Design Patterns na plataforma Java

Uma experiência no processo de migração para .NET

João Saraiva

Instituto Superior Técnico /
INESC-ID (Grupo de Sistemas de Informação)

Sumário

- Apresentação de alguns padrões de desenho:
  - *Proxy* e *Bridge*
  - *Composite*
  - *Visitor*
  - *Memento*
  - *Command*
  - *Observer*
  - *Strategy*

Proxy e *Bridge*

Proxy e *Bridge* (1/7)

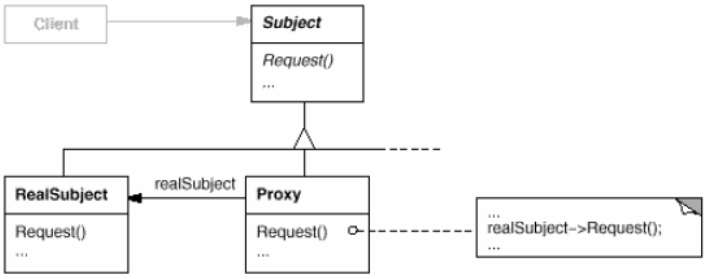
- Problema:
  - O utilizador precisa de manipular um "recurso" (*resource*) no sistema de ficheiros;
  - A informação destes "recursos" muda com o tempo
    (contudo, a sua identidade mantém-se);
  - Não queremos aceder a estado antigo destes recursos (por exemplo, quando temos um recurso e o apagamos).

Proxy e *Bridge* (2/7)

- Solução:
  - Introduzir um nível de indirecção
  - Dar ao utilizador apenas um *handle* para um recurso, em vez de lhe dar o recurso inteiro
  - Esta solução é a aplicação de uma mistura de 2 padrões de desenho: *Proxy* e *Bridge*

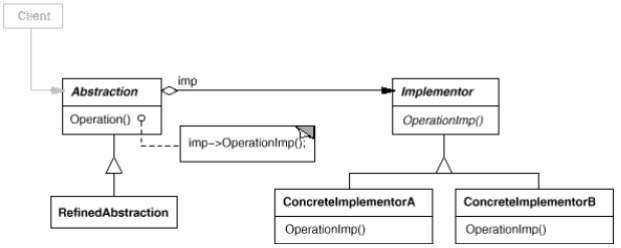
Proxy e *Bridge* (3/7)

- *Proxy*:
  - "Fornecer um objecto representante de outro, para controlar o acesso a este último"



Proxy e *Bridge* (4/7)

- *Bridge*:
  - "Separar uma abstracção da sua implementação, de modo a que os dois possam variar independentemente"



Proxy e *Bridge* (5/7)

- Relevância para este problema:
  - *Proxy:*
    - Controla o acesso ao objecto (fornece o *handle* – *IFile*, *IFolder*, *IProject*, *IWorkspaceRoot*)
    - O utilizador não guarda estado antigo do objecto
  - *Bridge*:
    - Separa a interface e a implementação
    - Fornece um objecto com informação que representa o estado do ficheiro (há apenas 1 objecto para cada *handle*)

Proxy e *Bridge* (6/7)

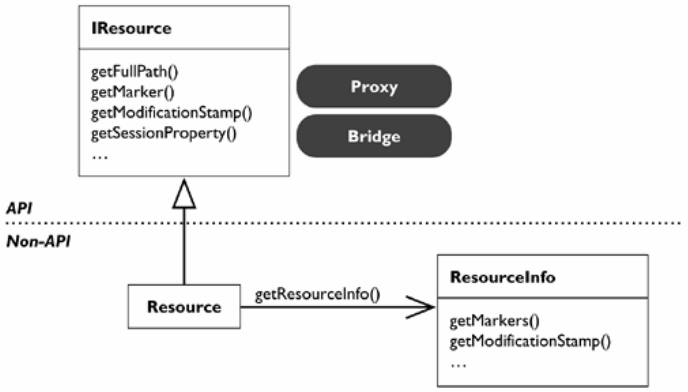
- Algumas curiosidades acerca destes *handles*:
  - São *Value Objects* – "pequenos" objectos cuja igualidade não se baseia na identidade do objecto
    - Uma vez criados, os seus valores não mudam
    - Podemos usá-los como chaves em *Maps*
  - Pode-se referir a recursos que não existam;
  - São criados a partir de um *handle*-pai:

```
IProject project;
IFolder folder = project.GetFolder("someFolder");
```

  - O utilizador não consegue guardar estado antigo do recurso; de cada vez que é necessário operar sobre este estado, tem que se o ir buscar;

Proxy e *Bridge* (7/7)

- *IResource* (e a sua implementação, *Resource*) consiste na aplicação do padrão *Proxy* e do padrão *Bridge*



Composite

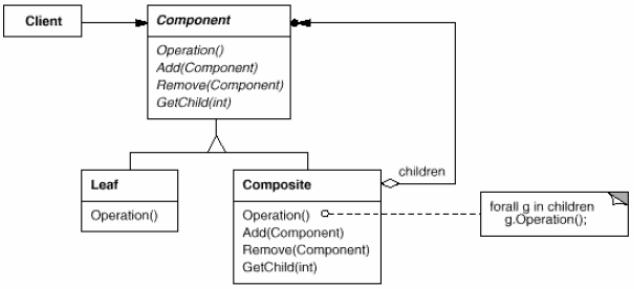
Composite (1/7)

- Problema: Como organizar os *handles* dos recursos, de modo a reflectir a organização do sistema de ficheiros?

- Solução: Usar uma árvore de recursos.

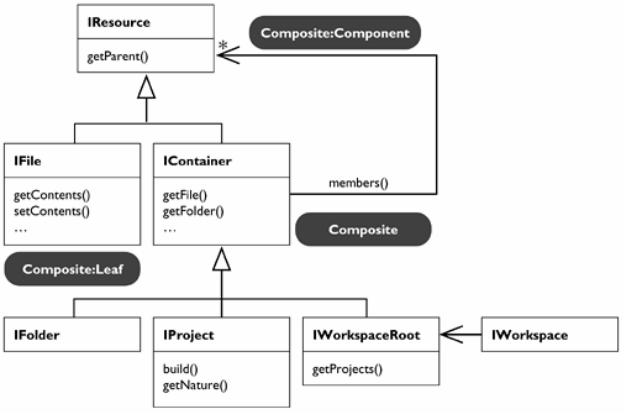
Composite (2/7)

- *Composite*:
  - "Compôr objectos em estruturas em árvore de modo a representar hierarquias totais ou parciais. Os utilizadores podem tratar objectos individuais e composições de objectos de modo igual."



Composite (3/7)

- O *Composite* permite-nos organizar os recursos de modo a reflectir o sistema de ficheiros:

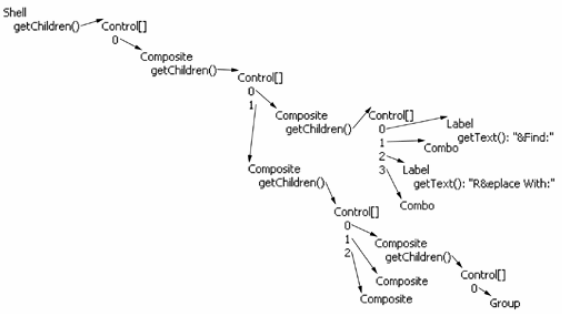


Composite (4/7)

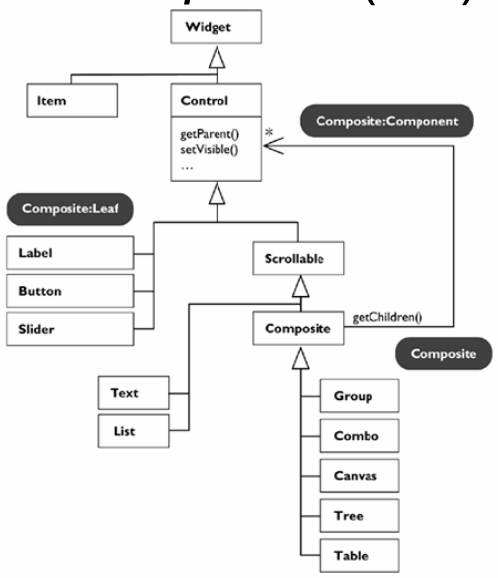
- Outro exemplo: o SWT (*Standard Widget Toolkit*)
  - Utiliza o padrão Composite
  - Elementos gráficos mais complexos são constituídos a partir de elementos gráficos simples
    - *Basic widgets*: não contêm outros *widgets*; são as folhas na árvore de *widgets*. Exemplo: botões e *labels*
    - *Compound widgets*: contêm outros *widgets* e são nós internos na árvore de *widgets*. A classe base destes *widgets* é a classe Composite

Composite (5/7)

- Exemplo: Uma janela e a sua árvore de *widgets*



Composite (6/7)



- (*Item* não faz parte da aplicação do padrão)

Composite (7/7)

- Uma curiosidade:
  - Os *widgets* no SWT não têm *addChild()* e *removeChild()*
    - O *widget*-pai é fornecido ao *widget* a criar (no construtor)
    - Isto garante que todos os *widgets* têm pai (elimina a complexidade que advém do comportamento que um *widget* deve ter quando não tem pai)

```
private Control addControl(Composite parent) {
  Composite composite= new Composite(parent, SWT.NONE);
  //...
  Label label= new Label(composite, SWT.NONE);
  //...
}
```

Visitor

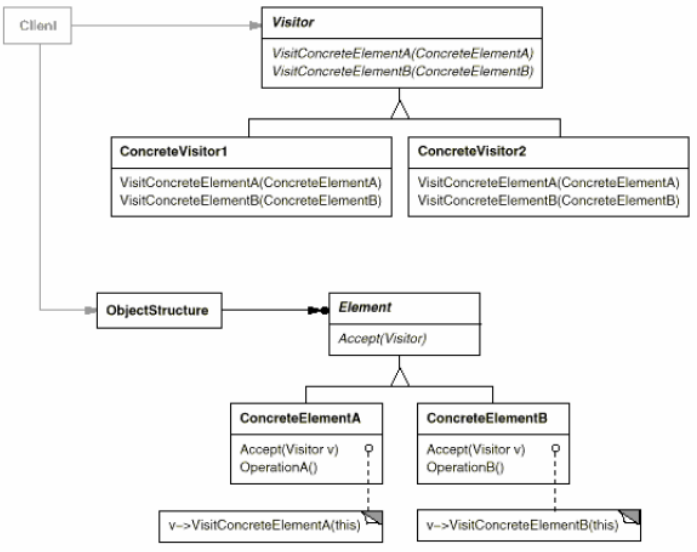
Visitor (1/4)

- Problema:
  - A cada nova operação que queiramos adicionar sobre uma estrutura de objectos, temos que adicionar (e recompilar) código a esses objectos;
  - Se percorrermos a árvore de recursos manualmente (através da chamada ao método *members*()), vamos ter muito código de fluxo de controlo no código-cliente;

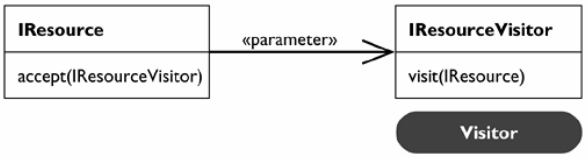
Visitor (2/4)

- Solução:
  - Usar o padrão *Visitor*; permite-nos adicionar operações sem alterar o código dos objectos
- *Visitor*:
  - "Representar uma operação a ser feita sobre os elementos de uma estrutura de objectos. Permite-nos definir uma nova operação sem alterar as classes dos elementos em que se executa."
- Neste caso, este padrão é usado para extrair o fluxo de controlo: se *visit*() devolver *true*, continua-se a operação com os filhos, caso contrário, pára-se por aí;

Visitor (3/4)



Visitor (4/4)




```
private void addToLocalHistory(IResource root, int depth) {
  IResourceVisitor visitor = new IResourceVisitor() {
    public boolean visit(IResource resource) throws CoreException {
      if (resource.getType() == IResource.FILE)
        addToLocalHistory((IFile) resource);
      return true;
    }
  };
  try {
    root.accept(visitor, depth, false);
  } catch (CoreException e) {
  }
}
```

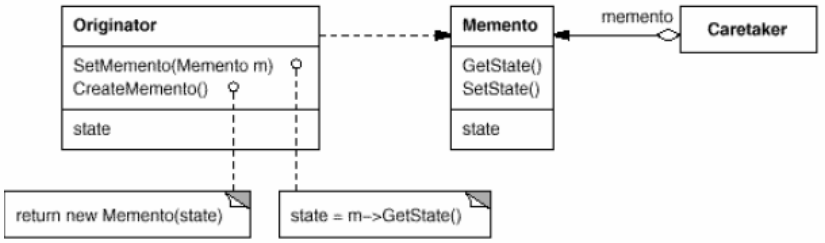

Memento

Memento (1/5)

- Problema:
  - Dar a impressão de continuidade entre sessões de utilização (ou seja, guardar o estado dos *widgets*).
    - O mecanismo de suporte a isto deve ser robusto face à mudança (mudança de versão, etc)
    - O mecanismo deve suportar estruturas complexas
    - Devemos poder guardar e carregar objectos arbitrários
    - Não podemos quebrar o encapsulamento do objecto
- Serialização de objectos pode dar problemas a longo prazo.

Memento (2/5)

- *Memento*:
  - "Sem violar o encapsulamento, capturar e exteriorizar o estado interno de um objecto, para que esse objecto possa ser reposto nesse estado mais tarde."



Memento (3/5)

- Um *IMemento* é um objecto no qual podemos guardar pares chave-valor de tipos primitivos
- Fornecemos a cada *widget* um *IMemento*, no qual o *widget* irá guardar toda a sua informação relevante
  - Cada *widget* tem um método *save()*, que recebe um *IMemento* e nele guarda o seu estado
  - Os *widgets* não têm nenhum método *restore()*; recebem o *IMemento* aquando da sua inicialização e vão buscando os valores quando são necessários
- Os dados são guardados de forma persistente em XML

Memento (4/5)

- Um exemplo (1/2):

```
public interface IMemento {
 void putString(String key, String value);
 String getString(String key);

 void putInteger(String key, int value);
 Integer getInteger(String key);

 IMemento createChild(String type);
 IMemento[] getChildren(String type);
 //...
}
```

Memento (5/5)

- Um exemplo (2/2):

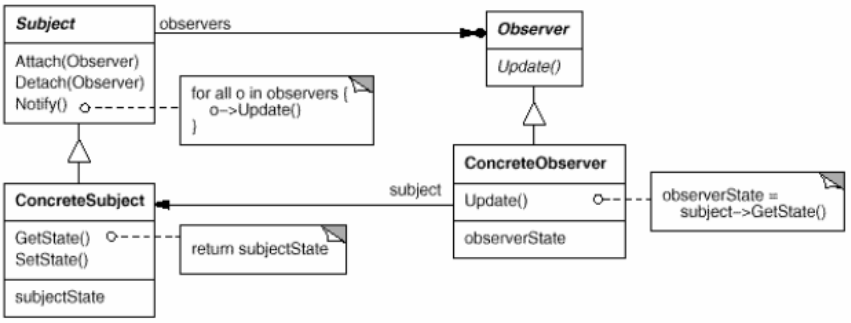
```
private static final String TAG_SELECTION = "selection";
private static final String TAG_ELEMENT = "element";
public void saveState(IMemento memento) {
  //...
  //save selection
  Object elements[] =
    ((IStructuredSelection)viewer.getSelection()).toArray();
  if (elements.length > 0) {
    IMemento selectionMem = memento.createChild(TAG_SELECTION);
    for (int i = 0; i < elements.length; i++) {
      IMemento elementMem = selectionMem.createChild(TAG_ELEMENT);
      elementMem.putString(
        TAG_PATH,
        ((IResource) elements[i]).getFullPath().toString());
    }
  }
}
```


Observer

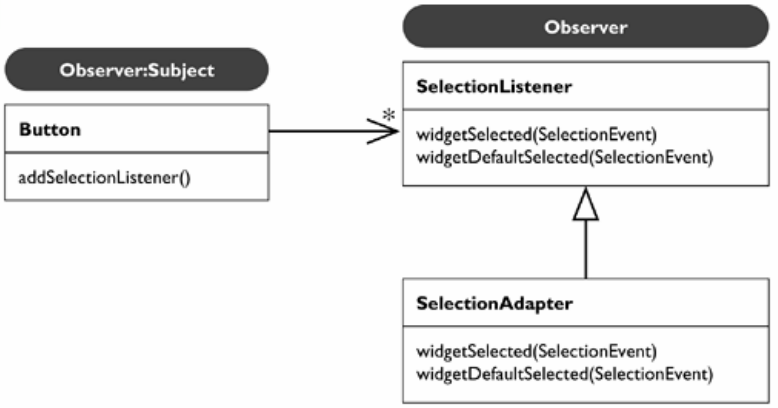
Observer (1/3)

- Problema:
  - Um utilizador carrega num *widget*. O que vem a seguir?
  - Convém acontecer alguma coisa...
  - Mas como avisar o que quer que seja que algo aconteceu?
- Solução:
  - Implementar um esquema de “se alguém te mandar executar, avisas-me e eu faço o meu trabalho”.
  - Costumamos chamar a isto *Observer*

Observer (2/3)



Observer (3/3)



Nota: isto é muito parecido com o sistema de eventos do Swing (e também apresenta semelhanças com o sistema de Delegates do .NET)

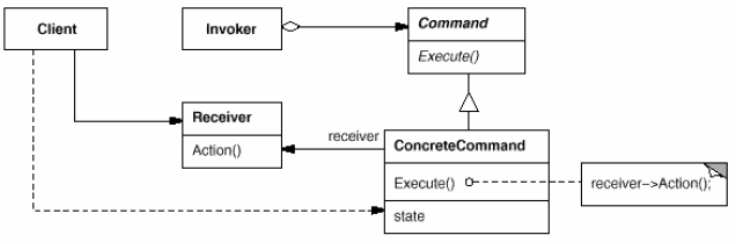
Command

Command (1/5)

- Problema:
  - Os *widgets* vão criando eventos à medida que o utilizador manuseia a ferramenta (cliques, passar com o rato por cima de algo, etc)
  - Definir como tratar um evento criado pelo utilizador
  - Esse tratamento deve ser independente do *widget* que a criou
- Solução:
  - Representar cada evento como um objecto

Command (2/5)

- Command:
  - "Encapsular um pedido como um objecto, permitindo assim:
    - Caracterizar utilizadores com diferentes pedidos
    - Guardar e/ou registar pedidos
    - Desfazer acções"

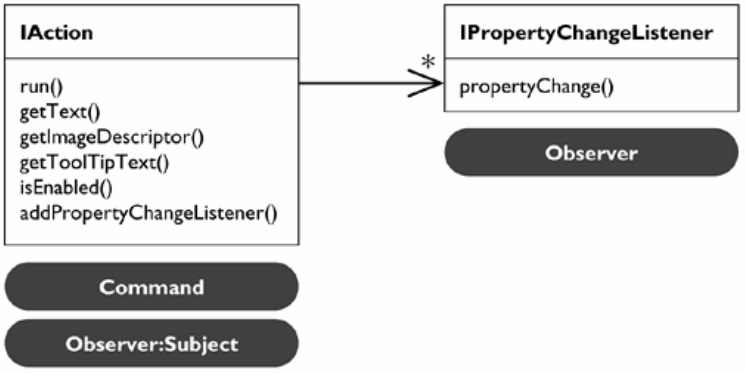


Command (3/5)

- Com este padrão, podemos representar a lógica a executar quando carregamos num botão, num menú, etc.. Esta lógica poderá:
  - Ser dependente apenas do contexto
  - Ser independente do *widget* que a criou
- O nosso *Command* pode ainda "disparar" um evento de mudança de propriedades, bem como armazenar informação sobre o estado (e sobre o *Command*); é só acrescentar isso ao código do *Command*
- O *Command* pode ainda ser reutilizado as vezes que quisermos

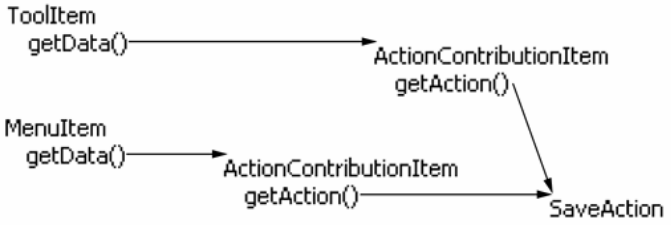
Command (4/5)

- Exemplos (1/2): Uma aplicação do *Command*



Command (5/5)

- Exemplos (2/2): Reutilização de *Command*s



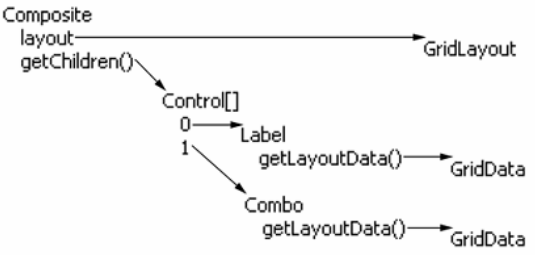
Strategy

Strategy (1/5)

- Problema:
  - Os tamanhos das etiquetas muda conforme a linguagem (PT, EN, etc.)
  - O tamanho dos *widgets* do SWT dependem da plataforma
  - Como resolver o *layout* de uma janela de modo a que não haja muito espaço desperdiçado nem haja *widgets* uns em cima dos outros?
- Solução:
  - Os *widgets* não têm posicionamento absoluto
  - Usar um *layout manager* que determine o posicionamento de cada um dos *widgets*

Strategy (2/5)

- Um exemplo de como um *layout manager* funciona:



Strategy (3/5)

- E o código para usarmos um *GridLayout*:

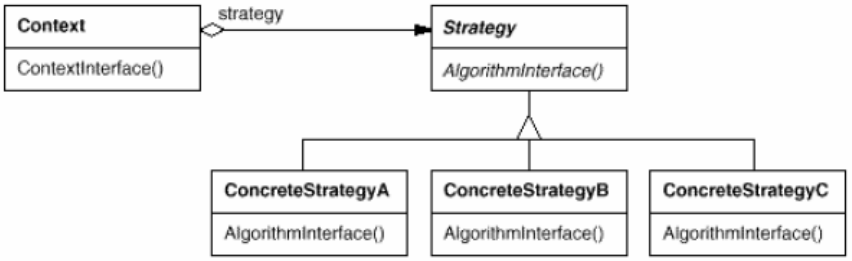
```
Composite panel= new Composite(parent, SWT.NULL);
GridLayout layout= new GridLayout();
layout.numColumns= 2;
panel.setLayout(layout);

Label findLabel= new Label(panel, SWT.LEFT);
findLabel.setText(EditorMessages.getString(
  "FindReplace.Find.label"));
setGridData(findLabel, GridData.BEGINNING, false,
  GridData.CENTER, false);
fFindField= new Combo(panel, SWT.DROP_DOWN | SWT.BORDER);
setGridData(fFindField, GridData.FILL, true,
  GridData.CENTER, false);
```

```
private void setGridData(Control component,
   int horizontalAlignment,
   boolean grabExcessHorizontalSpace,
   int verticalAlignment,
   boolean grabExcessVerticalSpace) {
  GridData gd= new GridData();
  gd.horizontalAlignment= horizontalAlignment;
  gd.grabExcessHorizontalSpace= grabExcessHorizontalSpace;
  gd.verticalAlignment= verticalAlignment;
  gd.grabExcessVerticalSpace= grabExcessVerticalSpace;
  component.setLayoutData(gd);
}
```

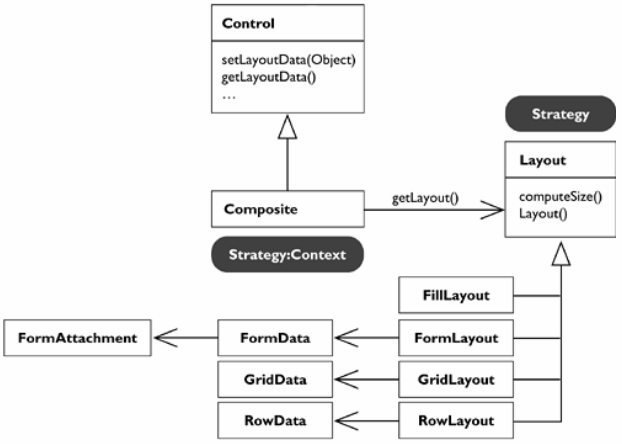

Strategy (4/5)

- Como funciona o *Strategy*:



Strategy (5/5)

- E como ele é aplicado na ferramenta:



Recapitulando...

- Vimos os padrões:
  - *Proxy* e *Bridge*
  - *Composite*
  - *Visitor*
  - *Memento*
  - *Command*
  - *Observer*
  - *Strategy*
- Vimos que eles SÃO úteis (e não num contexto meramente académico)
- E lembrem-se:

  desenvolvimento do que quer que seja sem qualquer tipo de planeamento →

Bibliografia

- Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- Kent Beck e Erich Gamma. *Contributing to Eclipse: Principles, Patterns and Plug-ins*. Addison-Wesley, 2003.
- Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

Fim

- Questões?
- Muito obrigado pela vossa presença e pela vossa atenção!