaborações**
  - Os Client invocam operações numa instância de *Adapter*.
  - Por sua vez, o *adapter* invoca as operações do *Adaptee* que executam o pedido.

Java Reflection API

- **Mecanismo de introspecção do Java**
  - `Class`
  - `Method getMethod(String methodName, Object[] argTypes)`
  - `void invoke(Object obj, Object[] args)`

JUnit - 4



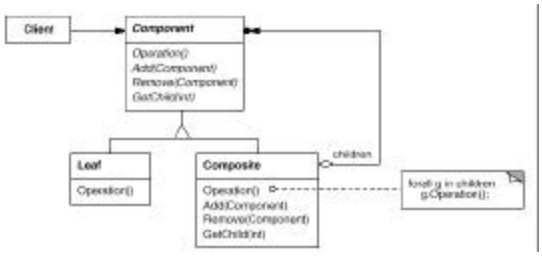

```
protected void runTest() throws Throwable {
      Method runMethod= null;
      try {
          runMethod= getClass().getMethod(fName, new Class[0]);
      } catch (NoSuchMethodException e) {
          assert("Method \""+fName+"\" not found", false);
      }
      try {
          runMethod.invoke(this,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

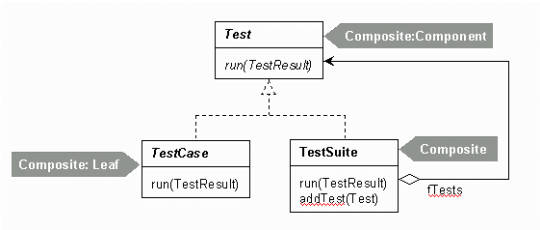
Composite

- **Estrutura**



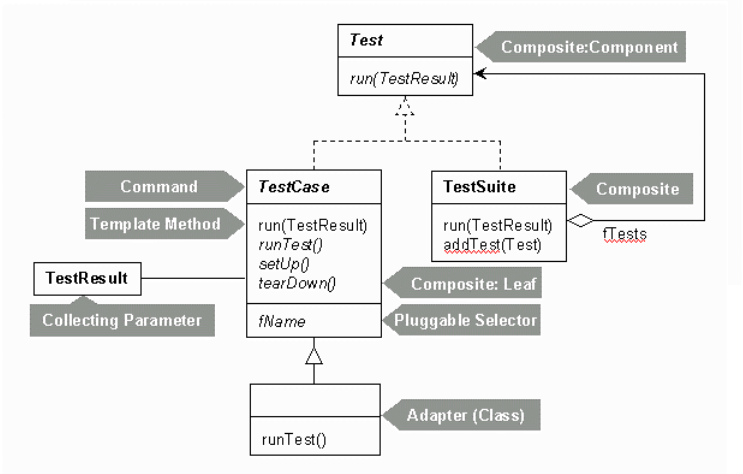
- **Colaborações**
  - Os clientes usam a interface de *Component* para interagir com os objectos na árvore de componentes. Se o componente é uma folha da árvore, o pedido é efectuado directamente, senão o pedido é redireccionado para os seus constituintes.

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), particularidade típica de *frameworks* com maturidade.
  - Desenhos com elevada densidade de padrões são normalmente mais fáceis de utilizar mas mais dificeis 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 reconhecidamente aceite.
- 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 no entanto 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 **aplicando-os!**

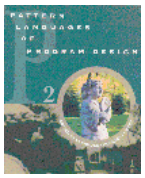
Bibliografia e Referências

Livros

"GoF book"
[Gamma95]

"POSA"
[Buschmann96]

"POSA2"
[Buschmann00]



"PLOPD 1,2,3,4

Web

- **Patterns Home Page**
  - http://hillside.net/patterns/
- **Cetus Links**
  - http://www.cetus-links.org/oo_patterns.html
- **Wiki Wiki Web**
  - http://c2.com/cgi-bin/wiki

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", www.junit.org
- [Aguiar00], "Software Patterns: uma forma de reutilizar conhecimento", em www.fe.up.pt/~aaguiar/patterns/, FEUP, 2000.