FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Padrões em Aplicações Empresariais

Arquitectura de Sistemas de Software

Luís Albino Nogueira Ramos
Mário Telmo Cabral Fonseca
Pedro Rodrigo Caetano Strecht Ribeiro

Junho 2005

Sumário

1. Introdução

Âmbito

Este trabalho foi desenvolvido no âmbito da disciplina de Arquitectura de Sistemas de Software do Mestrado em Engenharia Informática da Faculdade de Engenharia da Universidade do Porto.

Objectivo

O objectivo deste trabalho é fazer um resumo dos padrões de desenho em aplicações empresariais e, sempre que pertinente, fazer uma análise das diferenças de implementação destes padrões em duas plataformas específicas: Microsoft .NET e J2EE.

As áreas em que este trabalho incide são as aplicações empresariais (sistemas de informação), os padrões de desenho e as frameworks aplicacionais.

Aplicações Empresariais

As Aplicações Empresariais, que podem ser designadas também por Aplicações sobre Sistemas de Informação, são aplicações que lidam com a visualização, manipulação e arquivo de grandes quantidades de dados (normalmente complexos) e o suporte e automação de processos de negócio com esses mesmos dados.

Exemplos destes sistemas são sistemas de reservas, sistemas financeiros, sistemas de cadeias de fornecimento e muitos outros. Estes sistemas têm os seus próprios desafios e soluções e são diferentes de sistemas embebidos, sistemas de controlo, sistemas de telecomunicações e de aplicações de secretária.

Padrões de desenho

Os padrões de software são soluções comprovadamente boas para problemas recorrentes. Cada padrão descreve um problema que ocorre repetidamente num dado ambiente e também o cerne da solução para esse problema de tal forma que essa solução possa ser usada muitas vezes e dando particular atenção ao contexto do problema e às consequências e impacto da solução.

A ideia dos padrões foi inventada por Christopher Alexander na década de 1970s no contexto da arquitectura civil e mais tarde foram adaptados para outras áreas.

Padrões GoF

Mais tarde, a ideia dos padrões foi aproveitada para a área do software por um grupo de investigadores denominado Gang of Four que, após a publicação de um artigo, publicaram um livro com o nome "Design Patterns: Elements of Reusable Object-Oriented Software" onde descreviam os mais tarde famosos padrões GoF, isto é, uma lista base de padrões de desenho de software.

Na lista dos padrões GoF estão padrões como o Template Method, Factory Method, Observer, Wrapper, Composite, etc. Cada um destes padrões é apresentado segundo uma forma estereotipada. O conjunto destes padrões forma o cerne de uma linguagem, uma linguagem de padrões.

Padrões em Aplicações Empresariais

Ao longo do tempo foram publicados e "descobertos" muitos outros padrões de desenho de software e, mais recentemente, em 2003, Martin Fowler publicou um livro com o título: "Patterns of Enterprise Application Architecture" onde apresenta padrões para aplicações empresariais. Estes padrões descrevem soluções comprovadamente boas para problemas recorrentes no desenvolvimento de aplicações empresariais.

Padrões em Plataformas

Este trabalho tem como objectivo estudar os padrões apresentados por Fowler mas também a sua aplicação às plataformas .NET e J2EE. Para tal foram utilizados os livros "Enterprise Solution Patterns Using Microsoft .Net" da Microsoft e "Core J2EE Ptterns" por Alur, Crupi e Malks.

Nestes livros não são apresentados todos os padrões de Fowler mas apenas um pequeno conjunto, sendo sugeridos muitos outros padrões para cada uma das plataformas.

Áreas de Cobertura

Dentro da área das aplicações empresariais existem inúmeras áreas e problemáticas associadas. No seu livro, Fowler foca apenas algumas dessas áreas.

Existem muitas questões de arquitectura na construção de aplicações empresariais. Neste trabalho foram analisadas as áreas da estrutura da lógica de domínio (negócio), da interface web com o utilizador, da ligação de módulos em memória (particularmente objectos) a Bases de Dados relacionais, da gestão de sessões e dos princípios de distribuição.

Por outro lado, não foram analisadas temas como a validação, o messaging e comunicação assíncrona, a segurança, a gestão de erros, o clustering, a integração das aplicações, o refactoring ao nível da arquitectura nem as interfaces com utilizadores de rich-clients.

Grupos de Padrões

Conforme as áreas de cobertura, o trabalho está dividido nos seguintes grupos de padrões: padrões da lógica de negócio, padrões de mapeamento para BDs relacionais e padrões da camada de apresentação que reflectem a divisão em três camadas e os padrões de concorrência offline, padrões de sessões e padrões de distribuição.

Forma dos Padrões

Sempre que possível, os padrões são apresentados através das secções: nome, descrição, *how it works*, prós e contras, *when to use it*, exemplo em .NET e exemplo em Java e comparação (quando aplicável).

Plataformas

As plataformas que serão usadas para exemplificar os padrões e que, sempre que possível serão comparadas são as plataformas Microsoft .NET e J2EE.

Existe muita bibliografia relacionada com estas plataformas nomeadamente na sua descrição, utilização e até comparação. A seguinte tabela mostra as principais características/diferenças entre as plataformas:

	.NET	**J2EE**
Plataforma	Windows	Várias
Linguagem	Várias	Java
Runtime	CLR	JRE
IDE	Visual Studio .NET	Vários
Web	ASP.NET (code behind)	JSP (Beans e Tags)
Servidores Web	IIS	Apache, IIS, etc...
Acesso aos Dados	ADO.NET	JDBC
Browser Code	ActiveX	Applets
Componentes	COM + Managed Components	EJB
Invocação Remota	.NET Remoting	RMI
Messaging	MSMQ	JMS

Divisão em camadas

Um dos pressupostos no livro de Fowler e neste trabalho é a estruturação das aplicações empresariais em camadas. As vantagens da divisão em camadas são:

- Possibilidade de compreender uma única camada como um todo coerente sem se saber muito sobre as outras camadas
- Possibilidade de substituição de camadas por implementações alternativas dos mesmos serviços
- Minimização de dependências entre camadas
- Promovem a standardização
- Com uma camada construída, é possível usá-la para muitos serviços de mais alto nível

As desvantagens da divisão em camadas são:

- Impossibilidade de encapsulamento totalmente correcto, por exemplo, as alterações em cascata: adicionando um campo na Base de Dados que é necessário mostrar na interface, é necessário actualizar todas as camadas entre a Base de Dados e a interface
- Demasiadas camadas podem por em causa a performance. Em cada camada os dados necessitam de ser transformados de uma representação para outra. No entanto, o encapsulamento de um função muitas vezes trás ganhos de eficiência que mais do que compensam a potencial perda de performance. Por exemplo, uma camada que controla as transacções pode ser optimizada fazendo com que toda a aplicação fique mais rápida

A principal dificuldade da divisão em camadas é a decisão de que camadas usar e quais as suas responsabilidades.

As 3 camadas base dos sistemas de aplicação empresarial são:

- Apresentação: aprovisionamento de serviços, visualização de informação (em Windows ou em HTML), gestão dos pedidos do utilizador (eventos do rato e do teclado), pedidos http, invocações da linha de comando, etc.
- Lógica de domínio: a lógica que é o principal ponto da aplicação
- Fonte de Dados: comunicação com a Base de Dados, sistemas de messaging, gestores de transacções e outros pacotes

Existem múltiplos esquemas de divisão em camadas como por exemplo as 5 camadas de Brown (Presentation, Controller/Mediator (pattern Application Controller), Domain, Data mapping (pattern Data Mapper) and Data Source).

Dado que o comportamento da lógica de negócio pode mudar muito, é importante ser possível modificar, evoluir, testar esta camada facilmente. Sendo assim, uma ligação mínima entre a camada de domínio e as outras camadas é desejada. Uma das grandes forças motrizes de padrões de layering é separar a camada de domínio das outras camadas.

Um teste informal para verificar a separação entre camadas e as suas responsabilidades é imaginar a adição de uma camada completamente diferente à aplicação. Por exemplo, adicionar uma interface de linha de comando a uma aplicação web. Se houver muitas funcionalidades a duplicar, isso seria um sinal de que a lógica do domínio entrou na camada de apresentação. De igual modo, há duplicação de lógica de negócio para substituir uma Base de Dados por um ficheiro de XML?

2. Padrões da Lógica de Negócio

A camada da lógica de negócio contém em si a responsabilidade de implementar aquilo que a aplicação realmente faz, isto é, os algoritmos de cálculo e manipulação de dados.

Para a camada da lógica de negócio existem três padrões principais: *Transaction Script*, *Domain Model* e *Table Module*. Apesar de estes padrões poderem ser conjugados é normal optar pela utilização de um deles.

Adicionalmente existe o padrão Service Layer que pode ser usado em algumas circunstâncias específicas.

2.1. Transaction Script

O padrão *Transaction Script* organiza a lógica de negócio em procedimentos. Cada procedimento trata um único pedido da camada de apresentação. O procedimento recebe *inputs* da camada de apresentação, processa-o com validações e cálculos, armazena dados numa BD e/ou invoca operações de outros sistemas. Depois entrega dados novamente à camada de apresentação, eventualmente efectuando mais cálculos para organizar e formatar a resposta.

A organização fundamental é a de um procedimento para cada acção do utilizador, daí que se possa encarar este padrão como um *script* para uma acção ou transacção de negócio. Não é necessariamente um procedimento em termos de código, já que pode invocar sub rotinas partilhadas por outros *scripts*. A ideia fundamental é oferecer *scripts* sob a forma de acções. Um exemplo é um sistema de um supermercado em que poderia haver um *script* para colocar um objecto no cesto de compras, outro para indicar o total actual das compras e outro para fazer *checkout*.

A maioria das aplicações empresariais pode ser vista como um conjunto de transacções. Todas as interacções entre cliente e servidor contêm alguma lógica. Em alguns casos pode ser tão simples como mostrar dados guardados na base de dados. Noutros pode envolver várias validações e cálculos.

O Transaction Script organiza tudo em volta destes procedimentos que fazem chamadas directas (ou quase, no caso de utilizar wrappers) à Base de Dados. Cada transacção tem um procedimento associado.

How it Works

Com este padrão, na reserva de um quarto de hotel, a lógica de verificação de disponibilidade, cálculo de preços e actualização da base de dados é encontrada num procedimento de Reserva de Quarto de Hotel. A vantagem deste esquema é que não há necessidade de preocupação com o que as outras transacções estão a fazer.

O Transaction Script é usualmente colocado num CGI script mas pode ser colocado numa Server Page, num objecto de sessão distribuído ou mesmo num stored procedure. Esta opção depende da estrutura em camadas usada.

Os Transactions Scripts devem ser o mais separados possível, para tal, devem ser estruturados em módulos, em classes diferentes, em diferentes sub rotinas e/ou também separados da camada de apresentação e da de acesso a dados. No caso da divisão em camadas, as chamadas dos Transaction Scripts à camada de apresentação são proibidas!

Quanto à separação dos scripts por classes existem duas alternativas: a colocação de vários scripts numa mesma classe de área/assunto relacionado ou a

criação de uma classe para cada script através da implementação do Padrão GoF Command (método run).

Prós e Contras

As principais vantagens do padrão *Transaction Script* são as seguintes:

- É simples;
- É a organização natural para aplicações com pouca lógica de domínio;
- O modelo procedimental é familiar para a maioria dos programadores, o que o torna fácil de usar e compreender;
- Insere um overhead mínimo em termos de performance;
- Adere bem a uma simples camada de acesso a dados utilizando os padrões *Row Data Gateway* ou *Table Data Gateway*;
- Torna óbvio quais são as fronteiras de uma transacção, sendo adequado para ferramentas que as efectuam como processos de *background*.

Por outro lado, as desvantagens deste padrão são:

- A complexidade das soluções aumenta exponencialmente com o aumento da complexidade da lógica de negócio e torna-se difícil manter o bom design;
- Aumento da probabilidade de duplicação de código quando várias transacções efectuam tarefas similares. Algumas situações podem ser evitadas agrupando o código duplicado em sub rotinas, mas em outros casos a duplicação é de remoção mais complicada e de difícil detecção. A aplicação resultante corre o risco de se tornar uma teia de rotinas sem evidenciar a sua estrutura.

When to Use It

Em última análise, o padrão Domain Model oferece muitas mais opções de estruturação do código, aumentando a legibilidade e facilidade de compreensão e diminuindo a duplicação de código. No entanto, existem muitos problemas simples para resolver com o padrão Transaction Script e uma solução simples é mais rápida de construir.

Implementação em VB.NET

Como já foi dito, o Transaction Script pode ser implementando com várias scripts/transacções por classe/componente ou com uma transacção por classe (através da implementação do padrão GoF Command). De seguida é apresentado um exemplo em VB.NET para cada uma destas soluções.

Múltiplas transacções por classe

É a técnica mais comum e envolve agrupar as transacções e criar uma classe com um método público para cada transacção, isto é, criar um componente de negócio que contém um método público para cada script. Por exemplo, numa aplicação de processamento de encomendas, o processo de encomendar um produto implica vários passos e pode ser encapsulado num método efectuarEncomenda de um componente processarEncomendas:

```
Public Class processarEncomendas
  Public Function efectuarEncomenda(ByVal encomenda As EncInfo) As Long
        ' Iniciar transacção
        ' Verificar inventário
        ' Obter informação do cliente, verificar estado do crédito
        ' Calcular preço e imposto
        ' Calcular Valor de envio e total da encomenda
        ' Guardar encomenda na BD e fazer COMMIT da transacção
        ' Enviar e-mail de confirmação da encomenda
        ' Retornar o novo ID da encomenda
    End Function
End Class
```

Neste caso, a informação da encomenda é passada ao método `efectuarEncomenda` numa estrutura `EncInfo` definida da seguinte forma:

```
Public Structure EncInfo
   Public ProdutoID As Long
   Public Quantidade As Integer
   Public ClienteId As Long
   Public EnvioVia As TipoEnvio
End Structure

Public Enum TipoEnvio
  FedEx
  UPS
  Postal
End Enum
```

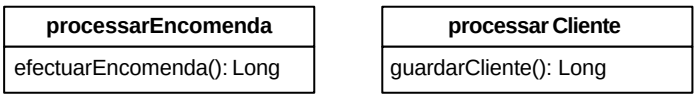
De forma idêntica, o processamento de um cliente pode ser encapsulado numa classe componente `processarCliente`:

```
Public Class processarCliente
  Public Function guardarCliente(ByVal cliente As ClienteInfo) As Long
        ' Validar endereço
        ' Iniciar transacção
        ' Pesquisar clientes duplicados a partir do endereço de e-mail
        ' Guardar cliente na BD e fazer COMMIT da transacção
        ' Retornar o novo ID do cliente
  End Function
End Class

Public Structure ClienteInfo
  Public Nome As String
  Public Endereco As String
  Public Cidade As String
  Public Estado As String
  Public CodigoPostal As String
  Public Email As String
End Structure
```

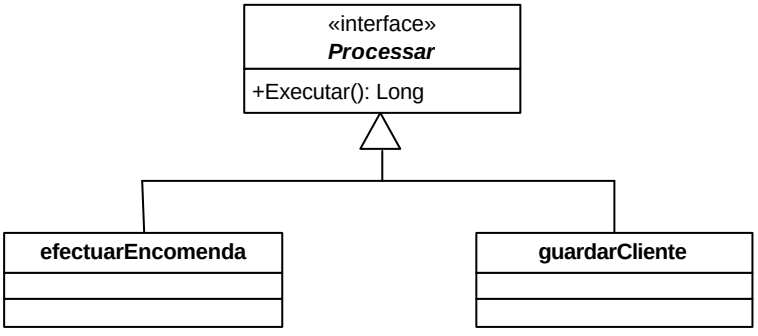
O processo do utilizador ou outros componentes que tratam da interface seriam responsáveis por invocar o método GuardarCliente, retendo o ID do cliente, criando uma estrutura EncInfo e passando-a ao método efectuarEncomenda. De notar que a informação sobre o produto já seria conhecida, uma vez que a interface do utilizador teria invocado um método da classe `processarEncomendas` para obter os produtos.

O esquema seguinte mostra os componentes e métodos deste exemplo:



Uma transacção por classe

Nesta segunda técnica, cada Transaction Script é implementado na sua própria classe. Por exemplo, os scripts `efectuarEncomenda` e `GuardarCliente` podem ser implementados como classes que herdam da interface `Processar`:



A implementação do componente `efectuarEncomenda` poderia ser o seguinte:

```
Public Interface Processar
  Function Executar() As Long
End Interface

Public Class efectuarEncomenda : Implements Processar
    Private _encomenda As EncInfo
    Public Sub New(ByVal encomenda As EncInfo)
       _ encomenda = encomenda
    End Sub
    Public Function Executar() As Long Implements Processar.Executar
        ' Iniciar transacção
        ' Verificar inventário
        ' Obter informação do cliente, verificar estado do crédito
        ' Calcular preço e imposto
        ' Calcular Valor de envoi e total da encomenda
        ' Guardar encomenda na base de dados e cometer transacção
        ' Enviar e-mail de confirmação da encomenda
        ' Retornar o novo ID da encomenda
    End Function
End Class
```

Este desenho é baseado no padrão Command documentado nos padrões GoF e permite tratar os scripts de forma polimórfica invocando o método Executar da interface Processar. Adicionalmente, os componentes da interface do utilizador podiam utilizar o padrão Factory para criar os objectos respectivos.

A lógica para executar os vários passos nestes esboços de código pode ser implementada directamente na classe ou invocando stored procedures. Esta última hipótese tem a vantagem de introduzir mais uma camada de abstracção e tirar partido dos mecanismos de optimização de desempenho das bases de dados.

Implementação em Java

A implementação deste padrão na plataforma J2EE não engloba qualquer diferença face à plataforma .NET dado que esta não se baseia em nenhuma característica de plataforma.

2.2. Domain Model

O padrão Domain Model é um modelo de objectos do domínio que incorpora o comportamento e os dados da aplicação.

A lógica de negócio de uma aplicação empresarial pode tornar-se bastante complexa dadas as regras, cálculos e os diferentes casos no comportamento. O paradigma orientado a objectos é comprovadamente uma boa solução para esta complexidade e a forma OO de organizar a lógica de negócio é o padrão *Domain Model*.

O padrão Domain Model cria uma rede de objectos interligados, aonde cada objecto representa um indivíduo com significado por si, tão grande como uma empresa ou pequeno como uma linha num formulário de encomenda. A lógica de negócio é colocada dentro desses objectos de domínio.

Por exemplo, um sistema de *leasing*, teria classes com os nomes 'Empréstimo', Prestação', Pagamento', etc. A lógica para lidar com validações e cálculos seria colocada neste modelo, pelo que, um objecto de 'Envio' incluiria toda a lógica necessária para calcular o custo de envio de uma encomenda. Neste objecto, podiam ainda existir rotinas para cálculo de uma factura, mas esse tipo de procedimentos rapidamente seriam convertidos num método de um objecto. Estes métodos chama-se *Domain Model methods*.

A utilização do padrão *Domain Model* em contraste com o *Transaction Script* assenta na mudança do paradigma imperativo para o orientado a objectos. O foco do modelo desvia-se das acções (verbos) para os objectos (nomes). Deste modo,

em vez de ser uma sub rotina a ter toda a lógica para uma acção do utilizador, as partes dessa lógica são distribuídas pelos objectos de tal forma que cada um tenha a parte que é relevante para si.

How It Works

Usar o padrão Domain Model implica normalmente incluir uma camada inteira de objectos que modelam a área de negócio da aplicação. Nesta camada existirão objectos que representam os dados do negócio (dados) e capturam as regras do negócio (comportamento). Este padrão junta dados e procedimentos numa rede complexa de associações em que usa o mecanismo de herança.

Um Domain Model OO é muitas vezes semelhante a um modelo de BD mas pode conter bastantes diferenças. Quanto a isto, existem dois estilos de Domain Model:

- Um Domain Model simples que é semelhante ao esquema de BD, em que existe um objecto de domínio para cada tabela da BD.
- Um Domain Model rico que pode ser bastante diferente do esquema da BD, com herança, estratégias, e outros padrões GOF e redes complexas de pequenos objectos interligados. Um modelo de dados rico é melhor para uma lógica complexa mas é mais difícil de mapear para uma base de dados.

Um Domain Model simples pode usar o padrão Active Record enquanto que um Domain Model rico requer o uso do padrão Data Mapper.

Usualmente é conjugado com o Domain Model o padrão Layer Supertype. O padrão Layer Supertype é um padrão bastante simples que define que todos os objectos de uma camada podem ser sub classes de uma super classe. Esta super classe pode implementar funcionalidades que são transversais à camada. No caso do Domain Model, esta classe pode ser usada para implementar o mapeamento para a Base de Dados e outros detalhes associados como a gestão dos IDs dos objectos na Base de Dados.

Prós e Contras

A grande vantagem no uso do padrão *Domain Model* é que este é mais adequado para lidar com o aumento progressivo da complexidade da lógica de negócio, já que esta vai sendo distribuída pelas várias classes (oferece soluções relativamente simples para problemas complexos).

Por outro lado, os modelos de domínio mais ricos são mais difíceis de compreender mas a curva de aprendizagem para programadores habituados a trabalhar numa lógica semelhante à do *Transaction Script* é longa mas recompensadora. A grande desvantagem do padrão Domain Model é que a ligação à Base de Dados é complicada, pois é sempre necessário fazer um mapeamento entre o modelo de objectos e o modelo relacional.

When to Use It

O conselho básico quanto ao Domain Model é que se a lógica de negócio for simples se devem usar Transaction Scripts, caso contrário, o Domain Model é mais aconselhado em quase todas as situações.

Um dos factores mais importantes é o conforto com que a equipa de desenvolvimento tem com objectos de domínio. Adoptar o padrão Domain Model com uma equipa inexperiente é considerado um grande mas recompensador esforço.

Quando se usa o padrão Domain Model, a interacção com a Base de Dados deve ser feita usando um Data Mapper. Adicionalmente pode ser usado o padrão Service Layer para definir melhor a API do modelo do domínio.

Implementação em Java

Existem bastantes dúvidas sobre qual a melhor forma de implementar um Domain Model em J2EE. Muito material de ensino sugere a utilização de Entity

Beans para implementar os Domain Models mas Fowler considera que existem vários problemas graves nesta abordagem.

Os Entity Beans são mais úteis quando é usado Container Managed Persistence (CMP). De facto, segundo Fowler, é muito pouco útil usar Entity Beans sem CMP, para além de que CMP é uma forma limitada de mapeamento objecto-relacional e não suporta muitos dos padrões necessários num Domain Model rico.

Uma das limitações dos Entity Beans é que não suportam a reentrada, isto é, se um Entity Bean chamar outro objecto, esse objecto (ou qualquer outro objecto que este chame) não pode voltar a chamar o primeiro bean. Esta limitação é considerada grave por Fowler pois um Domain Model rico utiliza frequentemente a reentrada. A reentrada é difícil de detectar pelo que alguns especialistas aconselham a que um Entity Bean nunca faça uma chamada a outro, o que, apesar de resolver o problema inicial, diminui as vantagens de usar o próprio Domain Model.

Para utilizar Entity Beans é necessário um contentor de beans e uma base de dados o que aumenta o tempo de construção e de testes. Para além disso, é difícil fazer debug a Entity beans.

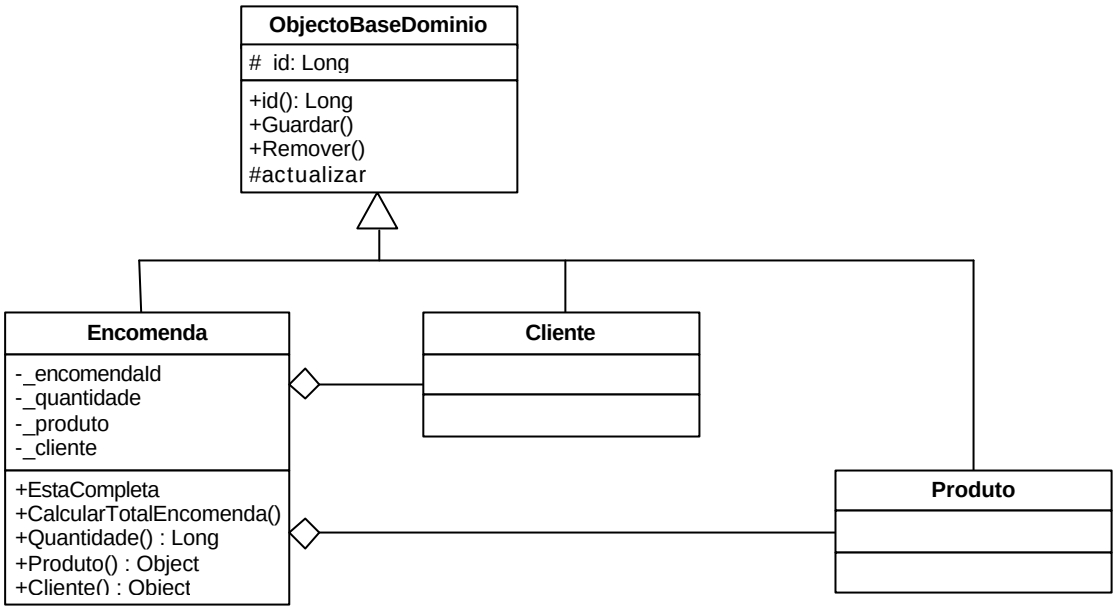
Segundo Fowler, só se deve usar Entity Beans se a lógica de domínio é reduzida. Nesse caso existirá aproximadamente um Entity Bean para cada tabela da base de dados. Modelos mais ricos com heranças, estratégias e outros padrões mais sofisticados devem ser implementados com objectos Java normais e um Data Mapper. O Data Mapper pode ser construído usando uma ferramenta comercial de mapeamento ou com uma camada construída para a aplicação.

Outro dos problemas dos EJBs é que estes obrigam a que o programador tenha de pensar não só no modelo de domínio mas também no ambiente de desenvolvimento EJBs.

A alternativa aos Entity Beans, no Domain Model, são os objectos Java normais. Um Domain Model com objectos normais é fácil de montar, rápido de construir, e pode ser testado independentemente de um contentor de EJBs. Fowler afirma que o facto destes objectos serem independentes de um contentor EJB faz com que os vendedores não aconselhem a não usar EJBs mas, por exemplo, nos manuais de utilização do contentores de EJBs da Oracle é aconselhado o uso de EJBs para modelar lógicas complexas com relações complexas. Partindo do princípio que a solução é usar objectos Java normais, deixam de existir diferenças na implementação deste padrão para a plataforma .NET, na qual se baseia o próximo exemplo.

Implementação em .NET

A figura seguinte mostra o diagrama de classes da solução Domain Model do problema das encomendas visto no padrão Transaction Script:



Esta figura mostra a estrutura dos vários objectos individuais e interrelacionados que representa o Domain Model.

Neste exemplo é usado o padrão Layer Supertype já referido anteriormente, isto é, existe uma classe chamada ObjectoBaseDominio da qual todos os objectos de domínio herdam que implementa o mapeamento para a Base de Dados e a gestão de IDs para todos os objectos de domínio.

A implementação do Objecto Base do Domínio é a seguinte (nota: MustInherit identifica uma classe abstracta e MustOverride identifica um método abstracto ou virtual):

```
Public MustInherit Class ObjectoBaseDominio
    Protected _id As Integer
    Public Property Id() As Integer
        Get
            Return _id
        End Get
        Set(ByVal Value As Integer)
            _id = value
        End Set
    End Property

   MustOverride Function Guardar() As Boolean
   MustOverride Function Remover() As Boolean

   Protected actualizar As Boolean = False

   (...)
End Class
```

A implementação da classe Encomenda é a seguinte:

```
Public Class Encomenda : Inherits ObjectoBaseDominio
    Private _encomendaId As Long
    Private _cliente As Cliente
    Private _produto As Produto
    Private _quantidade As Integer = 1 'default
    Private _envio As TipoEnvio = TipoEnvio.Postal 'default
    Public Sub Nova()
    End Sub
    Public Sub Nova(ByVal cliente As Cliente, ByVal produto As Produto)
       _InitClass(cliente, produto, Nothing, Nothing
    End Sub
    Public Sub Nova(ByVal cliente As Cliente, _
                    ByVal produto As Produto, ByVal quantidade As Integer)
       _InitClass(cliente, produto, quantity, Nothing)
```

```
    End Sub
    Public Sub Nova(ByVal cliente As Cliente, ByVal produto As Produto, _
                    ByVal quantidade As Integer, ByVal envoi As TipoEnvio)
       _InitClass(cliente, produto, quantidade, envio)
    End Sub
    Private Sub _InicializarClasse(ByVal cliente As Cliente, _
                        ByVal produto As Produto, ByVal quantidade As Integer, _
                        ByVal envio As TipoEnvio)
        _cliente = cliente
        _produto = produto
        Me.Quantidade = quantidade
        Me.EnvioVia = envio
        Me.actualizar = True
        ' Gerar ID encomenda novo ou temp.: usar chave atribuida pelo sistema
        ' _encomendaId = key table  GUID
    End Sub
    Public ReadOnly Property Cliente() As Cliente
        Get
            Return _cliente
        End Get
    End Property

    Public ReadOnly Property Produto() As Produto
        Get
            Return _produto
        End Get
    End Property

    Public Property Quantidade() As Integer
        Get
            Return _quantidade
        End Get
        Set(ByVal Valor As Integer)
            If Valor < 0 Then
                Throw New ArgumentOutOfRangeException( _
                 "A quantity tem que ser superior a 0")
            End If
            _quantidade = Value
           Me.actualizar = True
        End Set
    End Property

    Public Property EnvioVia() As TipoEnvio
        Get
            Return _envio
        End Get
        Set(ByVal Valor As TipoEnvio)
            _envio = Valor
            Me.actualizar = True
        End Set
    End Property

    Public Function CalcularCustoEnvio() As Double
        ' calcular o custom de envio baseado no cliente e produto
        ' guardar o custo de envio desta encomenda
    End Function

    Public Function CalcularTotalEncomenda() As Double
        ' calcular o total da encomenda com o imposto
        ' guarder o total desta encomenda
    End Function

    Public Function EstaCompleta() As Boolean
        ' Verifica se esta encomenda tem a informação necessária para ser
guardada
    End Function

    Public Overrides Function Guardar() As Boolean
        ' Guardar a encomenda
        Me.actualizar = False
    End Function

    Public Overrides Function Remover() As Boolean
        ' Remover a encomenda
    End Function
End Class
```

A ideia fundamental desta classe é combinar os dados (as propriedades Quantidade e EnvioVia), o comportamento (os métodos EstaCompleta, Guardar, CalcularTotalEncomenda, etc) e as relações com outros dados (as propriedades Produto e Cliente que se relacionam com as classes com os mesmos nomes).

Com a utilização do padrão Layer SuperType a gestão do IDs das encomendas está assegurada.

2.3. Table Module

A terceira forma de estruturar a lógica de negócio é utilizando o padrão *Table Module*. O padrão *Table Module* é de certa forma uma solução intermédia entre o *Transaction Script* e o *Domain Model*. Este padrão organiza a lógica de domínio com uma classe para cada tabela que contém todos os procedimentos com a lógica.

A grande diferença entre este padrão e o padrão Domain Module reside na interacção com uma base de dados.

A abordagem OO tradicional é baseada em objectos com identidade, na linha do padrão Domain Model, isto é, se existir uma classe Empregado, cada instância dessa classe representa um empregado. A vantagem deste esquema é que, uma vez obtida uma referência para o objecto podemos executar operações, seguir relações e recolher dados sobre ele. No padrão Table Module cada objecto corresponde a um conjunto de registos de uma tabela, obtidos a partir de um *Record Set*. A manipulação dos objectos é um pouco diferente pois, para manipular um registo em particular é necessário indicar um ID.

How It Works

À superfície um Table Module parece um objecto normal, a principal diferença é que nele não existe a noção de identidade de objecto com que trabalha.

Por exemplo, para obter o endereço de um Empregado, é usado um método como anEmployeeModule.getAddress(long employeeID). Sempre que se queira fazer algo a um Empregado em particular é necessário passar uma referência para a entidade, normalmente, é usada uma primary key da base de dados.

Normalmente este padrão é usado com estruturas de dados que são orientadas às tabelas. Os dados em formato tabular são normalmente o resultado de instruções SQL a que são guardados em Record Sets que imitam tabelas SQL.

O padrão Table Module oferece uma interface explícita baseada em métodos de interacção com os dados.

Frequentemente é necessário usar o comportamento de múltiplos Table Modules. Por isso, é usual vários Table Modules operarem sobre o mesmo Record Set.

O Table Module pode ser uma instância ou pode ser um grupo de métodos estáticos. A vantagem de usar uma instância é que se pode inicializar o Table Module com um Record Set já existente. Por exemplo, o resultado de uma query. O Table Module pode depois ser usado para manipular os dados. As instâncias tornam também possível o uso dos mecanismos de herança.

Um Table Module pode incluir queries como Factory Methods. A alternativa é uma Table Data Gateway que tem a desvantagem de ser necessária uma classe Table Data Gateway e o seu mecanismo associado. A vantagem do Table Data Gateway é que se pode usar um único Table Module para dados de diferentes origens dado que se usa um Table Data Gateway para cada fonte de dados.

Para usar um Table Data Gateway neste contexto a aplicação cria um Tabela Data Gateway para construir um Record Set que é usado para criar o Table Module. Se é necessário comportamento de vários Table Modules, estes podem ser criados com o mesmo Record Set. O Table Module pode finalmente processar a lógica de negócio e entregar o Record Set à camada de apresentação para a visualização e edição usando componentes de interface específicos para manipulação de tabelas. Depois de editados, os dados voltam ao Table Module para validação antes de

serem gravados na Base de Dados. Uma das vantagens deste esquema é que os Table Modules podem ser testados criando um Record Set em memória sem recorrer a uma Base de Dados.

Apesar do nome do padrão incluir a palavra Table e o exemplo mais óbvio do Table Module é a criação de um Table Module para cada tabela, é também possível e útil ter Table Modules para vistas ou outras queries. A estrutura do Table Module não depende directamente das tabelas na Base de Dados mas das tabelas virtuais com que a aplicação lida.

Prós e Contras

A grande vantagem do padrão Table Module é que permite empacotar os dados e o comportamento e, ao mesmo tempo, utilizar o poder das bases de dados relacionais.

Relativamente ao padrão Transaction Script, este padrão tem a vantagem de basear a organização da lógica de negócio em tabelas em vez de procedimentos o que impõe uma maior estruturação e facilita a identificação e remoção de duplicação. No entanto, não é possível utilizar muitas das técnicas do *Domain Model* para maior controlo dessa lógica, como herança e outras, típicas de padrões orientados a objectos.

Um dos grandes problemas do padrão Domain Model, que este Padrão vem resolver, é a interface com as bases de dados relacionais. O *Table Module* adere muito facilmente ao resto da arquitectura. Muitos ambientes gráficos estão concebidos de tal forma que os resultados de uma interrogação SQL sejam armazenados em *Record Sets*. Como o *Table Module* trabalha directamente com estes, torna-se simples executar uma interrogação, manipular os resultados e passá-los novamente ao ambiente gráfico para a sua apresentação.

When to Use It

Este padrão é muito baseado em dados orientados à tabela pelo que é obviamente útil em situações em que se está a lidar com dados tabulares através de Record Sets.

No entanto, os Table Modules não oferecem todo o poder dos objectos na organização de lógicas complexas. Algumas das incapacidades deste esquema são a impossibilidade de ter relações directas de instância para instância ou polimorfismo. Portanto, para lidar com lógicas de domínio complicadas, o padrão Domain Model é a melhor escolha.

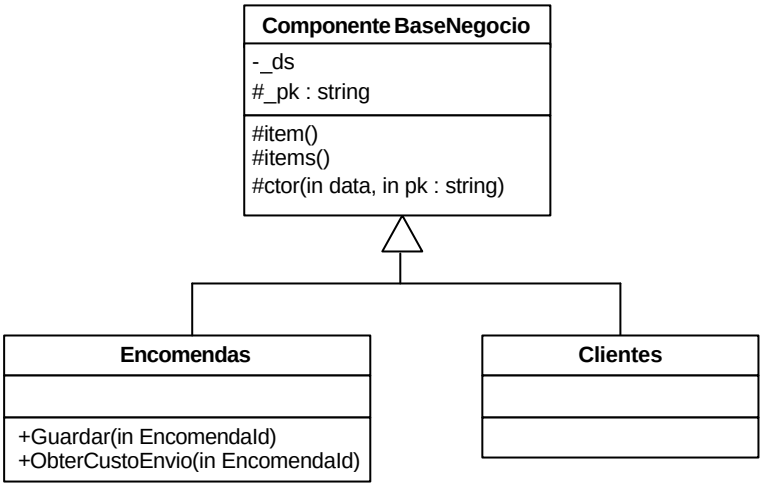
Essencialmente, é necessário negociar a habilidade do padrão Domain Model de lidar com lógicas complexas contra a facilidade do Table Module de integração com estruturas de dados orientadas às tabelas.

Quando os objectos do Domain Model e as tabelas da base de dados são relativamente semelhantes, é melhor usar o padrão Domain Model que use o padrão Active Record. Os Table Modules funcionam melhor que a combinação anterior quando outras partes da aplicação são baseadas em estruturas de dados orientadas às tabelas. Esta é a razão pela qual não se vê o uso de Table Module em ambientes Java.

A situação em que este padrão se encaixa melhor, em termos de plataforma, é no .NET, em que os Record Sets são os repositórios primários de dados numa aplicação. As bibliotecas ADO da Microsoft oferecem bons mecanismos para aceder a dados relacionais com record sets. Nesta situação Table Module permite inserir a lógica de negócio na aplicação de uma forma organizada.

Implementação .NET

Para implementar uma abordagem Table Module na aplicação de processamento de encomendas pode ser criada uma classe abstracta CompenteBaseNegocio como a apresentada no esquema seguinte. Esta classe é responsável por aceitar um DataSet sobre o qual a classe pode trabalhar, expondo registos individuais do DataSet utilizando um indexador.



Como é ilustrado, uma das vantagens de utilizar o Table Module em vez do Transaction Script é que facilmente encapsula as responsabilidades nos objectos apropriados. A classe Clientes contém os métodos EnviarEmail e VerificarCredito que podem ser invocados independentemente.

O código para implementar a classe ComponenteBaseNegocio é o seguinte:

```
Public MustInherit Class ComponenteBaseNegocio
    Private _ds As DataSet
    Private _pk As String
    Protected Sub New(ByVal data As DataSet)
        _ds = data
        _pk = _ds.Tables(0).PrimaryKey(0).ColumnName
    End Sub
    Default Protected ReadOnly Property Item(ByVal id As Integer) As DataRow
        Get
            Dim f As String = _pk & " = " & id
            Return _ds.Tables(0).Select(f)(0)
        End Get
    End Property
    Protected ReadOnly Property Items() As DataSet
        Get
            Return _ds
        End Get
    End Property
End Class
```

Neste código, podemos ver que o construtor da classe encontra a coluna da chave primária e guarda-a numa variável privada usada pela propriedade Item para seleccionar e retornar um DataRow específico. Esta classe pode ser herdada pelas sub classes Clientes e Encomendas. A classe Encomendas em C# é a seguinte:

```
public class Encomenda : ComponenteBaseNegocio
{
  public new EncomendasDs Items
  {    get { return (EncomendasDs)base.Items;  }
  }
  public Orders(EncomendasDs encomendas):base(encomendas)
  { }
  public TipoEnvio GetShipType(long EncomendaId)
  {
    // Retorna o tipo de envio para a encomenda
  }
  public double ObterCustoEnvio(long EncomendaId)
  {
    // Calcula os custos de envio para a encomenda
  }
  public  void Guardar(long EncomendaId)
  {
    // Guarda a encomenda na base de dados
```

```
    }
    public void Guardar()
    {
      // Guarda todas as encomendas na base de dados
    }
    public long Inserir(long produtoId, int clienteId,
                        long quantidade, TipoEnvio envio)
    {
      // Insere uma nova linha no DataSet
    }
  }
```

O construtor desta classe aceita um DataSet de tipo EncomendasDs que é um Data Set tipado.

Um exemplo do uso deste modelo é a instanciação da classe Encomendas em que é passado o DataSet do tipo EncomendasDs:

```
Encomendas e = new Encomendas(dsEncomendas); e.Guardar(123324);
```

Implementação em Java

Neste padrão podemos verificar uma diferença importante entre a duas plataformas, pois na plataforma J2EE não existe a classe DataSet que permite manipular os resultados das queries facilmente. De igual forma, não foi visto no exemplo anterior que as facilidades de manipulação e visualização destes DataSets em .NET estão muito facilitadas pela framework.

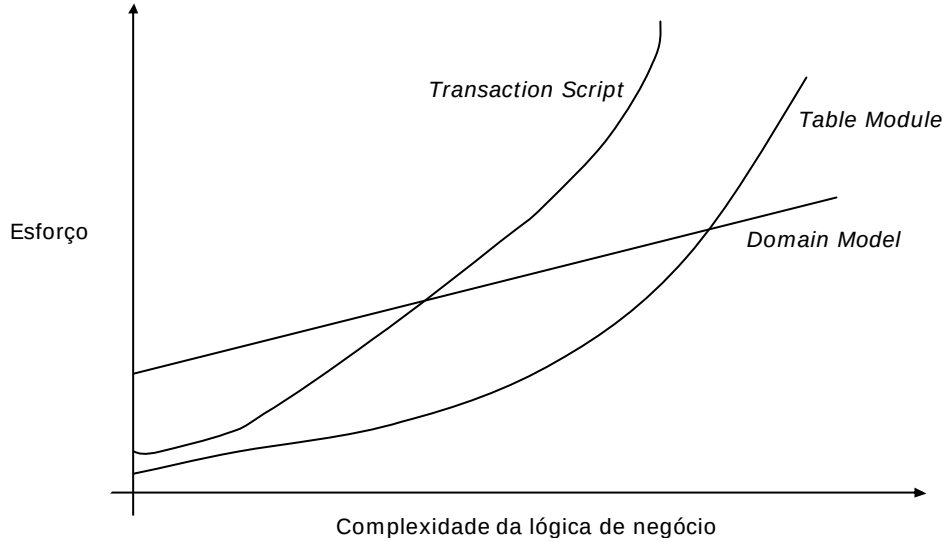
2.4. Optar por um dos 3 padrões

A decisão da adopção de um dos três padrões para estruturar a lógica de negócio depende essencialmente do tipo de problema que se tem em mãos e da complexidade dessa lógica. Em casos de muita simplicidade, não se obtém vantagens do *Domain Model* devido a um esforço adicional de desenvolvimento, pelo que será preferível uma solução baseada em *Transaction Script*. No entanto, se for previsível que a complexidade possa progressivamente aumentar, será preferível fazer esse investimento e começar a modelar em *Domain Model*, na expectativa de obter benefícios disso mais tarde.

A escolha pela adopção do *Table Module* pode estar condicionada pelo ambiente de desenvolvimento suportar *Record Sets*. Ambientes como o .Net ou Visual Studio, em que muitas ferramentas utilizam *Record Sets*, tornam mais favorável e atractiva a adopção do *Table Module* em vez do *Domain Model*. Por outro lado, não parecem existir quaisquer motivos para utilizar *Transaction Script* em .Net. Em ambientes em que não existam ferramentas que trabalhem com *Record Sets*, não existirão consequentemente vantagens em utilizar *Table Module*.

Os três padrões não são mutuamente exclusivos, sendo até comum encontrar a utilização de *Transaction Script* para umas partes da lógica de negócio e *Table Module* e *Domain Logic* para as restantes.

A seguinte figura mostra, para cada padrão de organização da lógica do domínio, o esforço necessário para lidar com o crescimento da complexidade das lógicas.



Verifica-se que o *Domain Model* é o padrão que melhor acomoda incrementos na complexidade da lógica de negócio. O *Transaction Script*, pelo contrário apresenta um limite para o qual é impossível introduzir mais complexidade. É importante compreender até que ponto poderá a lógica de negócio aumentar de complexidade dado que numa fase mais adiantada de um projecto pode ser demasiado tarde para voltar para trás. Por outro lado, não faz sentido estruturar um problema de complexidade garantidamente baixa com um padrão tão rico como o *Domain Model*.

2.5. Service Layer

Uma abordagem comum para lidar com a lógica de negócio é dividi-la em duas camadas: uma camada, designada *Service Layer*, sobreposta a uma *Domain Model* ou *Table Module*. Esta abordagem não faz sentido com *Transaction Script* pois a baixa complexidade não o justifica.

Este padrão define as fronteiras de uma aplicação com uma camada de serviços que estabelece uma série de operações e coordena a resposta da aplicação em cada operação.

Toda a lógica de apresentação interactua com a lógica de negócio através desta nova camada que se comporta como uma API para a aplicação. Para além desta faceta, a *Service Layer* é o melhor local para colocar validações sobre controlo de transacções e segurança. Tal fornece um modelo simples para colocar cada método nesta camada e descrever as suas características transaccionais e de segurança. Outra hipótese seria ter um ficheiro de propriedade à parte, mas os atributos do .Net fornecem uma boa forma de o fazer directamente no código.

Quando se decide implementar uma *Service Layer*, um aspecto fundamental a decidir é que parte da lógica se deve incluir. No limite mínimo, pode apenas constituir uma fachada (*facade*) para os objectos reais. Neste caso, tudo o que é feito é fornecer uma API orientada a casos de uso típicos. No outro extremo, muita da lógica de negócio é colocada na *Service Layer* sob a forma de *Transaction Scripts*. A camada inferior poderá ser um *Domain Model* com interacção directa com a base de dados. Entre estes extremos podem ser concebidas imensas situações intermédias.

3. Padrões de Mapeamento para BDs relacionais

O objectivo da camada de acesso a dados é assegurar a comunicação com os vários constituintes da infra-estrutura necessária. Uma grande parte dessa comunicação é com uma base de dados, geralmente relacional, devido à popularidade que o SQL adquiriu e do seu suporte por diversos fabricantes.

Os padrões são divididos em três grandes categorias: padrões arquitecturais, padrões comportamentais objecto-relacionais e padrões de mapeamente de estruturas (relações e hierarquias). Cada categoria propõe-se a resolver um problema diferente de várias formas.

Bases de Dados OO

Os padrões apresentados neste capítulo tentam sistematizar várias formas de fazer o mapeamento entre objectos e bases de dados relacionais. Apesar da grande proliferação das bases de dados relacionais, as bases de dados orientadas a objectos surgiram com a motivação de reduzir a "impedância" entre objectos e relações. O interesse foi o de trazer o paradigma de orientação a objectos para o armazenamento em disco, tornando os objectos persistentes. O programador ficaria assim liberto dos mapeamentos entre objectos e estruturas de disco. As vantagens imediatas das bases de dados orientadas a objectos são o aumento de produtividade, no entanto, a maioria dos projectos não as utiliza. A principal razão para tal é o facto de as bases de dados relacionais terem ganho muito mais terreno e serem uma tecnologia com provas dadas e suportadas por uma grande variedade de fabricantes de software. A linguagem SQL constitui uma interface padrão para muitas ferramentas.

Ferramentas de mapeamento O/R

Para estruturas com lógicas de negócio organizadas com *Domain Model*, será recomendável a aquisição de uma ferramenta de mapeamento O/R (objectos-relacional). Alguns padrões de mapeamento descritos neste capítulo como o *Data Mapper* não são de implementação fácil e as ferramentas, apesar de não serem baratas, são bastante sofisticadas e compensam o esforço de um desenvolvimento a partir do zero. O conhecimento dos padrões da camada de acesso a dados descritos neste capítulo continua, no entanto, a ser útil. As boas ferramentas oferecem várias opções de mapeamento que só são inteiramente compreensíveis após um estudo dos padrões.

3.1. Padrões arquitecturais

Os padrões arquitecturais descrevem a forma como a camada da lógica de negócio comunica com a base de dados. Cada padrão representa uma escolha que uma vez tomada será difícil de reestruturar para outra. Por outro lado, a própria camada de lógica de negócio pode condicionar o padrão a adoptar nesta camada.

Existem várias técnicas para embeber o SQL na linguagem de programação, no entanto, nenhuma é relativamente simples. Por um lado não se pretende que os progrmadores tenham acesso directo à base de dados porque muitas vezes não dominam a linguagem SQL de forma eficiente. Por outro lado, é desejável que os Administradores de Bases de Dados (DBA) continuem a poder ter acesso aos comandos SQL executados, para os auxiliar no seu trabalho de *tuning* e definição de índices.

3.1.1. Data Gateways

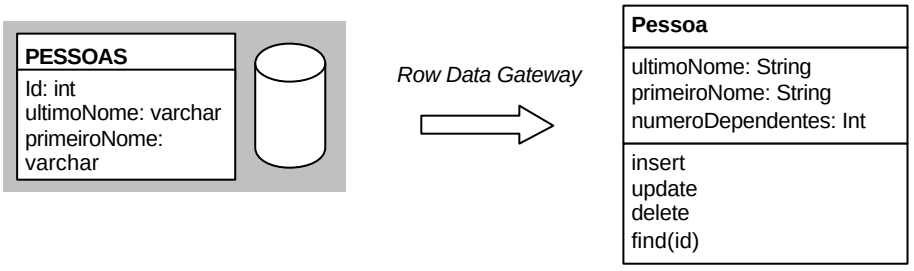
É benéfico separar os acessos SQL da lógica de negócio e colocá-los em classes distintas de forma a que todos os comandos SQL sejam encapsulados em métodos. O acesso à base de dados pelas classes da lógica de negócio terá que ser sempre efectuado por intermédio deste métodos, nunca podendo utilizar comandos SQL directamente.

Uma forma de se organizar estas classes é baseando-as na estrutura da tabela que manipulam, de tal forma que exista uma classe para cada tabela. Cada classe forma uma *Gateway* para a respectiva tabela. Pode definir-se as *Data Gateways* como classes com métodos que facultam acesso a registos da base de dados. O resto da aplicação não precisa de invocar comandos SQL e todo o SQL é fácil de encontrar, facilitando a tarefa dos DBAs.

Uma *Gateway* pode ser utilizada de duas formas. A primeira, designada *Row Data Gateway,* consiste em ter uma instância para cada registo retornado por uma interrogação. A segunda, designada por Table Data Gateway, consiste em ter uma instância para cada tabela da base de dados.

Row Data Gateway

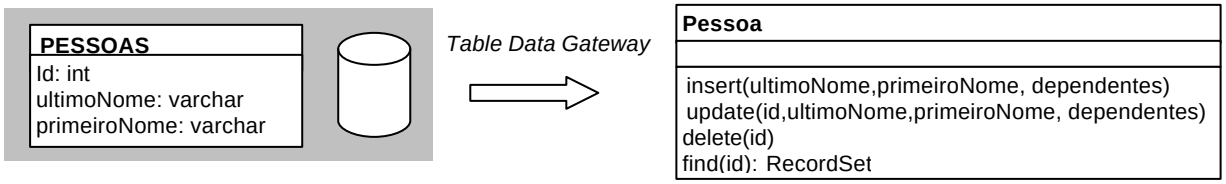
Cada objecto corresponde a um registo de uma tabela mapeada pela classe. Os atributos da classe correspondem às colunas da tabela. Neste exemplo, as colunas da tabela Pessoas são replicadas na classe Pessoa. Esta classe fornece à cama de lógica de negócio métodos para efectuar as operações básicas em registos (insert, update e delete). O método *find* é um método de classe que permite pesquisar registos.



Este padrão é utilizado em conjunto com o padrão de lógica de negócio *Transaction Script*.

Table Data Gateway

Cada objecto corresponde a uma tabela na base de dados. Em vez de usar atributos, a classe cria um *Record Set* que será devolvido à classe da lógica de negócio que os solicitou os registos. O *Record Set* é uma estrutura de dados genérica que emula a estrutura em tabelas de uma base de dados. Os métodos de insert, update e delete necessitam que se especifique os valores dos respectivos campos, já que neste padrão não estão contidos nos atributos da classe.



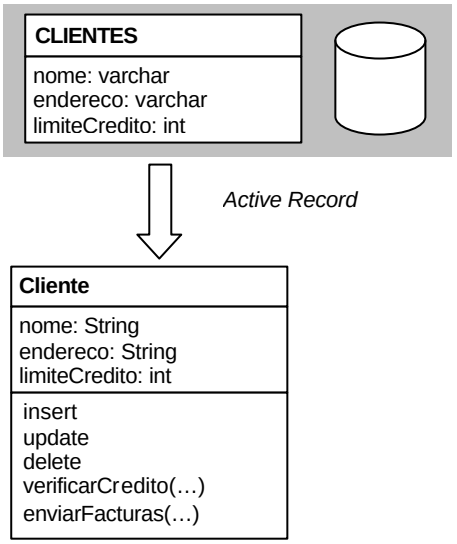
Se a camada da lógica de negócio estiver a ser assegurada por uma padrão *Table Module*, então a utilização de um *Table Data Gateway* será a escolha mais lógica, devido ao seu encaixe com os *Record Sets*. O *Table Data Gateway* também é

um padrão favorável para uma organização em procedimentos na base de dados (*stored procedures*). Neste caso, pode definir-se uma colecção de procedimentos como um *Table Data Gateway* para uma tabela, constituindo um pacote de procedimentos (*package*).

3.1.2. Active Record

O padrão *Active Record* é uma extensão do padrão *Row Data Gateway, em que se junta* à gateway a lógica de negócio necessária para cálculos e validações.

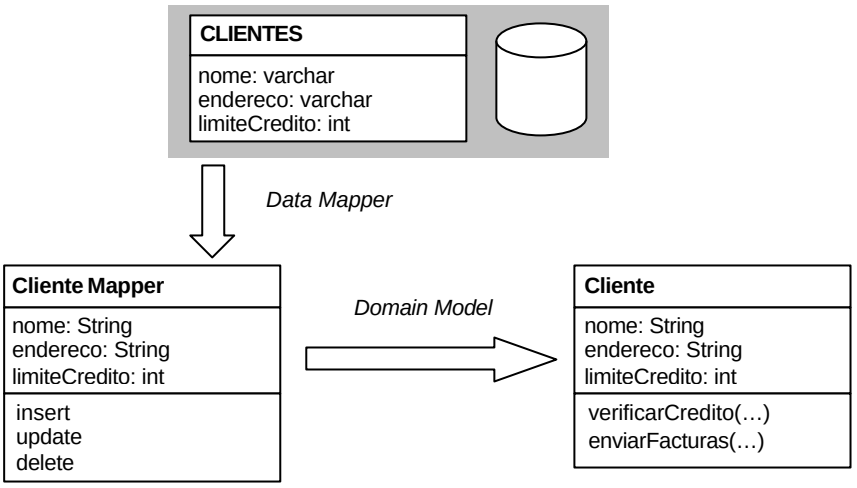
Este padrão pode ser utilizado nos casos em que, para estruturar uma lógica de negócio não muito complexa, *é usado* o padrão *Domain Model*. Em situações muito simples, esta abordagem será preferível, no entanto, a distinção entre camadas fará mais sentido à medida que a lógica de negócio aumente em complexidade.



3.1.3. Data Mapper

A abordagem no outro extremo consiste em isolar completamente o *Domain Model* da base de dados através de um *Data Mapper* que efectua todo o carregamento e armazenamento necessário. É a estrutura mais complexa, mas é também aquela que permite maior variação entre as duas camadas. O *Data Mapper* fornece uma camada de interface que mapeia os objectos do *Domain Model* e as interacções com a base de dados.

Não se recomenda a utilização do *Gateway* como principal mecanismo de persistência de objectos para uma estrutura de *Domain Model*. Se a lógica for simples e existe uma correspondência evidente entre classes e tabelas, deve utilizar-se o *Active Record*. Se a lógica for complexa, será mais recomendável a utilização do *Data Mapper*.



Note-se que a classe *Domain Model* não tem métodos de acesso à base de dados porque todo o mecanismo de persistência é assegurado pelo *Data Mapper* tornando-o transparente para a lógica de negócio

3.2. Padrões comportamentais

O problema comportamental consiste na forma como os vários objectos são trazidos da base de dados e nela guardados novamente, ou seja, o problema a resolver é a definição do momento em que os objectos são lidos ou escritos na base de dados

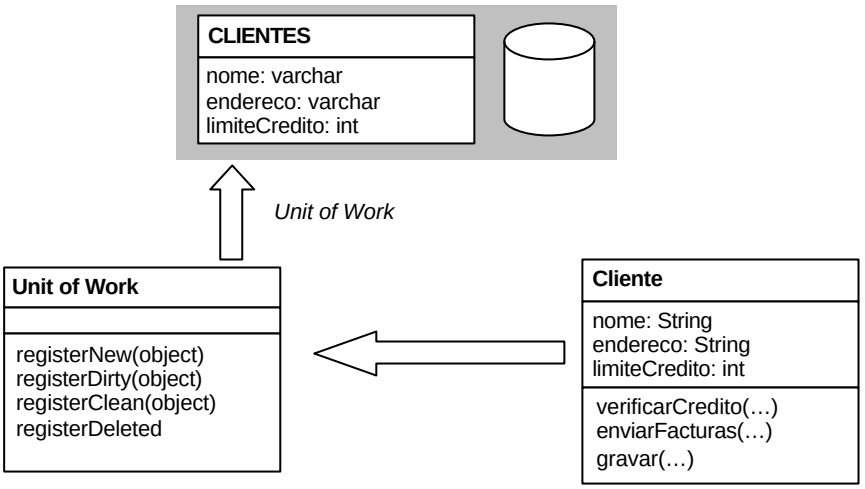
Numa primeira análise o problema parece ser simples, existindo até padrões como o *Active Record* inclui métodos de carregamento e gravação para o efeito. O problema complica-se quando vários objectos são trazidos para a memória e modificados. É necessário controlar quais os objectos que foram modificados e garantir que são guardados dessa forma na base de dados. Sendo poucos os objectos o problema é simples, no entanto, à medida que o número de objectos em memória aumenta o problema aumenta de complexidade. Os principais problemas levantados são os seguintes:

- Alterações na base de dados aquando de uma alteração num objecto levam a muitas chamadas à base de dados tornando o sistema lento;
- Se não houver cuidado, é possível carregar o mesmo registo da base de dados em dois ou mais objectos diferentes
- Se se actualizar mais do que um desses objectos, o estado do registo correspondente poderá ficar inconsistente
- A criação de novos objectos e a modificação de outros pode ser complexa quando é necessário manter as chaves dos primeiros para os referenciar nos segundos.

3.2.1. Unit of Work

À medida que se vão carregando e modificando objectos é necessário garantir a consistência da base de dados. O carregamento de alguns objectos para leitura tem que ser isolado de tal modo que outro processo não o possa alterar em simultâneo, sob pena de promover inconsistência. Este tipo de problemas levanta a necessidade de mecanismos de gestão de concorrência. Um padrão que se destina a resolver estes problemas é o *Unit of Work*, que controla os objectos lidos da base de dados e os que foram de alguma forma alterados. Encarrega-se também de gerir a forma como as actualizações são efectuadas na base de dados. Em vez de ser a aplicação a ter métodos explícitos de gravação, o programador indica a

unidade (*unit of work*) a ser gravada. É delegada para a *Unit of Work* todo o esforço de mapeamento, sequenciação e gravação do objecto em causa.



Uma *Unit of Work* anota todas as alterações efectuadas aos dados dos objectos que intervém numa transacção. Quando a transacção termina, todas as alterações são cometidas de uma só vez na base de dados de forma consistente. A aplicação, em vez de chamar métodos explícitos de gravação, indica a *Unit of Work* a ser gravada.

3.2.2. Identity Map

Com o carregamento de vários objectos, pode ocorrer o carregamento do mesmo registo em dois ou mais objectos diferentes. Se se actualizar mais do que um desses objectos, o estado do registo correspondente poderá ficar inconsistente.

Para lidar com estes problemas, mantém-se um *Identity Map* para cada tabela de todos os registos já carregados em objectos. Sempre que se invocar um registo em particular, consulta-se o *Identity Map* para verificar se este já está carregado num objecto, podendo daí advir duas situações:

- Se já estiver, retorna-se o objecto
- Se não estiver, carrega-se da base de dados e inclui-se no mapa

A chave do objecto no mapa coincide com a chave do registo na base de dados.

3.2.3. Lazy Load

É cómodo para o programador que quando é efectuado o carregamento de um objecto sejam também carregados todos os que lhe estão ligados. O perigo deste mecanismo é que pode ser carregado um grande número de objectos desnecessários, levando a problemas de memória e eventual degradação geral do desempenho do sistema.

Com a utilização do *Domain Model*, os objectos relacionados são todos carregados em simultâneo. Ao carregar-se um objecto que representa uma encomenda, é carregado o objecto que corresponde ao cliente associado à encomenda. Contudo, quando os objectos interligados formam uma rede muito grande, uma simples leitura de objecto pode implicar um grande volume de dados que necessitam de ser carregados da base de dados. Muitas vezes tal não será necessário, uma vez que se pretende apenas consultar um subconjunto muito mais pequeno desses dados.

Para evitar este tipo de acessos, é necessário um mecanismo que, por um lado, reduza o que é trazido para a memória, mas por outro, conheça os objectos

todos do conjunto a ser trazido. O padrão *Lazy Load* interrompe o processo de carregamento mas deixa uma marca na estrutura de objectos de forma que os dados sejam carregados apenas quando forem necessários.

O padrão pode ser implementado de várias formas, entre as quais se destacam as seguintes:

- *Lazy initialization* – Verifica se um campo é nulo antes de o carregar através de um método;
- *Virtual proxy* – Um objecto que imita o real, só é carregado quando é invocado um dos seus métodos;
- *Ghost* – É o objecto num estado parcial só com o ID. Quando se acede a um campo os dados restantes são carregados de imediato.

A essência do padrão *Lazy Load* é manter um representante de cada objecto com uma referência para o mesmo. Só quando se pretende realmente um acesso ao objecto em causa é que este é trazido para a memória, no entanto, toda a "rede" de referências já está carregada.

3.3. Padrões Estruturais

É frequente, mesmo em modelos simples, que as classes se relacionem entre si formando estruturas. O objectivo dos padrões estruturais é descrever a forma como as estruturas formadas por vários objectos são mapeadas em tabelas na base de dados. Os padrões reflectem as estruturas analisadas, nomeadamente, mapeamento de relações e mapeamento de hierarquias.

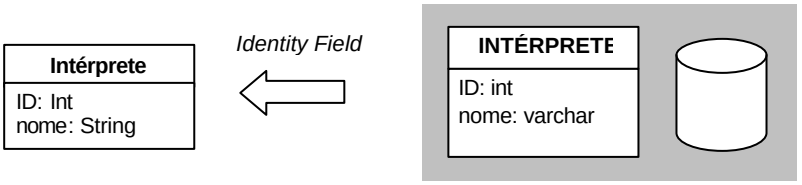
3.3.1. Mapeamento de relações

Neste ponto tentamos descrever as diferentes formas de representação das ligações entre entidades numa base de dados e entre objectos num modelo de domínio. Relativamente a isto existem dois problemas base:

- A diferença de representação: os objectos tratam as ligações como referências (que são mantidas através de endereços de memória) enquanto que as bases de dados relacionais tratam as ligações através de chaves para outras tabelas.
- As colecções: os objectos podem ter colecções para lidar com várias referências num único campo. Por exemplo, um objecto 'encomenda' tem uma colecção de objectos 'linha de encomenda' que não têm qualquer referência para esta. Para se obter esta estrutura em bases de dados relacionais é necessário fazer o contrário, ou seja, são as linhas de encomenda que referenciam a encomenda, através de uma chave estrangeira.

Identity Field

A forma de lidar com este problema de é ter um campo unicamente para representar a identidade de um objecto, um *Identity Field*. As bases de dados relacionais distinguem os registos através da chave primária, no entanto, os objectos em memória não precisam dessa chave, já que o próprio sistema trata da sua identidade.

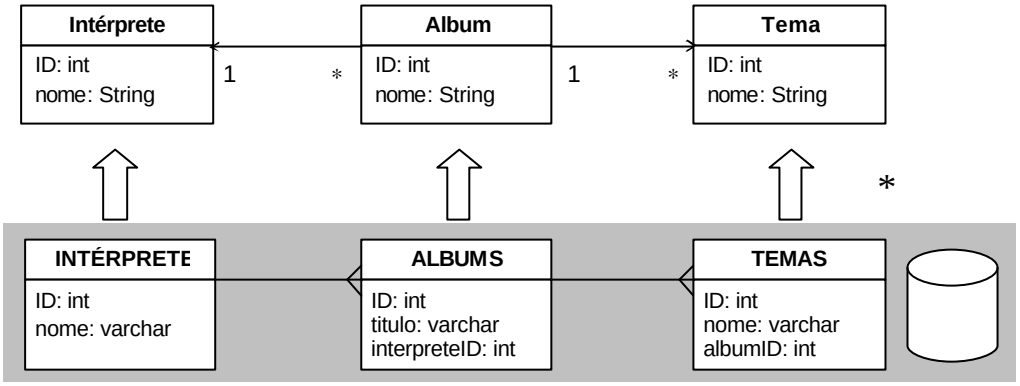


Para criar um novo objecto é necessária uma chave. Esta pode ser obtida de três formas: pedir à base de dados que gere uma chave ou gerar uma que garante

unicidade. Deve utiliza-se o *Identity Field* quando é necessário fazer mapeamento entre objectos em memória e registos na base de dados. Ocorre quando a lógica de negócio é estruturada com *Domain Model* e ligada à base de dados com *Row Data Gateway*. Não é necessário este mapeamento se for utilizado *Transaction Script*, *Table Module* ou *Table Data Gateway*.

Foreign Key Mapping

Este campo é utilizado para mapear em qualquer sentido as referências ao objecto e as chaves relacionais. Quando são lidos objectos de disco, utiliza-se um *Identity Map* como uma tabela de mapeamento de chaves relacionais para objectos. Cada vez que se encontra uma chave estrangeira numa tabela, utiliza-se o *Foreign Key Mapping* para se ligar a referência entre objectos. Se não existir a chave no *Identity Map*, será necessário carregar o objecto da base de dados ou utilizar o padrão *Lazy Load* para o fazer. Qualquer referência entre objectos é substituída pelo identificador do objecto referenciado.



Utilização de *Foreign Key Mapping* para mapear uma referência única (associação muitos-para-um)

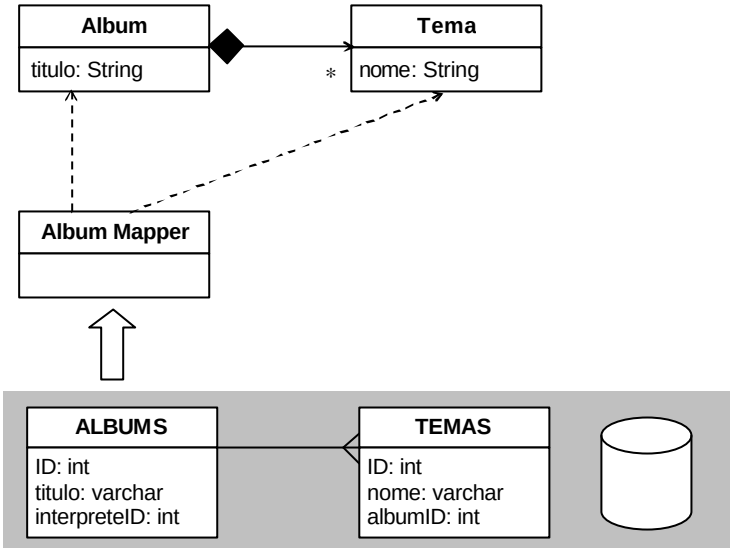
Se um objecto tem uma colecção de referências, é necessária uma outra interrogação para encontrar todas as linhas que ligam ao ID do objecto em causa. Cada um desses objectos tem que ser criado e adicionado à colecção. Armazenar a colecção implica armazenar cada objecto e garantir que tem a referência para o objecto referenciado. Este processo pode ser complexo principalmente quando tem que lidar com novos objectos inseridos na colecção e outros removidos.

Este padrão é adequado para mapear associações 1 para muitos, no entanto, não pode ser utilizado para associações muitos para muitos.

Dependent Mapping

O padrão *Dependent Mapping* consiste numa simplificação do *Foreign Key Mapping* para os casos em que há dependência entre classes. Nestes casos define-se uma classe como *proprietário* e a(s) outra(s) *dependente(s)*. O objectivo é ser a classe *proprietário* a responsável pela sua persistência e pelas suas classes *dependentes*. Não faz sentido que as classes *dependentes* utilizem o padrão Identity Field. A coerência com a base de dados é facilmente conseguida apagando sempre os registos de todos os dependentes e inserindo novos com as alterações efectuadas. As chaves dos registos correspondentes a classes dependentes são obtidas da base de dados ou geradas pela classe *proprietário*.

Tal como no padrão *Foreign Key Mapping*, este padrão é adequado para mapear associações 1 para muitos, mas não pode ser utilizado para associações muitos para muitos.

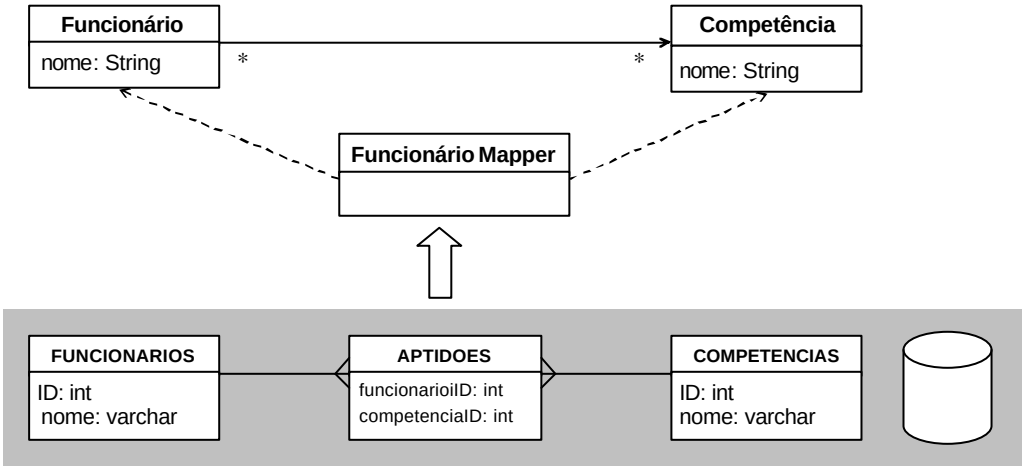


Utilização de *Dependent Mapping* para mapear uma referência única (associação muitos-para-um)

Associate Table Mapping

Um caso distinto surge nas relações muitos-para-muitos em que existem colecções nos dois extremos. O padrão *Associate Table Mapping* representa uma ligação entre objectos como uma tabela de associação com chaves estrangeiras entre as tabelas ligadas. Pode representar qualquer associação mas é mais apropriada para as muitos-muitos.

Um exemplo consiste numa pessoa poder ter várias competências e cada competência saber quais são as pessoas que a possuem. As bases de dados relacionais não lidam com estes casos directamente e por isso usam *Associate Table Mapping* para criar uma nova tabela relacional para lidar com associação muitos-para-muitos.

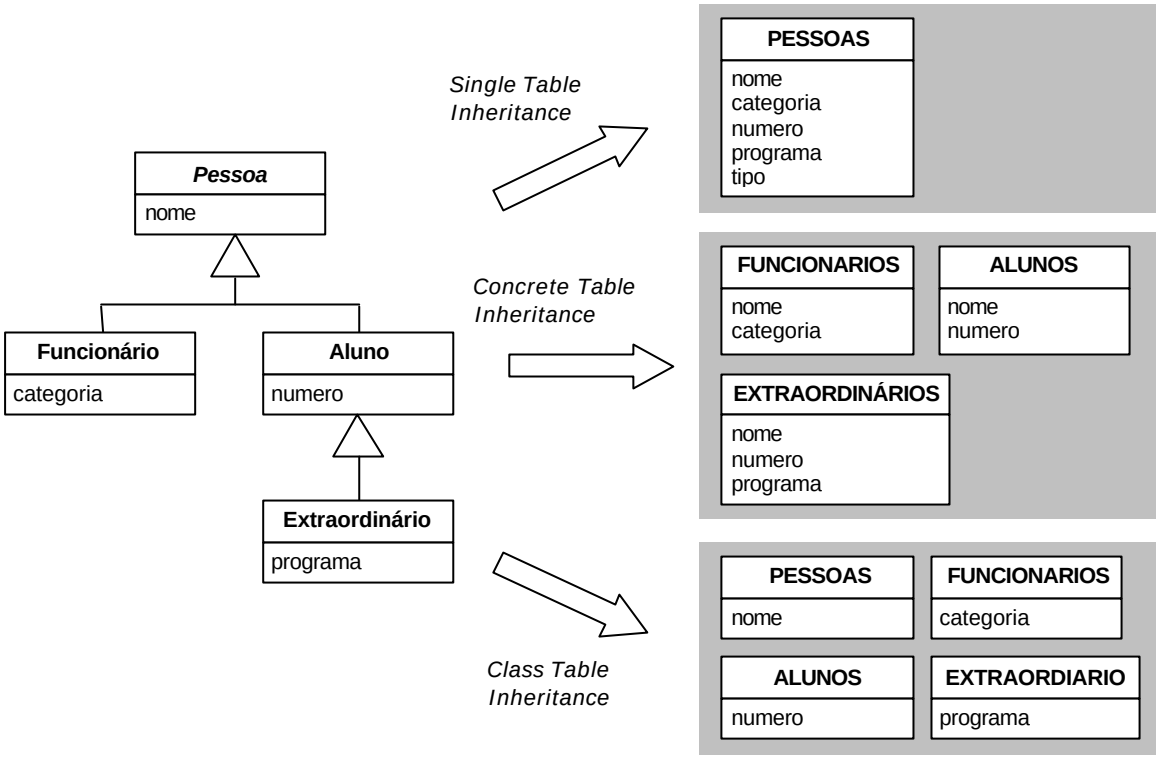


Utilização de *Association Table Mapping* (associação muitos-para-muitos)

3.3.2. Herança

Os sistemas relacionais não lidam com mecanismos de herança. Existem três formas de mapear a informação que está organizada em classes e subclasses na representação de hierarquias de em objectos:

- *Single Table Inheritance* – Uma tabela para todas as classes da hierarquia
- *Concrete Table Inheritance* – Uma tabela para cada classe concreta da hierarquia (classes finais)
- *Class Table Inheritance* – Uma tabela para cada classe da hierarquia



No exemplo acima, modeliza-se uma hierarquia entre pessoas, funcionários, alunos e alunos-estraordinários, com relações de herança entre si as classes correspondentes. O mapeamento para tabelas pode ser feito de uma das três formas apresentadas.

Os compromissos centram-se na duplicação da estrutura de dados e rapidez de acesso. O padrão *Class Table Inheritance* constitui a relação mais simples entre classes e tabelas, mas necessita de muitas junções para carregar um único objecto o que reduz o desempenho. *Concrete Table Inheritance* evita as junções permitindo carregar um objecto da base de dados de uma única tabela. Ocorrem problemas se houver necessidade de alterar a estrutura da super classe. Tal implica alterar a estrutura de todas as tabelas correspondentes às subclasses. Em algumas bases de dados, o padrão *Single Table Inheritance* tem o inconveniente de desperdiçar espaço já que todos os registos têm que ter colunas para todas as subclasses possíveis levando a colunas vazias. Outro problema deste padrão é o tamanho da tabela, levando a contenção elevada nos seus acessos. A grande vantagem é que todos os dados vivem na mesma tabela o que facilita as alterações e evita totalmente as junções.

As três opções não são mutuamente exclusivas, sendo possível que numa hierarquia ocorra mistura de padrões. Um caso possível é ter várias classes mapeadas com *Single Table Inheritance* e utilizar *Class Table Inheritance* para alguns casos mais invulgares. A mistura de padrões trás, no entanto, um aumento da complexidade.

Tendo as três soluções vantagens e inconvenientes, não há nenhuma que se possa afirmar como a mais adequada. As circunstâncias e preferências são muitas vezes os factores de desempate. Uma primeira escolha poderá ser o *Single Table Inheritance* já que é o mais simples e mais permeável a alterações. As outras duas escolhas poderão ser utilizadas quando o desperdício de espaço possa ser prejudicial. A melhor abordagem muitas vezes vem da consulta do DBA que faz uma análise sob uma perspectiva do impacto da escolha do desempenho da base de dados.

4. Padrões da Camada de Apresentação

4.1. Model-View-Controller

Existem duas formas básicas de estruturar um programa num servidor web: como um *script* ou como uma *server page*.

Um script é um programa com funções ou métodos que lidam com pedidos http. O script recebe os dados da página Web através do pedido HTTP. O output do servidor Web é uma string, a resposta, que o script pode escrever usando as operações normais de escrita para um stream. Exemplos de scripts são os CGI scripts e as servlets Java.

Nas server pages, o desconforto de escrever HTML com comandos de escrita para um stream é ultrapassado através da estruturação dos programas em volta do texto da resposta. O programador escreve a resposta em HTML e insere, em certos pontos no HTML, scriptlets de código a executar. São exemplos desta abordagem as linguagens PHP, ASP e JSP.

A abordagem das server pages funciona melhor quando há pouco processamento da resposta. Os programas ficam bastante complicados quando é necessário fazer opções baseados no input. Os scripts funcionam melhor para interpretação dos pedidos e as server pages para formatação da resposta. O padrão MVC consiste na combinação desta ideia com a noção de que a lógica de negócio deve ser separada da lógica de visualização.

MVC é um dos padrões mais referenciados (e dos mais incompreendidos). Começou como uma framework desenvolvida por Trygve Reenskaug para Smalltalk no final da década de 1970's. Desde então tem tido um importante papel no desenho de interfaces e na maioria das frameworks de interfaces com o utilizador.

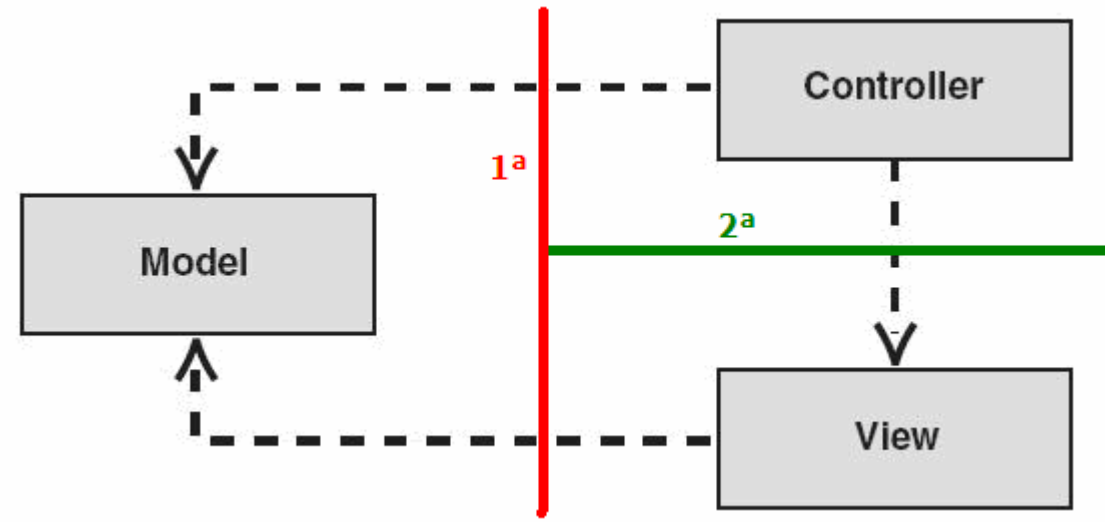
Componentes

O padrão MVC divide um sistema em três componentes distintos:

O modelo é um objecto que representa a informação sobre o domínio. É um objecto não visual que comporta todos os dados e comportamentos (para além dos que são usados na interface com o utilizador). O modelo é, em princípio, estruturado com um dos padrões da camada da lógica de negócio apresentados no ponto 2. Na sua forma mais pura de orientação por objectos o modelo é um Domain Model mas pode ser estruturado com qualquer um dos padrões da camada da lógica de negócio.

A vista representa a visualização do modelo na interface com o utilizador. Consiste na visualização de informação, aquilo que o utilizador realmente vê.

O controlador usando o input do utilizador, manipula o modelo e selecciona a vista a utilizar. A interface com o utilizador é portanto a combinação do controlador com a vista.



A figura anterior esquematiza as ligações entre os componentes do MCV. Este padrão pode ser visto como duas divisões fundamentais: a divisão entre apresentação e modelo (na figura a vermelho) e, na apresentação, a divisão entre o controlador e a vista (na figura a verde).

A separação entre apresentação e o modelo é uma das heurísticas mais importantes do bom desenho de software. Esta divisão é importante porque:

- Fundamentalmente, apresentação e modelo envolvem preocupações totalmente diferentes: no desenvolvimento de uma vista pensamos nos mecanismos de interface e como estruturar uma boa interface. No desenvolvimento de um modelo pensamos nas políticas do negócio, interacções com a base de dados, etc. Diferentes bibliotecas são usadas nos dois casos.
- Dependendo do contexto, os utilizadores querem ver a mesma informação básica do modelo de formas diferentes. Separar a apresentação e o modelo permite construir várias vistas sobre o mesmo modelo, por exemplo, o mesmo modelo poderia ser visto num rich client, num browser Web, numa API remota, numa interface de linha de comando, etc.
- Objectos não visuais são, normalmente, mais fáceis de testar do que objectos visuais. Separar a apresentação do modelo permite testar toda a lógica de negócio facilmente sem ter de recorrer a, por exemplo, ferramentas de teste de interfaces que são usualmente difíceis.

A segunda divisão, a separação entre a vista e controlador é menos importante. No entanto, colocar o controlador em objectos separados torna mais fáceis os testes da aplicação.

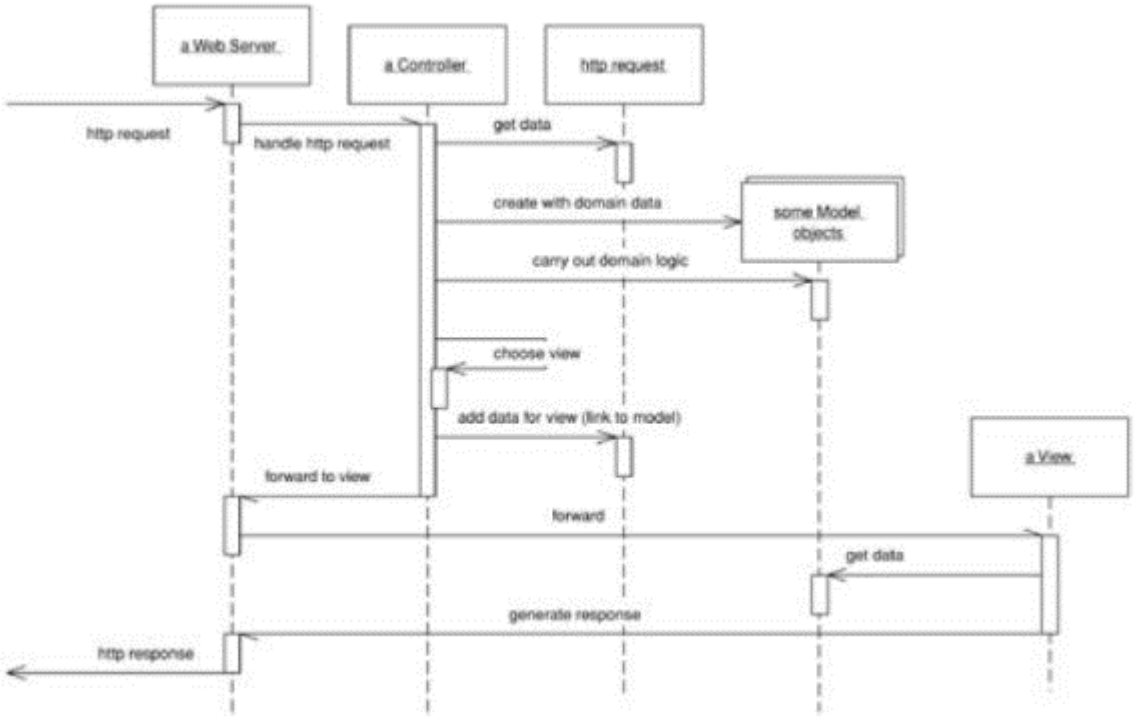
Dependências

Um dos pontos-chave das separações é a direcção das dependências. Como se pode ver na figura do padrão, a apresentação depende do modelo mas o modelo não depende da apresentação. Os programadores do modelo não necessitam conhecer a apresentação. Por outro lado, esta separação torna mais fácil a alteração da camada da apresentação e também a adição de novas apresentações. Por outro lado, as vistas são independentes do controlador, isto é, uma vista pode ser referida em diferentes contextos de navegação/utilização.

How It Works

O funcionamento do modelo, vista e controlador de input num servidor web é, basicamente o seguinte: o controlador de input lida com o pedido http, usa o modelo para tratar da lógica de negócio e depois usa a vista para criar uma resposta baseada num modelo.

A seguinte figura mostra um exemplo de funcionamento do padrão MVC:



1. Chega um pedido http do cliente ao controlador que processa a informação do pedido.
2. O controlador determina qual a acção especificada pelo pedido http e decide qual o objecto do modelo a usar para processar a lógica do negócio; o controlador cria uma instância desse objecto e activa-o;
3. O objecto do modelo interage com a fonte de dados, faz o indicado pelo pedido e reúne informação para a construção da resposta;
4. O objecto de modelo devolve o controlo ao controlador que, analisando os resultados, decide que vista deve ser usada para processar a resposta;
5. O controlador passa o controlo para a vista, juntamente com os dados da reposta (na forma de um objecto de sessão http ou um simples objecto de apoio partilhado pelo controlador e pela vista) que mostra a resposta com os dados do modelo.

When to Use It

O padrão MVC baseia-se em duas separações:

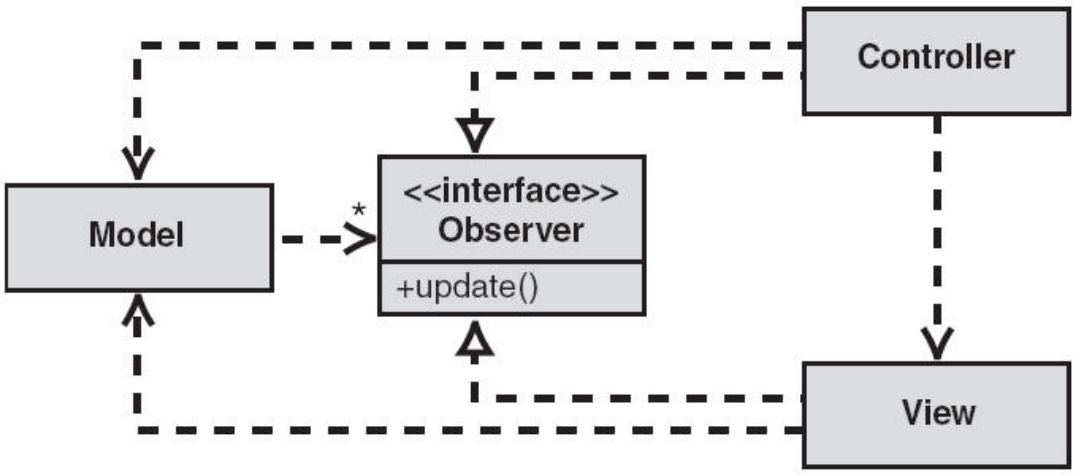
A separação entre a apresentação e o modelo, excepto em casos de extrema simplicidade, deve ser sempre feita pois é uma das pedras basilares do bom desenho de software.

A separação entre a vista e o controlador é menos importante e só deve ser usada quando é realmente útil. Num rich-client, esta separação é raramente útil. Mas nos front ends web pode ser uma solução bastante útil.

Variação: MVC Activo

O MVC denominado MVC Passivo é a versão do MVC utilizada em aplicações empresariais, em que o controlador age sobre o modelo e selecciona a vista a ser utilizada. Neste cenário o modelo é completamente independente da vista e do controlador. O protocolo http é um exemplo desta versão do padrão, pois o browser não reflecte as alterações assíncronas da base de dados. O browser mostra a apresentação (vista) e responde aos inputs do rato e do teclado (controlador), mas não reflecte as alterações efectuadas no servidor a não ser que o utilizador o "provoque" essa apresentação.

A seguinte figura mostra o esquema da variação deste padrão, denominada MVC Activo:



Esta variação é essencialmente utilizada em rich clients quando o modelo pode ser alterado sem que o controlador tenha tido envolvimento nesse processo. Isto pode acontecer quando outros programas estão a alterar os dados (Bases de Dados) e estas alterações têm que ser reflectidas na interface.

Nesta variação é usado o padrão Observer para implementar as notificações às vistas das alterações do modelo, isto é, quando o modelo é alterado as vistas são notificadas e podem reagir automaticamente a essas alterações em intrevenção do controlador.

Implementação em .NET

Este exemplo implementa uma página Web com uma simples combobox e uma datagrid para apresentar os dados da selecção da combobox.

Recordings

Select a Recording:

Album Title Submit

id	title	time
1	Track1	6:51
2	Track2	7:33
3	Track3	6:38
4	Track4	7:53
5	Track5	7:24
6	Track6	7:16
7	Track7	6:29
8	Track8	6:02
9	Track9	7:36
10	Track10	3:01

A possibilidade de separar, no .NET, o HTML do código associado aos eventos (code-behind), torna fácil a separação entre a apresentação dos dados (View) do Model e Controller. Cada página asp.net tem um mecanismo que permite que os métodos chamados possam ser implementados numa classe separada.

A construção deste exemplo está descrita a seguir nos seguintes módulos View, Controller, Model.

View

```
<%@ Page language="c#" Codebehind="Solution.aspx.cs"
    AutoEventWireup="false" Inherits="Solution" %>
<html>
    <head>
        <title>Solution</title>
    </head>
    <body>
        <form id="Solution" method="post" runat="server">
            <h3>Recordings</h3>
            Select a Recording:<br/>
            <asp:dropdownlist id="recordingSelect" runat="server" />
            <asp:button id="submit" runat="server" text="Submit"
                enableviewstate="False" />
            <p/>
            <asp:datagrid id="MyDataGrid" runat="server" width="700"
                  backcolor="#ccccff" bordercolor="black" showfooter="false"
                  cellpadding="3" cellspacing="0" font-name="Verdana" font-size="8pt"
                  headerstyle-backcolor="#aaaadd" enableviewstate="false" />
        </form>
    </body>
</html>
```

Como se pode verificar, o código corresponde só à apresentação dos dados e não faz nenhuma referência aos dados da Base de Dados. Assim qualquer alteração à apresentação deve ser feita neste ficheiro. A primeira linha do código corresponde à especificação da classe que irá implementar os métodos desta página.

Controller

A segunda parte corresponde ao code-behind da apresentação dos dados. São necessárias algumas alterações sintácticas para ligar os dois ficheiros. Os eventos das acções efectuadas na apresentação são controlados pelo método **Initialize Component** onde se definem métodos como o *click* e *Load*. No Controller os dados retornados do Model (DataSet) são adaptados aos eventos de apresentação (View). O Code-behind é um mecanismo elegante para separar a regras de apresentação das regras do modelo e de controle.

```
public class Solution : System.Web.UI.Page
{
    protected System.Web.UI.WebControls.Button submit;
    protected System.Web.UI.WebControls.DataGrid MyDataGrid;
    protected System.Web.UI.WebControls.DropDownList recordingSelect;

    private void Page_Load(object sender, System.EventArgs e)
    {
        DataSet ds = DatabaseGateway.GetRecordings();
        recordingSelect.DataSource = ds;
        recordingSelect.DataTextField = "title";
        recordingSelect.DataValueField = "id";
        recordingSelect.DataBind();
    }

    void SubmitBtn_Click(Object sender, EventArgs e)
    {
       DataSet ds =
          DatabaseGateway.GetTracks(
          (string)recordingSelect.SelectedItem.Value);

       MyDataGrid.DataSource = ds;
       MyDataGrid.DataBind();
    }

    private void InitializeComponent()
    {
        this.submit.Click += new System.EventHandler(this.SubmitBtn_Click);
        this.Load += new System.EventHandler(this.Page_Load);
    }
}
```

Model

O exemplo seguinte mostra o Model e como se pode verificar não depende do da View nem do Controller. A sua função é reflectir os dados da Base de Dados.

```
public class DatabaseGateway
{
   public static DataSet GetRecordings()
   {
      String selectCmd = "select * from Recording";

      SqlConnection myConnection =
         new SqlConnection(
            "server=(local);database=recordings;Trusted_Connection=yes");
      SqlDataAdapter myCommand = new SqlDataAdapter(selectCmd, myConnection);

      DataSet ds = new DataSet();
      myCommand.Fill(ds, "Recording");
      return ds;
   }

   public static DataSet GetTracks(string recordingId)
   {
      String selectCmd =
         String.Format(
         "select * from Track where recordingId = {0} order by id",
         recordingId);

      SqlConnection myConnection =
         new SqlConnection(
            "server=(local);database=recordings;Trusted_Connection=yes");
      SqlDataAdapter myCommand = new SqlDataAdapter(selectCmd, myConnection);

      DataSet ds = new DataSet();
      myCommand.Fill(ds, "Track");
      return ds;
   }
}
```

4.2. Padrões para as Vistas

Os padrões de visualização definem padrões para a vista do padrão MVC. Quanto a este componente do MVC existem duas decisões básicas a tomar: usar o padrão Transform View ou o padrão Template View e, por outro lado, usar as vistas a um passo ou o padrão Two Step View (uma variação aplicável tanto ao padrão Transform View como ao padrão Template View).

4.2.1. Template View

O padrão Template View permite escrever a apresentação sob a forma de páginas e marcas embebidas nas páginas que indicam aonde deve estar o conteúdo dinâmico.

Escrever um programa que gera HTML numa linguagem como o Java ou o C# pode ser bastante complicado dada a dificuldade de criar e concatenar strings. Se o HTML a gerar for simples, não é problemático, se falarmos de uma página HTML completa, a complicação pode ser inaceitável.

Para páginas HTML estáticas os editores WYSIWYG são bastante úteis e resolvem a maioria dos problemas com facilidade, mas para páginas dinâmicas (que normalmente incorporam dados dinâmicos vindos, por exemplo, de uma base de dados) os resultados são sempre diferentes a cada pedido http. A melhor

solução para este problema é compor páginas Web dinâmicas, Server Pages, como se escreve páginas estáticas e incluir nestas marcas que façam chamadas a procedimentos para reunir a informação dinâmica. Como a parte estática da página actua como uma template para cada resposta, este padrão chama-se Template View.

How It Works

A ideia básica do Template View é incluir marcas no HTML estático. Quando a página é usada para responder a um pedido, as marcas são substituídas pelos resultados de alguma computação, como por exemplo, uma query a uma Base de Dados. Desta forma, a página pode ser escrita normalmente, até com recurso a editores HTML WYSIWYG por não programadores. As marcas referenciam programas que produzem os resultados.

Muitas plataformas implementam este padrão sob a forma de linguagens como ASP, JSP e PHP. Estas linguagens, além de permitirem embeber marcas específicas no HTML também permitem utilizar uma linguagem de programação na própria página HTML (scriptlets).

Marcas específicas embebidas

Existem várias formas de incluir as marcas no HTML. A forma mais usada é a de incluir marcas tipo HTML no HTML estático, outra é fazer isto recorrendo a marcas de texto especiais. Esta segunda opção, apesar de não funcionar tão bem com editores de HTML WYSIWYG, oferece uma legibilidade bastante superior pois torna fácil a distinção entre as marcas HTML normais e as marcas especiais.

Muitas plataformas oferecem um conjunto de marcas para uso mas cada vez mais plataformas permitem definir marcas específicas.

Código embebido (scriptlets)

Uma das formas mais populares do padrão Template View são as Server Pages como ASP, JSP ou PHP. De facto, estas server pages vão um pouco adiante da noção base de Template View dado que permitem incluir código arbitrário no HTML estático, estes bocados de código embebido chamam-se scriptlets.

Fowler considera que esta funcionalidade é um grande problema por diferentes motivos:

- A colocação de muito código nas páginas elimina a possibilidade de edição das páginas por não programadores;
- As páginas são modelos pobres para um programa que fazem com que se percam as capacidades de modularização quer em OO ou procedimental;
- A colocação de muito código nas páginas faz com que a estrutura em camadas da aplicação empresarial seja posta em causa muito facilmente. A implementação da lógica de domínio nas server pages faz com que a sua estruturação se perca e a duplicação de funcionalidades ocorra.

Helper Object

Este padrão oferece claramente muito poder e flexibilidade, infelizmente leva a código extremamente complicado que dificulta a manutenção. Sendo assim, para usar tecnologias baseadas em Server Pages, os programadores devem ser muito disciplinados para manter a lógica da programação fora da estrutura da página. Isso consegue-se usando um objecto de apoio (Helper Object) à página. Este objecto de apoio deverá ter toda a lógica de programação. A página só tem chamadas a este objecto, o que simplifica a página e a torna mais numa Template View. A simplicidade resultante permite aos não programadores editarem as páginas e aos programadores concentrarem-se no objecto de apoio.

Dependendo da plataforma em que se está a trabalhar podemos transformar todas as chamadas a este objecto de apoio em marcas especiais. Isto torna a página mais consistente. As marcas mais simples são as que recuperam informação

do sistema e as colocam no sítio correcto na página. Estas são facilmente traduzíveis para chamadas ao objecto de apoio que resultam em texto ou HTML.

Apesar de esta solução parecer um princípio simples e recomendável, existem alguns problemas que se podem tornar complicados de resolver, como o comportamento condicional ou as iterações nas Server Pages.

Condicionais

Uma das problemáticas é o comportamento condicional das páginas. O caso mais simples é quando algo só deve ser mostrado se uma condição é verdadeira. Para este caso deverá haver uma marca do género:

```
<IF condition = "$pricedrop > 0.1"> ...show some stuff </IF>
```

O problema deste tipo de marcas é que, quando o seu uso se prolifera, a página começa a ficar menos um Template View e mais um programa. Por este facto, é desaconselhado o uso deste tipo de marcas genéricas.

A solução mais óbvia é mover a condição para dentro do objecto de apoio. Neste caso, se a condição não for verdadeira a resposta da chamada ao objecto de apoio será uma string vazia e é garantido que a lógica está contida no objecto de apoio. Esta abordagem funciona bem se não houver marcas na resposta.

No entanto, se quisermos, por exemplo fazer o highlight de um campo numa listagem se o registo respeitar uma determinada condição, esta solução não funciona bem. Neste caso, queremos sempre mostrar o campo da listagem mas, em alguns casos, queremos adicionar as marcas que fazem o highlight do campo.

Neste caso, a forma mais simples seria por o objecto de apoio a gerar as marcas. Dessa forma a lógica seria mantida fora da página, com o custo de mover a opção do mecanismo de highlight para fora da página.

Para uma solução óptima deste problema, é necessário o uso de uma marca condicional, mas que vá além da simples marca <IF>.

Neste caso, em vez de uma solução semelhante a:

```
<IF expression = "isHighSelling()"><B></IF>
   <property name = "price"/>
<IF expression = "isHighSelling()"></B></IF>
```

Teríamos algo como:

```
<highlight condition = "isHighSelling" style = "bold">
    <property name = "price"/>
</highlight>
```

É importante manter a condição baseada numa só propriedade boleana do objecto de apoio pois complicar esta condição é colocar a lógica na página.

Outro exemplo desta visualização condicional é a questão da internacionalização. Por exemplo, um texto que só seria mostrado nos EUA ou no Canada, em vez de:

```
<IF expression = "locale = 'US' || 'CA'"> ...special text </IF>
```

Teríamos algo como:

```
<locale includes = "US, CA"> ...special text </locale>
```

Iteração

A iteração dentro de uma colecção representa um problema semelhante. Para mostrar, por exemplo, uma tabela com a lista de itens de uma encomenda é necessária uma marca que permita a construção fácil desta tabela. Para este tipo de problemas normalmente existem marcas da plataforma que permitem fazer este tipo de iteração. Ao contrário da marca genérica <IF>, estas marcas de iteração não introduzem qualquer tipo de problema.

Mais que a Vista

O nome Template View sublinha que a principal função deste padrão é representar a vista no padrão MVC. Para muitos sistemas este padrão deve representar apenas a vista mas para sistemas mais simples é aceitável que seja usado como controlador.

Quando a Template View toma responsabilidades que não a de vista é importante manter essas responsabilidades no objecto de apoio e não na página.

Processamento no servidor Web

Qualquer sistema de templates obriga o servidor Web a um processamento extra. Este processamento pode ser feito através da compilação da página depois de criada, compilação da página quando é feito o pedido ou interpretação da página quando é feito o pedido.

Excepções

Um ponto importante neste padrão é o tratamento de excepções. Se a excepção chega ao contentor web o resultado do processamento pode ser um erro para o utilizador (o tratamento de excepções varia de plataforma para plataforma). Esta é mais uma razão para evitar os scriptlets pois se o código for colocado no objecto de apoio as excepções podem ser todas tratadas.

Template View em Scripts

Apesar de as server pages serem o meio mais comum para implementar o padrão Template View, é possível escrever scripts no estilo Template View. Para tal basta, no script, em vez de usar a concatenação de strings usar funções que geram as marcas para o output. Evitando assim misturar o código da lógica com o output.

Prós e Contras

A grande vantagem do padrão Template View é que permite construir as páginas de uma aplicação olhando para a estrutura da página. É normalmente mais fácil de usar e aprender que o padrão Transform View, especialmente, se a equipa de desenvolvimento estiver dividida em designers (template views) e programadores (objectos de apoio).

As desvantagens deste padrão são:

- Facilita a introdução de lógica complicada nas páginas, aumentando com isso a dificuldade de manutenção, particularmente por não programadores. Torna necessário a existência de uma boa disciplina para evitar este problema.
- É mais difícil de testar que o padrão Transform View pois está ligada a um web server. As Transform Views são muito mais fáceis de testar independentemente do web server.
- Pode ser considerado que o padrão Two Step View é mais fácil de usar o com o padrão Transform View.

Quando se usa o padrão MVC a opção central é entre o padrão Template View ou o Transform View.

Exemplo na plataforma J2EE

De seguida apresentamos um exemplo do padrão Template View na plataforma J2EE com uma página JSP como vista e uma servlet como controlador separado.

Nesta plataforma quando se usa JSPs como vista a invocação deste JSP é quase sempre feita através de uma servlet que age como um controlador do MVC. Este controlador tem de passar à vista alguma informação para que o JSP saiba o que mostrar. Uma forma de fazer esta passagem de informação do controlador para a vista é obrigar o controlador a criar o objecto de apoio da vista e passá-lo à vista através do objecto que representa o pedido http.

Neste exemplo, o controlador (Page Controller) será algo como:

```
class ArtistController...

   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws IOException, ServletException {
      Artist artist = Artist.findNamed(request.getParameter("name"));
      if (artist == null)
         forward("/MissingArtistError.jsp", request, response);
      else {
         request.setAttribute("helper", new ArtistHelper(artist));
         forward("/artist.jsp", request, response);
      }
   }
```

A informação do modelo é passada ao objecto de apoio na sua criação. Adicionalmente, o objecto de apoio vai conter métodos de acesso aos dados do modelo. O código do objecto de apoio é o seguinte:

```
class ArtistHelper...

   private Artist artist;
   public ArtistHelper(Artist artist) {
      this.artist = artist;
   }

   public String getName() {
      return artist.getName();
   }
```

A página JSP terá apenas de usar o objecto de apoio. Este objecto é criado através da marca <useBean> e referido através de uma expressão Java ou de uma propriedade.

```
<jsp:useBean id="helper" type="actionController.ArtistHelper" scope="request"/>

<B> <%=helper.getName()%></B>
<B> <jsp:getProperty name="helper" property="name"/> </B>
```

A escolha entre expressões ou propriedades é uma questão de legibilidade. Os programadores preferirão as expressões que são de mais fácil leitura, por outro lado, os não programadores preferirão as propriedades pois seguem o forma HTML.

Para mostrar um atributo de um objecto do modelo bastou-nos usar estas referências simples. Para, por exemplo, mostrar uma lista de álbuns de um dado artista, será necessário usar um ciclo. Este problema tem várias soluções possíveis:

Fazer o ciclo num scriptlet dentro da server page:

```
<UL>
<%
   for (Iterator it = helper.getAlbums().iterator(); it.hasNext();) {
      Album album = (Album) it.next();%>
   <LI><%=album.getTitle()%></LI>
<%   } %>
</UL>
```

Esta solução faz com que a server page contenha código Java e a torne difícil de ler. Outra alternativa é mover o ciclo para o objecto de apoio:

```
class ArtistHelper...

   public String getAlbumList() {
      StringBuffer result = new StringBuffer();
      result.append("<UL>");
      for (Iterator it = getAlbums().iterator(); it.hasNext();) {
         Album album = (Album) it.next();
         result.append("<LI>");
         result.append(album.getTitle());
         result.append("</LI>");
      }
      result.append("</UL>");
      return result.toString();
   }

   public List getAlbums() {
      return artist.getAlbums();
   }
```

Esta solução é bastante mais fácil pois a quantidade de HTML a produzir é pequena.

Tal como o código Java na Server Page, o código HTML no objecto de apoio deve ser evitado. A melhor solução para o problema da iteração, que permite respeitar esta regra, é uma marca especializada para a iteração.

```
<UL><tag:forEach host = "helper" collection = "albums" id = "each">
   <LI><jsp:getProperty name="each" property="title"/></LI>
</tag:forEach></UL>
```

Exemplo na plataforma .NET

Este próximo exemplo é semelhante ao anterior em Java mas usa a plataforma .NET, mais especificamente a linguagem C# e as páginas ASP. NET.

Em .NET a forma usual de implementar o controlador é usando um objecto que age como controlador e ao mesmo tempo objecto de apoio à vista. Para não colocar o código num scriptlet define-se uma classe de apoio a que se dá o nome de *code behind*. Na vista em ASP é referida da seguinte forma:

```
<%@ Page language="c#" Codebehind="bat.aspx.cs" AutoEventWireup="false"
trace="False"
  Inherits="batsmen.BattingPage" %>
```

A página pode aceder aos métodos e propriedades da classe com o code behind directamente (objecto de apoio). Para além disso, o code behind pode definir um método Page_Load para tratar o pedido (controlador).

```
class ArtistPage...

      protected void Page_Load(object sender, System.EventArgs e) {
         db = new OleDbConnection(DB.ConnectionString);
         …
         applyDomainLogic (ds);
         DataBind();
      // prepareUI
         DataGrid1.DataSource = ds;
         DataGrid1.DataBind();
      }

      protected String name {
         get {return Request.Params["name"];}
      }
      protected String age {
         get {return Request.Params["age"];}
      }
      protected String numAlbums {
         get {return Request.Params["numAlbums "];}
      }
```

No método Page_Load, o método applyDomainLogic pode fazer a pesquisa de um artista por nome, por exemplo e colocar o resultado no DataSet passado. A chamada a DataBind faz a ligação da informação da fonte de dados aos objectos de domínio. A chamada a prepareUI serve para preparar alguns objectos para serem usados pela interface.

Os métodos com as propriedades funcionam como objecto de apoio para a vista, que pode referenciar estas propriedades da seguinte forma:

```
<P> Artist:
   <asp:label
          id="artistLabel"
          Text="<%# artist %>"
          runat="server"/> , 
   <asp:label
          id=ageLabel
          Text="<%# age %>"
          runat="server" font-bold="True">
   </asp:label>
<P>
```

Para mostrarmos a tabela com a lista de albuns em .NET, deve ser usado um componente denominado DataGrid (que é usado na implementação do padrão TableModule) pois este permite mostrar directamente na interface os dados de um DataSet vindo da camada de negócio.

De facto, a utilização deste componente é muito fácil dadas as potencialidades da feramenta Visual Studio. É possível fazer esta ligação através do código de preparação da interface mostrado no método Page_Load: *DataGrid1.DataSource = ds; DataGrid1.DataBind();*.

Neste caso, e como estamos a usar a plataforma .NET, a classe de apoio (code behind) implementa o padrão Table Module que implementa também a lógica de negócio da tabela de artista na Base de Dados.

Com o objecto DataGrid do ASP.NET é possível seleccionar que colunas da tabela mostrar e, ao mesmo tempo, controlar o aspecto da página web. Neste caso seleccionamos o nome do álbum e o ano de edição.

```
<asp:DataGrid id="DataGrid1" runat="server" Width="480px" Height="171px"
      BorderColor="#336666" BorderStyle="Double" BorderWidth="3px"
BackColor="White" CellPadding="4" GridLines="Horizontal"
AutoGenerateColumns="False">
   <SelectedItemStyle Font-Bold="True" ForeColor="White" BackColor="#339966"></
   SelectedItemStyle>
   <ItemStyle ForeColor="#333333" BackColor="White"></ItemStyle>
   <HeaderStyle Font-Bold="True" ForeColor="White"
BackColor="#336666"></HeaderStyle>
   <FooterStyle ForeColor="#333333" BackColor="White"></FooterStyle>
   <Columns>
      <asp:BoundColumn DataField="title" HeaderText="Album Title">
         <HeaderStyle Width="70px"></HeaderStyle>
      </asp:BoundColumn>
      <asp:BoundColumn DataField="year" HeaderText="Runs">
         <HeaderStyle Width="30px"></HeaderStyle>
      </asp:BoundColumn>
   </Columns>
   <PagerStyle HorizontalAlign="Center" ForeColor="White" BackColor="#336666"
   Mode="NumericPages"></PagerStyle>
</asp:DataGrid></P>
```

O HTML desta vista torna-se bastante assustador mas, de facto, o Visual Studio permite manipular todo este código através de listas de propriedades.

A vantagem de colocar estes controlos na web que compreendem os DataSets é que tornam a movimentação e visualização de dados trivial através do Visual Studio. Por outro lado, a desvantagem deste tipo de soluções é que só funciona quando é usado com o padrão Table Module. Quando a lógica de domínio se torna

complexa, aconelha-se o uso de um Domain Model que, para usar estas feramentas terá de criar os seus DataSets.

4.2.2. Transform View

Uma Transform View é uma vista que processa os dados do modelo elemento a elemento e transforma-os em HTML. Neste padrão é utilizado um estilo de programa de transformação. O exemplo típico é a linguagem XSLT. Esta tecnologia pode ser muito eficiente se a aplicação trabalhar com dados do domínio já em XML ou que possam facilmente ser convertidos para XML.

Um controlador de input selecciona a stylesheet XSLT apropriada e aplica-a a um XML retirado do modelo.

How It Works

Basicamente, uma Tranform View é um programa que converte dados de um modelo de domínio em HTML. O programa precorre a estrutura do modelo de dados e, à medida que reconhece os elementos do modelo de domínio, proiduz o HTMl apropriado.

A principal diferença entre uma Tranform View e uma Template View é a sua estrutura. Uma Templave View está organizada em volta do resultado. Uma Transform View é organizada em várias transformações, uma para cada tipo de elemento de entrada do modelo de domínio. A transformação é controlada por algo como um ciclo que percorre todos os elementos a serem transformados, procura a transformada apropriada e executa-a.

Apesar de ser possível escrever uma Transform View em qualquer linguagem, a escolha dominante para este padrão é a linguagem funcional XSLT.

Uma XSLT necessita de XML como dados de entrada, logo, o modelo deve retornar, de alguma forma, dados em XML, que possam ser transformados em HTML. Depois de obtido um ficheiro XML com os dados necessários ao resultado, é necessário passar esse XML a um motor de XSLT que, com uma stylesheet XSLT que inclua as regras de transformação, produz o resultado (que pode ser escrito directamente como resposta http).

When to Use It

A escolha entre Transform View e Template View depende da preferência da equipa que trabalha na parte da vista.

Enquanto que existem muitas ferramentas para edição de HTML, os editores de XSLT são poucos e muito menos sofisticados. A linguagem XSLT, por ser uma linguagem funcional, pode ser considerada mais difícil de aprender.

A linguagem XSLT é portável para quase qualquer plataforma web, isto é, a mesma stylesheet pode ser usada para transformar XML gerado numa plataforma J2EE ou .NET, etc. Pode inclusivamente ajudar a manter uma vista comum sobre dados de diferentes fontes. Por seu lado, as Template Views são, usualmente, escritas em linguagens específicas de plataformas: JSP, ASP, PHP, etc.

Se o modelo estiver naturalmente estruturado em XML, a linguagem XSLT é muito apropriada dado que não é necessário nenhum tipo de processamento do XML. Noutras linguagens, o programador, tem usualmente de criar objectos e usar uma API DOM ara navegar e transformar o documento XML.

As Transform Views evitam dois dos maiores problemas das Template Views: o excesso de lógica na vista e a dificuldade de execução de testes (dependência do servidor web).

Para não repetir tipos de transformação ao longo das stylesheet de uma aplicação, deve ser usada a funcionalidade do XSLT include, que ajuda a reduzir o problema da duplicação de alterações.

Exemplo simples de uma Tranform View

Para fazer uma transformação de um documento XML é necessário preparar o código procedimental que invoque a stylesheet correcta para produzir a resposta.

Na utilização de Tranform Views, o processamento da resposta a uma página é bastante semelhante, portanto é usual conjugar este padrão com o padrão Front Controller. Enquanto que a selecção da stylesheet e do documento XML com os dados é feita por cada um dos comandos específicos, o processamento da transformação é feito na classe do comando do Front Controller.

Neste padrão as diferenças entre as plataformas .NET e J2EE são mínimas, pelo que nos limitamos a dar um exemplo em Java:

```
class AlbumCommand...

   public void process() {
      try {
        Album album = Album.findNamed( request.getParameter( "name" ) );
        Assert.notNull(album);
        PrintWriter out = response.getWriter();
        XsltProcessor processor = new SingleStepXsltProcessor( "album.xsl" );
        out.print( processor.getTransformation( album.toXmlDocument() ) );
      } catch ( Exception e ) {
         throw new ApplicationException( e );
      }
   }
```

O documento XML pode ser:

```
<album>
   <title>Stormcock</title>
   <artist>Roy Harper</artist>
   <trackList>
      <track><title>Hors d'Oeuvres</title><time>8:37</time></track>
      <track><title>The Same Old Rock</title><time>12:24</time></track>
      <track><title>One Man Rock and Roll Band</title><time>7:23</time></track>
      <track><title>Me and My Woman</title><time>13:01</time></track>
   </trackList>
</album>
```

A transformação do documento XML é feita por um programa XSLT. Cada template trata uma parte do documento XML e transforma-a no HTML da página a produzir. Neste caso, é mantido o aspecto da página a uma simplicidade máxima para se compreender o essencial da linguagem.

O básico da transformação produz o topo da página:

```
<xsl:template match="album">
   <HTML><BODY bgcolor="white">
   <xsl:apply-templates/>
   </BODY></HTML>
</xsl:template>

<xsl:template match="album/title">
   <h1><xsl:apply-templates/></h1>
</xsl:template>

<xsl:template match="artist">
      <P><B>Artist: </B><xsl:apply-templates/></P>
</xsl:template>
```

As outras templates produzem a tabela com a lista de álbuns do artista:

```
<xsl:template match="trackList">
   <table><xsl:apply-templates/></table>
</xsl:template>

<xsl:template match="track">
   <tr><xsl:apply-templates/></tr>
</xsl:template>

<xsl:template match="track/title">
   <td><xsl:apply-templates/></td>
</xsl:template>
```

```
<xsl:template match="track/time">
   <td><xsl:apply-templates/></td>
</xsl:template>
```

4.2.3. Two Step View

Uma vista com um só passo tem, usualmente, um componente de visualização para cada ecrã da aplicação (ecrãs logicamente iguais podem partilhar vistas). A vista aceita dados do domínio e mostra-os em HTML.

Com o padrão Two Step View a transformação dos dados do domínio em HTML é feita a dois passos: a produção de uma página lógica a partir dos dados do domínio e, só depois, a transformação dessa página lógica em HTML.

Existe uma vista de primeiro passo para cada ecrã mas apenas uma vista de segundo passo para a aplicação inteira. Este padrão é usado quando se quer manter um aspecto consistente ao longo de uma aplicação.

A vantagem no uso deste padrão é que este coloca a decisão de qual o HTML a usar num só local. Isto faz com que as alterações globais ao HTML sejam fáceis dado que só existe um objecto para alterar todos os ecrãs da aplicação. Isto só funciona se a lógica de apresentação se mantém ao longo da aplicação, isto é, diferentes ecrãs usam a mesma lógica de apresentação de base.

Aplicações intensivamente gráficas não terão certamente uma estrutura de ecrã lógica.

Este padrão funciona ainda melhor se existirem diferentes front ends que necessitem de diferentes aspectos, basta, neste caso, usar um segundo passo diferente para cada front end.

Da mesma fora, este padrão pode ser usado para lidar com vários tipos de output, por exemplo, usando um segundo passo para um web browser normal e outra para um palmtop.

Se o aspecto dos ecrãs for muito diferente, este padrão não deve ser usado, como acontece frequentemente, por exemplo, na diferença entre a interface para um browser web e para um telefone portátil.

How It Works

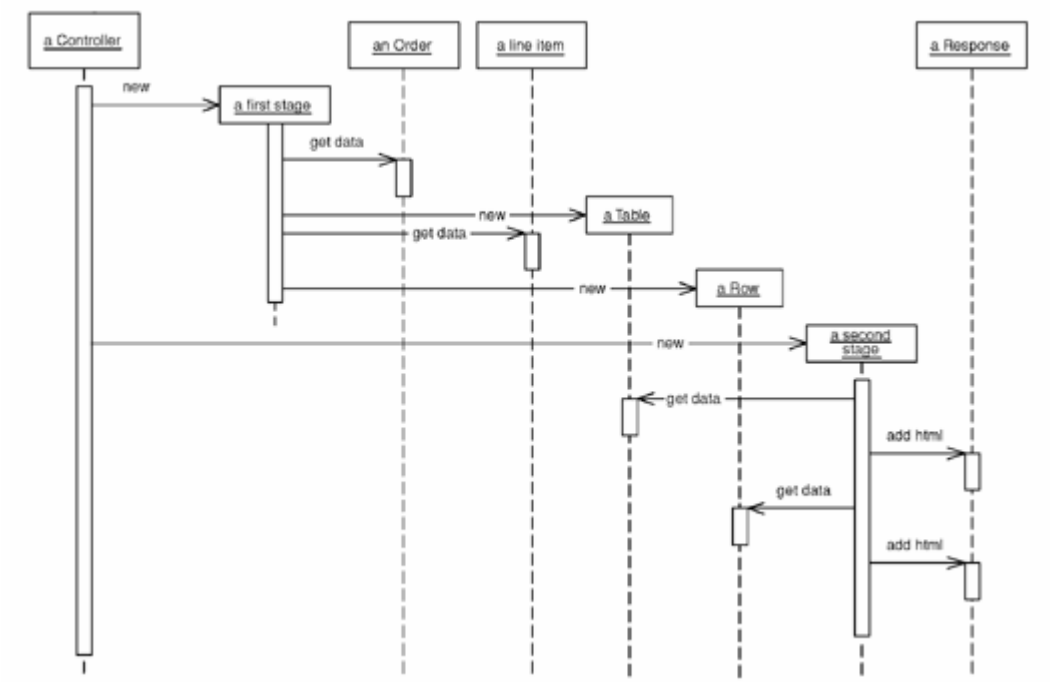
A chave para este padrão está em tornar a transformação dos dados em HTML num processo em dois passos. Dos dados numa página lógica daí em HTML.

Esta página lógica inclui elementos como campos, cabeçalhos, rodapés, tabelas, escolhas, etc. Esta página é orientada à apresentação e força a página a ter um determinado estilo. Nesta página são definidos os elementos constituintes da apresentação mas não é definido o aspecto do HTML.

Esta página lógica é construída por código escrito especificamente para a página, acedendo, por exemplo, a uma base de dados ou dados do domínio, que fornece os dados propriamente ditos que serão mostrados na página final. Este código tem apenas de construir a página lógica na sua forma já orientada à apresentação.

O segundo passo transforma esta página lógica em HTML. Este segundo passo conhece cada um dos elementos da página lógica e como transformar cada um em HTML. Assim, um sistema com múltiplos ecrãs pode ter apenas um segundo passo que transforma os elementos lógicos do primeiro passo em HTML sempre da mesma forma. Fazendo assim com que todas as decisões de HTML sejam feitas num só lugar. A restrição aqui é que o resultado em HTML tem de ser derivável dos elementos do ecrã lógico.

No seguinte diagrama de sequência podemos ver como é feita a construção de uma página numa Two Step View.



Existem várias formas de construir um sistema com o padrão Two Step View.

Para implementar uma Two Step View juntamente com uma Tranform View basta usar uma XSLT para cada passo. No exemplo da Tranform View tínhamos uma XSLT que transforma os dados de domínio em HTML. Neste caso teremos uma XSLT para transformar os dados do domínio num XML já orientado à apresnetação (uma para cada página) e uma XSLT para transformar esse XML orientado à aprensetação em HTML (uma para todo o sistema).

Para implementar uma Two Step View juntamente com uma Template View é necessário recorrer a marcas especiais. Neste caso, a Template View não inclui nenhumas marcas HTML mas apenas estas marcas especiais que representam a estrutura orientada à apresentação. Estas marcas são transformadas posteriormente em HTML. Por exemplo, na Template View podemos incluir uma marca <field label = "Name" value = "getName" /> que, num segundo passo, é transformada num conjunto de marcas HTML apropriadas.

When To Use It

Ao decidir que tipo de vista usar para uma aplicação empresarial é necessário ter em conta se a aplicação vai usar o padrão Two Step View em conjunto com um dos outros dois padrões de visualização.

A principal vantagem deste padrão é a separação entre o primeiro e o segundo passo, o que permite fazer alterações globais ao sistema, no segundo passo, muito facilmente.

Este padrão facilita a manutenção de um aspecto coerente em aplicações web em que isso é pretendido. Por outro lado, este padrão possibilita gerar muito facilmente vários aspectos para a mesma aplicação, apenas através da alteração do segundo passo. Por exemplo, numa aplicação com 10 páginas e um aspecto, com uma vista a um passo são necessárias 10 módulos de processamento do primeito passo tal como numa vista a dois passos, em que é necessário adicionlemtne a construção do segundo passo. Mas, se passarmos para uma aplicação com 10 páginas e 3 aspectos, com uma vista a um passo passamos a ter 30 modulos de processamento do primeiro passo enquanto que numa vista a dois passos precisamos apenas de 10 mais 3 de segundo passo. Quanto maior a aplicação maior é o ganho no uso deste padrão.

No entanto, esta vantagem só pode ser obtida se a estrutura orientada à apresnetação suportar as necessidades da apresentação. Por exemplo, aplicações

com grande componente de design, em que cada página deve parecer diferente das outras, não funcionam bem com este padrão pois a aplicação é limitada pela estrutura intermédia orientada à apresentação. Muitas vezes esta limitação é demasiada.

Outra desvantagem deste padrão é que existem muitas ferramentas para gerar e editar HTML usando o padrão Template View mas as Two Step Views obrigam os programadores a escrever os transormadores da estrutura intermédia em HTML, logo, todas as alterações de design têm de envolver os programaores.

É também verdade que este padrão com a sua estrutura em camadas representa um modelo de programação mais difícil de aprender, apesar de depois de aprendido pode ser bastante fácil de usar e pode até ajudar a diminuir a duplicação de código.

Para segundos passos para difenretes aparelhos, existe a limitação de se ter de usar a mesma estrutura lógica e, para aparelho demasiado, isto pode não ser aceitável.

Exemplo de uma XSLT a dois passos

Este exemplo implementa uma transformação XSLT a dois passos: a primeira transformação XSLT transforma os dados de domínio em XML numa página lógica em XML e o segundo passo transforma essa página lógica em HTML.

Um exemplo do XML com os dados de domínio é:

```
<album>
   <title>Zero Hour</title>
   <artist>Astor Piazzola</artist>
   <trackList>
      <track><title>Tanguedia III</title><time>4:39</time></track>
      <track><title>Milonga del Angel</title><time>6:30</time></track>
      <track><title>Concierto Para Quinteto</title><time>9:00</time></track>
      <track><title>Milonga Loca</title><time>3:05</time></track>
      <track><title>Michelangelo '70</title><time>2:50</time></track>
      <track><title>Contrabajisimo</title><time>10:18</time></track>
      <track><title>Mumuki</title><time>9:32</time></track>
   </trackList>
</album>
```

O XML orientado à apresnetção (página lógica) a gerar deve ser qualquer coisa como:

```
<screen>
   <title>Zero Hour</title>
   <field label="Artist">Astor Piazzola</field>
   <table>
      <row><cell>Tanguedia III</cell><cell>4:39</cell></row>
      <row><cell>Milonga del Angel</cell><cell>6:30</cell></row>
      <row><cell>Concierto Para Quinteto</cell><cell>9:00</cell></row>
      <row><cell>Milonga Loca</cell><cell>3:05</cell></row>
      <row><cell>Michelangelo '70</cell><cell>2:50</cell></row>
      <row><cell>Contrabajisimo</cell><cell>10:18</cell></row>
      <row><cell>Mumuki</cell><cell>9:32</cell></row>
   </table>
</screen>
```

A XSLT que faz esta primeira transformação é algo semelhante a:

```
<xsl:template match="album">
   <screen><xsl:apply-templates/></screen>
</xsl:template>

<xsl:template match="album/title">
   <title><xsl:apply-templates/></title>
</xsl:template>

<xsl:template match="artist">
   <field label="Artist"><xsl:apply-templates/></field>
</xsl:template>

<xsl:template match="trackList">
```

```
   <table><xsl:apply-templates/></table>
</xsl:template>

<xsl:template match="track">
   <row><xsl:apply-templates/></row>
</xsl:template>

<xsl:template match="track/title">
   <cell><xsl:apply-templates/></cell>
</xsl:template>

<xsl:template match="track/time">
   <cell><xsl:apply-templates/></cell>
</xsl:template>
```

Para transformar a página lógica em HTML usamos a seguinte XSLT:

```
<xsl:template match="screen">
   <HTML><BODY bgcolor="white">
   <xsl:apply-templates/>
   </BODY></HTML>
</xsl:template>

<xsl:template match="title">
   <h1><xsl:apply-templates/></h1>
</xsl:template>

<xsl:template match="field">
   <P><B><xsl:value-of select = "@label"/>: </B><xsl:apply-templates/></P>
</xsl:template>

<xsl:template match="table">
   <table><xsl:apply-templates/></table>
</xsl:template>

<xsl:template match="table/row">
   <xsl:variable name="bgcolor">
      <xsl:choose>
         <xsl:when test="(position() mod 2) = 1">linen</xsl:when>
         <xsl:otherwise>white</xsl:otherwise>
      </xsl:choose>
   </xsl:variable>
   <tr bgcolor="{$bgcolor}"><xsl:apply-templates/></tr>
</xsl:template>

<xsl:template match="table/row/cell">
   <td><xsl:apply-templates/></td>
</xsl:template>
```

Adicionalmente é necessário implementar o controlador que faz as duas transformações (ver exemplo do Transform View):

```
class AlbumCommand...

   public void process() {
      try {
         Album album = Album.findNamed(request.getParameter("name"));
         album = Album.findNamed("1234");
         Assert.notNull(album);
         PrintWriter out = response.getWriter();
         XsltProcessor processor = new TwoStepXsltProcessor("album2.xsl",
"second.xsl");
         out.print(processor.getTransformation(album.toXmlDocument()));
      } catch (Exception e) {
         throw new ApplicationException(e);
      }
   }
```

No exemplo do padrão Tranform View, para alterar qualquer detalhe de HTML teríamos de editar todas as XSLTs, usando as Two Step Views, é apenas necessário alterar a XSLT do segundo passo.

Exemplo de um JSP com Custom Tags em Java

Para além da dupla transformação XSLT, uma das formas das Two Step Views mais usadas são as *custom tags* com JSPs.

O fundamento das Two Step Views é que a decisão do que mostrar é feita separadamente da decisão de que HTML usar.

Neste exemplo, o primeiro passo da vista é uma página JSP (que usa o seu objecto de apoio) e o segundo passo é um conjunto de custom tags. Cada custom tag usada no JSP é transformada num conjunto específico e configurável de tags HTML.

O JSP do primeiro passo é o seguinte:

```
<%@ taglib uri="2step.tld" prefix = "2step" %>
<%@ page session="false"%>
<jsp:useBean id="helper" class="actionController.AlbumConHelper"/>
<%helper.init(request, response);%>
<2step:screen>
<2step:title><jsp:getProperty name = "helper" property = "title"/></2step:title>
<2step:field label = "Artist"><jsp:getProperty name = "helper" property =
"artist"/></
   2step:field>
<2step:table host = "helper" collection = "trackList" columns = "title, time"/>
</2step:screen>
```

As marcas do namespace 2step são custom tags e são transformadas em HTML no segundo passo. Podemos ver neste exemplo que não existem marcas HTML, apenas marcas de acesso ao objecto de apoio e marcas de segundo passo.

Cada marca de segundo passo tem uma implementação própria. Por exemplo, a marca title terá a seguinte implementação:

```
class TitleTag...

   public int doStartTag() throws JspException {
      try {
         pageContext.getOut().print("<H1>");
      } catch (IOException e) {
         throw new JspException("unable to print start");
      }
      return EVAL_BODY_INCLUDE;
   }

   public int doEndTag() throws JspException {
      try {
         pageContext.getOut().print("</H1>");
      } catch (IOException e) {
         throw new JspException("unable to print end");
      }
      return EVAL_PAGE;
   }
```

As custom tags são implementadas através de métodos do tipo hook que são chamados no início e no final do processamento das marcas de segundo passo.

Neste caso, a marca limita-se a colocar a marca <h1> em torno do conteúdo da marca de segundo passo <2step:title>.

As marcas mais elaboradas como a de tabelas poderão ter processamentos mais sofisticados.

4.3. Padrões para os Controladores

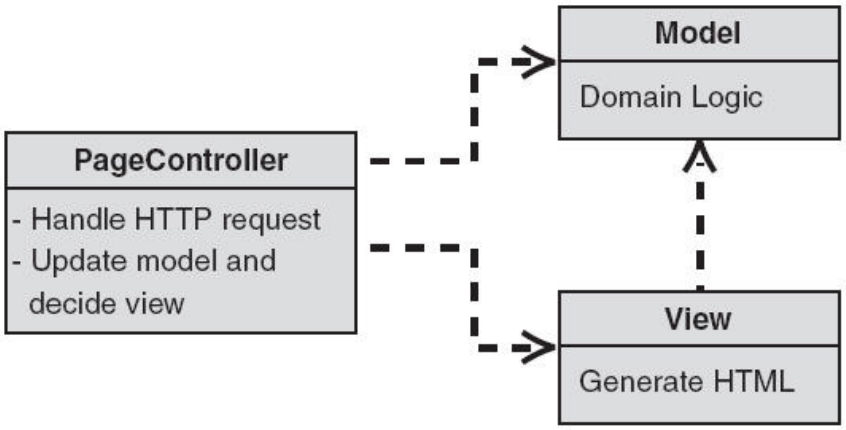
No padrão MVC, a primeira separação é entre a interface do utilizador e os componentes de negócio. Neste ponto apresentamos os padrões que fazem a segunda divisão no MVC: a separação do código da apresentação em vistas e controladores.

Esta divisão deve ser feita apenas em aplicações Web com muitas páginas onde existam múltiplas acções e onde se utiliza a mesma camada de apresentação.

Este padrão vem retirar das vistas a lógica de navegação das aplicações e, portanto, torná-las mais simples.

4.3.1. Page Controller

O Page Controller é um objecto que trata os pedidos HTTP para uma única página ou acção específica da aplicação web. Este objecto aceita os pedidos, invoca a acção no modelo correspondente ao pedido recebido e determina qual a vista a utilizar. A figura seguinte mostra como o controlador se relaciona com o modelo e com a vista:



Como se mostra na figura, o Page Controller recebe o pedido de uma página, processa-o, invoca alterações ao modelo, decide a vista a utilizar e invoca-a.

O Page Controller depende do modelo e da vista. A vista, por sua vez, depende do modelo para a recuperação dos dados a serem mostrados.

How It Works

No contexto das páginas estáticas, um pedido http é feito passando o nome e o caminho para o documento HTML que se encontra armazenado no servidor. De igual forma no padrão Page Controller é usado este esquema em que um caminho dá a um ficheiro: o Page Controller.

A ideia base é ter um módulo no servidor web que age como controlador para cada página da aplicação web. Na prática não há ligação entre um módulo e uma página mas existe um Page Controller para cada acção, onde uma acção é um botão ou um link da aplicação. Na maioria das vezes as acções correspondem a páginas de destino, mas ocasionalmente não, por exemplo, um link pode ir para diferentes páginas consoante uma condição.

Existem duas formas típicas de estruturar o Page Controller:

- Script: o Page Controller é implementado num script (CGI script, servlet, etc.) que faz o papel de controlador e depois redirecciona o pedido para a vista apropriada (tipicamente uma server page) que mostra os resultados;
- Server Page: o Page Controller reside numa server page (JSP, ASP, PHP, etc) que combina os papéis de controlador e vista. Para tal deve usar um objecto de apoio que é chamado no início da server page para processar toda a lógica. Este objecto de apoio pode retornar o controlo para a server page original ou redireccionar o pedido para uma server page diferente para agir como vista.

É possível uma solução mista com alguns pedidos controlados por server pages e outros por scripts. Pedidos que não têm lógica de controlo associada podem ser tratados por Server Pages dado que isso representa um mecanismo simples de compreender e alterar. Os pedidos com lógicas mais complexas devem ser tratados por scripts.

As responsabilidades básicas de um Page Controller são:

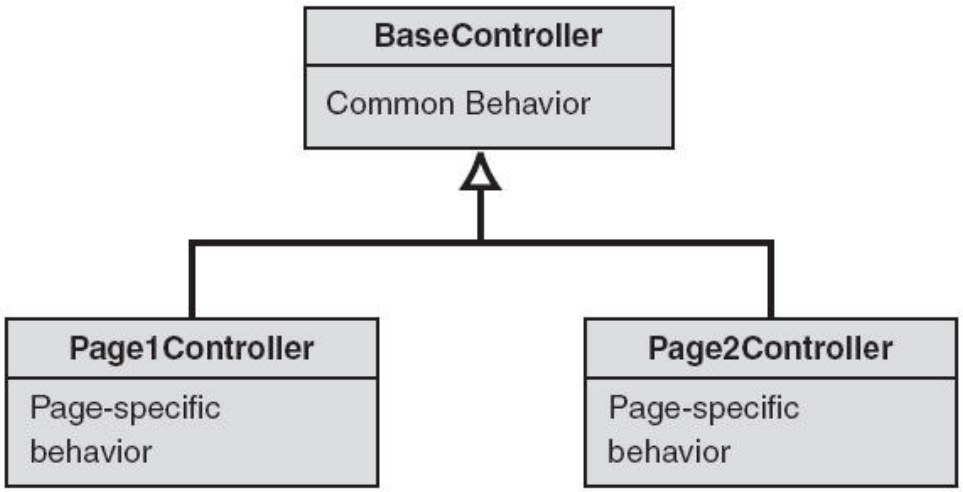
- Descodificar o URL e extrair todos os dados do pedido;

- Criar e invocar qualquer objecto do modelo para processar os dados. Todos os dados do pedido http devem ser passados ao modelo para que este não necessite de aceder ao pedido http;
- Determinar que vista deve mostrar a página com os resultados e enviar a informação do modelo para a vista seleccionada.

Variação: BaseController

Uma variação/extensão do padrão Page Controller, no caso dos controladores implementados num script, é o BaseController. Nesta solução, é criada adicionalmente uma classe base que contém funcionalidades transversais aos controladores, como por exemplo, extracção de dados da query string, validações de parâmetros, autenticação, internacionalização, gestão de cabeçalhos e rodapés, etc. Todos os Page Controllers devem estender o Base Controller onde são implementadas as funcionalidades comuns. Este esquema permite evitar a duplicação de código comum ao processamento dos pedidos.

A figura seguinte resume esta solução:



When to Use It

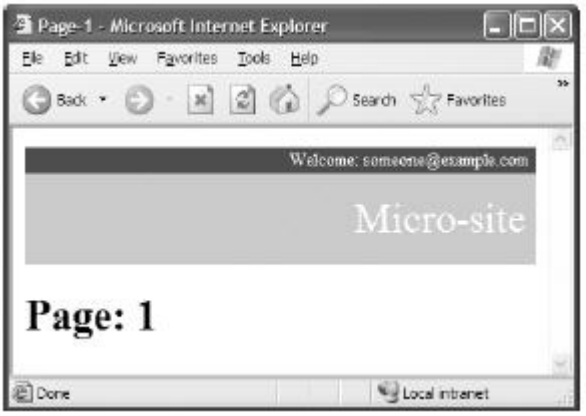
Quanto aos controladores, a principal opção é entre o Page Controller e o Front Controller. Dos dois padrões, o Page Controller é o mais simples e leva à estruturação natural do mecanismo de controlo. A opção pelo Front Controller leva à maior complexidade de implementação e compreensão do Front Controller mas também às inúmeras vantagens que o uso deste padrão acarreta, nomeadamente no tratamento da complexidade de navegação.

O padrão Page Controller funciona bem quando a lógica de navegação é bastante simples, casos em que o padrão Front Controller pode até adicionar overheads e complexidade desnecessários.

É ainda possível uma solução mista em que ambos os padrões são usados. A mistura dos dois não é um grande problema e acontece inevitavelmente no refactoring de uma aplicação de um padrão para outro.

Implementação em .NET

O exemplo seguinte implementa o padrão Page Controller em Microsoft .NET. O objectivo é usar este padrão para construir o cabeçalho de uma página Web que mostra o email do utilizador autenticado (vindo da base de dados). É utilizada a variação Base Controller apresentada anteriormente pois este cabeçalho é utilizado em todas as páginas da aplicação. A implementação baseia-se portanto na construção de uma classe base que todos os controladores vão estender. A figura seguinte mostra uma das páginas deste site.

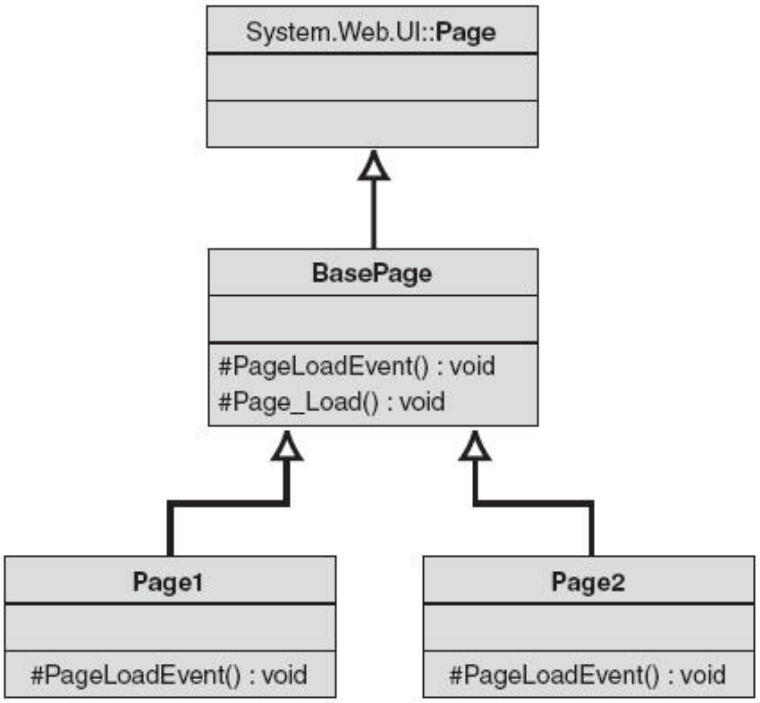


Cada uma das páginas é responsável pelo tratamento dos seus conteúdos, enquanto que a classe base tem a responsabilidade de preencher o cabeçalho. Uma vez que todas as páginas herdam da classe base, todas elas têm as mesmas funcionalidades.

Na classe base deste padrão é também implementado o padrão *Template Method*. Este padrão define o esqueleto de um algoritmo deixando a (re)definição de alguns métodos para as sub classes (as sub classes não podem mexer na sua estrutura base do algoritmo). O processamento dos pedidos é o Template Method da classe base. Apenas as partes não comuns do processamento dos pedidos são implementadas pelas sub classes, isto é, os controladores específicos.

A aplicação da variação Base Controller a este problema implica mover algum código de cada uma das páginas para a classe base. Isto garante que o código comum a todas as páginas está num único sítio sendo assim de mais fácil gestão.

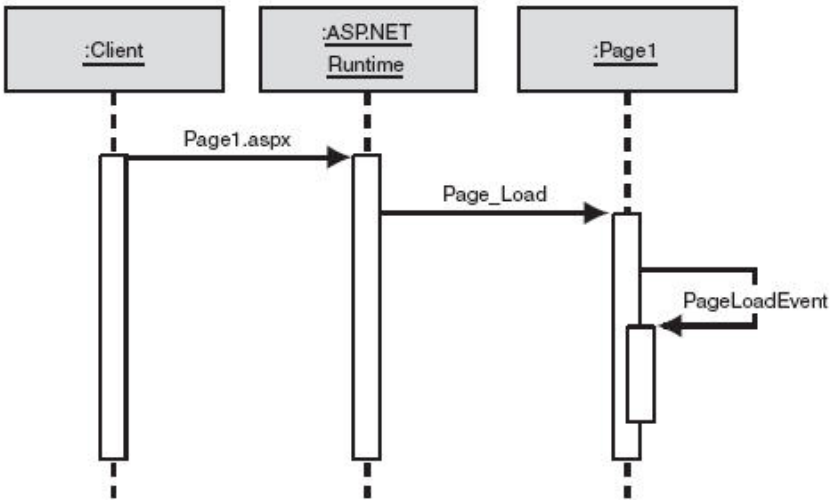
Neste exemplo, a classe base chama-se ***BasePage*** e é responsável por efectuar a ligação do método ***Page_Load*** ao evento ***Load***. Depois da associação à BasePage, que vai à base de dados e retira o email do utilizador autenticado, a função ***Page_Load*** chama o método ***PageLoadEvent*** que é implementado por cada uma das subclasses que executam o seu código correspondente. A figura seguinte exemplifica esta estrutura.



Como estamos na plataforma .NET, estes controladores são sempre implementados no code behind da vista respectiva, daí as classes terem o prefixo Page. Neste caso específico existem duas páginas da aplicação (Page1 e Page2) e uma terceira página (BasePage) que incluem vista e code behing. A parte da vista da BasePage representa uma parte comum às vistas (ficheiro BasePage.inc), a parte do code behind da BasePage representa a implementação da variação Base Controller do Page Controller (ficheiro BasePage.cs).

Quando uma página é chamada, o evento ***Load*** é chamado, que por sua vez chama o método ***Page_Load*** na ***BasePage***. O método ***BasePage*** retorna os dados que precisa e depois chama o evento ***PageLoadEvent*** na página que foi chamada para executar uma determinada acção.

A figura seguinte ilustra esta sequência de pedidos.



A página ***BasePage.cs*** tem as seguintes funcionalidades:

- Ligar o evento ***Load*** ao método ***Page_Load*** para inicializar determinado pedido;
- Recolhe o nome do utilizador e com o apoio da class DatabaseGateway encontra o email do mesmo para ser mostrado no label definido no cabeçalho;
- Associa o nome do site ao label correspondente;
- Evoca o método PageLoadEvent que é implementado nas sub classes.

```
public class BasePage : Page
{
    protected Label eMail;
    protected Label siteName;

    virtual protected void PageLoadEvent(object sender, System.EventArgs e)
    {}

    protected void Page_Load(object sender, System.EventArgs e)
    {
          if(!IsPostBack)
          {
                string name = Context.User.Identity.Name;

                eMail.Text = DatabaseGateway.RetrieveAddress(name);
                siteName.Text = "Micro-site";

                PageLoadEvent(sender, e);
          }
    }
    private void InitializeComponent()
    {
          this.Load += new System.EventHandler(this.Page_Load);
    }
}
```

O ficheiro BasePage.inc contém o código HTML correspondente ao cabeçalho e que deverá ser incluído em cada uma das páginas de apresentação de dados.

```
<table width="100%" cellspacing="0" cellpadding="0">
   <tr>
      <td align="right" bgcolor="#9c0001" cellspacing="0" cellpadding="0"
width="100%" height="20">
         <font size="2" color="#ffffff">Welcome:
         <asp:Label id="eMail" runat="server">username</asp:Label>  </font>
      </td>
   </tr>
   <tr>
      <td align="right" width="100%" bgcolor="#d3c9c7" height="70">
         <font size="6" color="#ffffff">
         <asp:Label id="siteName" Runat="server">Micro-site Banner</
asp:Label>  </font>
      </td>
   </tr>
</table>
```

Neste exemplo, o acesso à base de dados é assegurado pela classe DataBaseGateway.

```
public class DatabaseGateway
{
   public static string RetrieveAddress(string name)
   {
         String address = null;

         String selectCmd =
               String.Format("select * from webuser where (id = '{0}')",
               name);

         SqlConnection myConnection =
               new
SqlConnection("server=(local);database=webusers;Trusted_Connection=yes");
          SqlDataAdapter myCommand = new SqlDataAdapter(selectCmd, myConnection);

         DataSet ds = new DataSet();
         myCommand.Fill(ds,"webuser");
         if(ds.Tables["webuser"].Rows.Count == 1)
         {
               DataRow row = ds.Tables["webuser"].Rows[0];
               address = row["address"].ToString();
         }

         return address;
   }
}
```

A implementação da vista de uma das páginas é a seguinte:

```
<%@ Page language="c#" Codebehind="Page1.aspx.cs" AutoEventWireup="false"
Inherits="Page1" %>
<HTML>
   <HEAD>
      <title>Page-1</title>
   </HEAD>
   <body>
      <!-- #include virtual="BasePage.inc" -->
      <form id="Page1" method="post" runat="server">
         <h1>Page:
            <asp:label id="pageNumber" Runat="server">NN</asp:label></h1>
      </form>
   </body>
</HTML>
```

The following directive from the file loads the common HTML for the header:

```
<!-- #include virtual="BasePage.inc" -->
```

Podemos notar a inclusão do cabeçalho para a página com a inclusão da linha ***<!--#include virtual="BasePAge.inc"-->***.

O code behind respectivo é o seguinte:

```
using System;
using System.Web.UI;
using System.Web.UI.WebControls;

public class Page1 : BasePage
{
   protected System.Web.UI.WebControls.Label pageNumber;

   protected override void PageLoadEvent(object sender, System.EventArgs e)
   {
      pageNumber.Text = "1";
   }
}
```

Podemos reparar que na Page1.aspx.cs a classe do code-behind herda da classe ***BasePage*** e implementa o método ***PageLoadEvent***.

4.3.2. Front Controller

Um controlador tem sempre duas responsabilidades: processar o pedido http e decidir o que fazer com ele. Muitas das vezes faz sentido separar estas duas responsabilidades

O Front Controller é um objecto controlador que trata todos os pedidos de uma aplicação web. Este processador de pedidos interpreta o URL para saber com que tipo de pedido está a lidar e cria um objecto separado para processá-lo. Um script pode processar o pedido http e delegar a um objecto de apoio separado a decisão do que fazer com ele.

Desta forma é possível centralizar o processamento de todos os pedidos http num único objecto evitando a necessidade de reconfigurar o servidor web sempre que é mudada a estrutura de acções da aplicação.

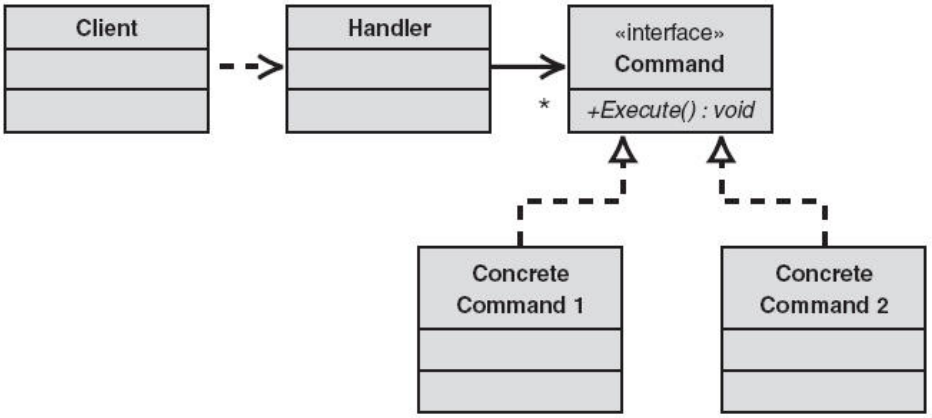
Um Front Controller consolida o tratamento de pedidos através da canalização dos pedidos para um único objecto de tratamento. Este objecto pode fazer tratamentos comuns a todos os pedidos (que pode ser alterado em runtime com decorators) e depois redireciona o pedido para objectos comando para comportamentos específicos aos pedidos.

How It Works

O Front Controller resolve o problema da descentralização que existe no Page Controller pois cada pedido é tratado por um só controlador.

- No Front Controller os pedidos de uma aplicação são estruturado em duas partes: um handler web e uma hierarquia de comandos. O handler web processa os pedidos http e, usando os parâmetros do pedido, escolhe qual o comando a executar e executá-o.

A figura seguinte ilustra o esquema básico do padrão Front Controller:

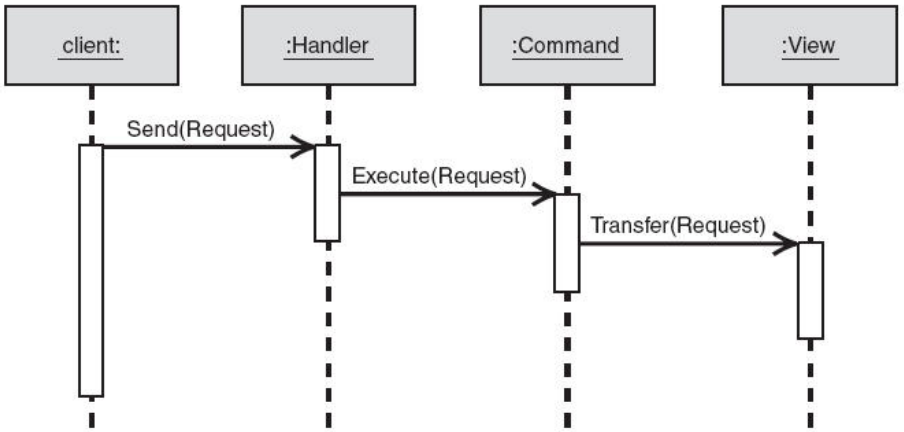


Existem duas formas distintas de implementar no handler a escolha do comando a executar:

- Estática: envolve o parsing do URL e lógica condicional;
- Dinâmica: envolve retirar uma parte standard do URL e usar a instanciação dinâmica para criar uma classe comando.

A versão estática tem a vantagem da lógica explícita, erros em compile time e muita flexibilidade para gerir os URLs. A versão dinâmica tem a vantagem de se poder adicionar comandos sem mudar o handler web. Na versão dinâmica o nome da classe de comando pode ir no URL ou pode ser usado um ficheiro de propriedades que faz a ligação entre URLs e classes.

A figura seguinte apresenta um cenário típico:



O cliente envia o pedido, o handler (depois de escolher qual o comando a executar) executa o comando, o comando evoca a lógica de negócio e (depois de decidir qual a vista a chamar) transfere o controlo para a vista.

Tanto o handler como os comandos são quase sempre objectos normais pois não produzem resposta mas apenas implementam o papel do controlador.

Variação: handler a dois passos

Uma variação do Front Controller é a criação de um handler a dois passos: um handler do pedido e um dispatcher. A única responsabilidade do handler é processar o pedido HTTP (parâmetros associados) e passá-lo ao dispatcher de tal forma que o dispatcher fique completamente independente do servidor Web. Isto torna os testes da aplicação muito mais fáceis pois o código de teste pode interagir directamente com o dispatcher sem ter de executar um servidor Web.

Prós e Contras

O padrão Front Controller tem um desenho mais complicado que a sua alternativa directa, o padrão Page Controlller, mas tem também uma série de vantagens que justificam o seu uso em determinadas circunstâncias.

Assim que o Front Controller está configurado no servidor web, o handler web faz o resto do trabalho. Isto simplica a configuração do servidor web que muitas vezes é difícil e trabalhoso de configurar.

No Front Controller só existe um controlador pelo que o seu comportamento pode ser melhorado e/ou extendido com a utilização de Decorators. Podem ser usados Decorators para autenticação, codificação de caracteres, internacionalização, logging, etc. Esta abordagem é descrita no padrão Intercepting Filter que define um decorator que encapsula o handler do front controller permitindo a criação de uma cadeia de filtros para lidar com questões como autenticação, logging e internacionalização.

Uma vantagem normalmente apontada do Front Controller face ao Page Controller é que este não implica a duplicação de código que é muitas vezes replicada pelos Page Controllers. De facto, esta questão pode ser facilmente resolvida no padrão Page Controller usando a variação Base Controller. A implementação desta variação do Page Controller envolve a criação de uma class base para ser partilhada por uma série de páginas. O problema surge quando estas classes base crescem de tal forma que o código inserido nestas não é comum a todas as páginas. É necessário periodicamente rever esta classe para garantir que o código é comum a todas as páginas.

No Page Controller existe um objecto por cada página. Esta solução é impraticável quando é necessário controlar processos através de múltiplas páginas. Supondo por exemplo que existe uma navegação de páginas relativamente complexa e que está guardada num ficheiro XML. Quando um pedido chega, a aplicação deve saber encontrar para onde ir com base no estado actual.

A associação de um URL a um controlador pode ser uma má opção para uma aplicação web. Por exemplo, supondo que o site tem uma interface do tipo wizard para recolha de informação e que a navegação nesta depende das escolhas do mesmo. Este tipo de situação implementada no Page Controller obriga a que a lógica de navegação esteja implementada na classe base.

Implementação em .NET: Base

Suponhamos que, como no exemplo anterior, existe um site com um cabeçalho que contém a informação do utilizador autenticado na aplicação. Desta vez o cabeçalho será diferente dependendo do URL.

Este exemplo cria dois sites: Micro-Site e Macro-Site. Cada um consulta uma base de dados diferente para ler a informação relativa ao utilizador autenticado. As páginas por sua vez continuam inalteradas, só o cabeçalho é que se altera.

Neste exemplo, toda a implementação é muito idêntica ao Page Controller, a única classe que deve ser modificada é o ***BasePage***. A figura seguinte mostra essa alteração.

```
.....
protected void Page_Load(object sender, System.EventArgs e)
{
   if(!IsPostBack)
   {
      string site = Request["site"];

      if(site != null && site.Equals("macro"))
         LoadMacroHeader();
      else
         LoadMicroHeader();

      PageLoadEvent(sender, e);
   }
}

private void LoadMicroHeader()
{
   string name = Context.User.Identity.Name;

   eMail.Text = WebUsersDatabase.RetrieveAddress(name);
   siteName.Text = "Micro-site";
}


private void LoadMacroHeader()
{
   string name = Context.User.Identity.Name;

   eMail.Text = MacroUsersDatabase.RetrieveAddress(name);
   siteName.Text = "Macro-site";
}

.....
```

O Micro Site como o Macro Site utilizam bases de dados diferentes para preencher o endereço de email do utilizador autenticado. Os dois métodos ***LoadMicroHeader*** e ***LoadMacroHeader*** utilizam diferentes classes gateway, ***WebUsersDatabase*** e ***MacroUsersDatabase*** para ler informação da base de dados. As responsabilidades do método ***PageLoad*** foram alteradas: no exemplo do PageController lia informação da base de dados, nesta implementação escolhe qual

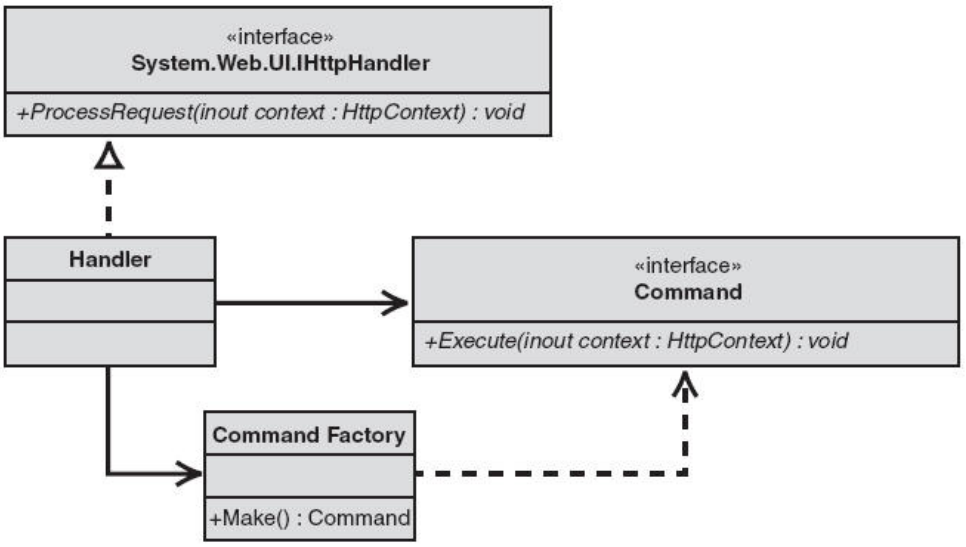
a função (***LoadMicroHeader*** ou ***LoadMacroHeader***) que deverá ser executada e executa-a a seguir.

Implementação em .NET: com Command Factory

O exemplo que se segue implementa o mesmo esquema de sites do exemplo anterior, mas utiliza uma Command Factory para gerar os comandos de tratamento os pedidos.

O Front Controller, regra geral, é implementado em duas partes. O objecto handler que recebe os pedidos (http Get e http Post) do servidor web, recolhe os parâmetros mais importantes e selecciona o comando apropriado baseado nos parâmetros recolhidos. A segunda parte do controlador, o Command Processor, executa uma acção específica para satisfazer o pedido http.

Na seguinte figura podemos ver o esquema usado neste exemplo:



A class ***Handler*** redirecciona o pedido web e delega a responsabilidade de determinar o objecto ***Comando*** correcto a executar à classe ***CommandFactory***. Quando a ***CommandFactory*** retorna um objecto ***Command***, o ***Handler*** chama o método ***Execute*** no ***Command*** para tratar o pedido.

Handler.cs

```
using System;
using System.Web;
public class Handler : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
          Command command =
CommandFactory.Make(context.Request.Params);
          command.Execute(context);
    }
    public bool IsReusable
    {
          get { return true;}
    }
}
```

Command.cs

A classe ***Command*** é util nesta situação pois não queremos que a classe ***Handler*** dependa directamente dos comandos mas que possa ser retornada do ***CommandFactory***.

```
using System;
using System.Web;
public interface Command
{
    void Execute(HttpContext context);
}
```

CommandFactory.cs

Esta classe é fundamental para a implementação. Com base nos parâmetros do form (query string) cria o comando a ser executado. Neste caso o query string será "site" que poderá ser *micro* ou *macro* e que cria os comandos MicroSite ou MacroSite respectivamente. Se o valor da query string não contiver nada será retornado ***UnknownCommand***.

```
using System;
using System.Collections.Specialized;

public class CommandFactory
{
    public static Command Make(NameValueCollection parms)
    {
          string siteName = parms["site"];
          Command command = null;
          if(siteName == null || siteName.Equals("micro"))
                command = new MicroSite();
          else if(siteName.Equals("macro"))
                command = new MacroSite();
          return command;
    }
}
```

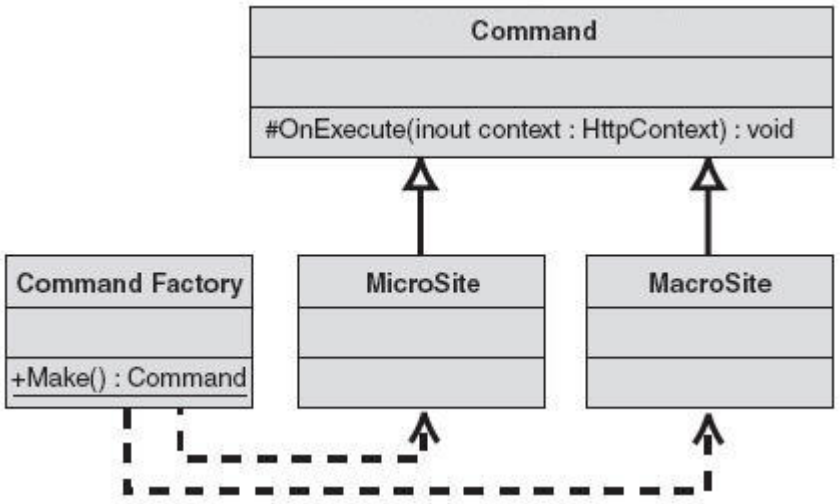
Configuração do Handler

No ASP.NET a configuração dos handlers é feita no ficheiro web.config. O ASP.NET define uma secção <***httphandlers***> onde os handlers podem ser adicionados ou removidos. Neste caso, o ASP.NET mapeia todos os pedidos do tipo Page*.aspx para a ***classe Handler*** definida no ficheiro web.config.

```
<httpHandlers>
  <add verb="*" path="Page*.aspx" type="Handler,FrontControllerMVC" />
</httpHandlers>
```

Command

O command representa a variação no web site. Neste exemplo, a funcionalidade de retornar dados da base de dados para cada site está contido nesta. Quando o objecto ***Execute*** é chamado, primeiro chama um método abstracto chamado ***OnExecute*** e no retorno transfere o controlo para a View. A View é obtida a partir da leitura do Web.config onde estão definidas as views a correr.




```
...
public abstract class Command
{
 protected  abstract void OnExecute(HttpContext context);
 public void Execute(HttpContext context)
 {
        OnExecute(context);
        string url = context.Request.Url.AbsolutePath;

        /// ********************************
        /// ******* Redirecting Commands ****
        /// ********************************
        foreach ( string key in
                    ConfigurationSettings.AppSettings.Keys)
        {
              if (context.Request.Url.AbsolutePath == key )
              {
                    url = (string) ConfigurationSettings.AppSettings[key];
              }
        }
        url=String.Format("{0}{1}",url,context.Request.Url.Query);

        //Transfere o control para a View
        context.Server.Transfer(url);
 }
}
```

MicroSite e MacroSite

Estas classes são idênticas ao código do LoadMicroHeader e LoadMacroHeader respectivamente do exemplo1. A diferença está no facto de não fazerem nenhum acesso às labels da página. Em vez disso, é necessário incluir informação relativa ao objecto HttpContext.

```
using System;
using System.Web;
public class MacroSite : RedirectingCommand
{
     protected override void OnExecute(HttpContext context)
     {
           string name = context.User.Identity.Name;
           context.Items["address"] =
     MacroUsersDatabase.RetrieveAddress(name);
```

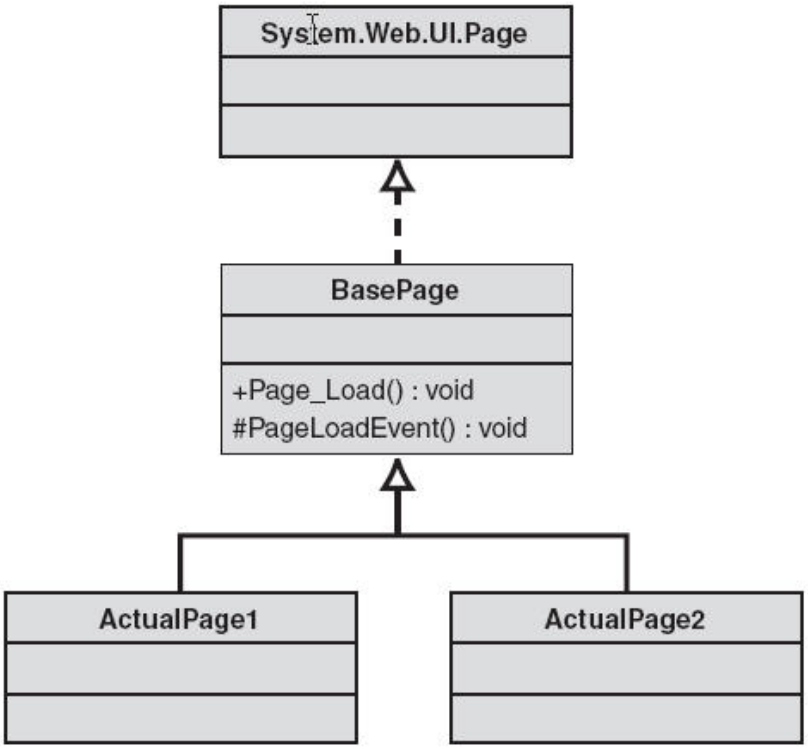
```
                context.Items["site"] = "Macro-Site";
            }
        }
```

WebUsersDatabase.cs e MacroUsersDatabase.cs

Estas duas classe são responsáveis por ler os dados das respectivas bases de dados (endereço de email).

Views

As vistas são o último aspecto da implementação. A vista no primeiro exemplo era responsável por ler os dados da base de dados dependendo do site que o utilizador estava a pedir e apresentava os dados. Neste exemplo, o acesso à base de dados foi passado para o command e a apresentação recebe os dados pelo objecto ***httpcontext***. Como ainda existem procedimentos comuns, existe a necessidade das páginas herdarem de uma base page para evitar duplicação de código. A figura seguinte ilustra esta estrutura.



BasePage.cs

A classe BasePage já não é responsável por saber qual o cabeçalho a incluir, mas simplesmente mostrar os dados que o command guarda no objecto ***httpcontext*** e associa-o ao label respectivo.

```
public class BasePage : Page
{
    protected Label eMail;
    protected Label siteName;
    virtual protected void PageLoadEvent(object sender, System.EventArgs e)
    {}
    protected void Page_Load(object sender, System.EventArgs e)
    {
        eMail.Text = (string)Context.Items["address"];
        siteName.Text = (string)Context.Items["site"];
        PageLoadEvent(sender, e);
    }
}
```

ActualPage1.aspx.cs and ActualPage2.aspx

As páginas ActualPage1 e ActualPage2 são as classes code-behind das respectivas páginas e ambas derivam da classe BasePage para garantir cabeçalho é preenchido em ambas.

```
...
public class ActualPage1 : BasePage
{
        protected System.Web.UI.WebControls.Label pageNumber;
        protected override void PageLoadEvent(object sender, System.EventArgs e)
        {
                pageNumber.Text = "1";
        }
}
...
public class ActualPage2 : BasePage
{
        protected Label pageNumber;
        protected override void PageLoadEvent(object sender, System.EventArgs e)
        {
                pageNumber.Text = "2";
        }
}
```

4.4. Application Controller

Um segundo uso da palavra controlador é o padrão Application Controller que é diferente do controlador do padrão MVC.

Este padrão consiste numa camada que separa os objectos de apresentação dos objectos do domínio. O objectivo deste padrão é controlar o fluxo de uma aplicação, decidindo que ecrãs devem aparecer e em que ordem.

Esta camada pode aparecer como parte da camada de apresentação ou como uma camada separada que media entre as camadas de apresentação e domínio.

When To Use It

Este padrão deve ser usado se existirem diferentes apresentações com o mesmo fluxo e navegação; se o sistema tiver muita lógica sobre a ordem dos ecrãs e a navegação entre eles; ou se não existir um mapeamento simples entre as páginas e as acções no domínio. Este padrão não deve ser usado se o utilizador pode ver qualquer ecrã em qualquer ordem.

Deve usar-se um Application Controller quando a aplicação tem o controlo da sequência de ecrãs. Quando é o utilizador que controla este padrão é inútil.

5. Padrões de Concorrência Offline

A concorrência engloba os aspectos mais complicados e enganadores do desenvolvimento de software. Os problemas de concorrência são originados pelo simples facto de haverem vários processos/threads a manipular os mesmos dados. Nestes casos, é sempre muito difícil enumerar os cenários possíveis.

As aplicações empresariais usam muita concorrência mas, em geral, não se preocupam muito com isso pois existem os gestores de transacções.

No seu livro, Fowler diferencia transacções de sistema e transacções de negócio. Transacções de sistema são por exemplo as transacções entre a aplicação e a base de dados (por exemplo, um insert ou um update) enquanto que as transacções de negócio são entre o utilizador final e a aplicação (por exemplo, login, escolha de um produto e compra).

Os padrões de concorrência só devem ser usados se forem realmente necessários, se possível, todas as transacções de negócio devem ser colocadas dentro de transacções de sistema (colocando estas dentro de um pedido http).

Para problemas em que isso não é possível existem alguns padrões, como por exemplo: Optimistic Offline Lock e Pessimistic Offline Lock que servem para controlar a concorrência das transacções de negócio; Coarse-Grained Lock que permite gerir conjuntamente a concorrência de um grupo de objectos; e Implicit Lock que permite evitar a gestão directa de locks.

6. Padrões de Sessões

Neste ponto serão apresentados padrões para guardar o estado das sessões. Existem dois tipos de sessões: sessões com estado (statefull) e sessões sem estado (stateless).

A informação sobre o estado da sessão pode ser guardada de três formas distintas, correspondendo a três padrões.

Foram já vistas no ponto anterior as diferenças entre transacções de negócio e de sistema. Estas diferenças estão associadas à discussão sobre sessões com ou sem estado.

Um servidor sem estado não retém informação de estado entre pedidos. Os objectos nestes servidores podem ter campos, mas quando se invoca um método no servidor, os valores desses campos não estão definidos. A manutenção do estado dos objectos, para muitos autores é uma mudança com um impacto enorme especialmente na utilização de recursos do servidor. Isto é, se não é mantido o estado dos objectos não é importante qual é o objecto que vai atender a um pedido, no entanto, se houver manutenção de estado, terá que ser sempre o mesmo objecto a fazê-lo. Se não há estado, então serão necessários menos objectos para tratar mais utilizadores. Quando mais tempo esses utilizadores estiverem parados, mais partido se tiram os objectos sem estado, pelo que são muito usados em sites web com tráfego intenso. O protocolo HTTP também não tem estado pelo que adere bem ao conceito de objectos sem estado.

Poderia concluir-se que a ausência de estado seria o ideal. Na verdade, tal seria o desejável se fosse possível, no entanto, o problema é que muitas interacções dos clientes são inerentemente com estado. Considere-se o exemplo de um carrinho de compras virtual. Através da sua interacção, o utilizador consulta vários livros e vais seleccionando para o carrinho os que pretende comprar. O carrinho de compras tem que ser mantido para toda a sessão do utilizador. Tal leva a que existe uma transacção de negócio com estado o que implica que a sessão que a suporta tem que ter também estado. Se apenas se fizer consulta de livros mas não se comprar nada, a sessão é sem estado, mas a partir do momento em que se compra alguma coisa, passa a ser com estado. Desta forma, é impossível evitar que as sessões tenham estado para enriquecer um sistema, a questão essencial é definir como o fazer. É possível usar um servidor sem estado para implementar sessões com estado.

Os detalhes de um carrinho de compras são relevantes apenas para uma sessão em particular. O estado mantido é entre uma transacção de negócio, o que implica que é separado das outras sessões e das suas transacções associadas.

6.1.1. Client Session State

Este padrão guarda os dados das sessões no cliente. Existem várias formas de fazer isto: codificando os dados no URL, usando cookies, serializando os dados num campo escondido de um form HTML, ou guardando os dados em objectos num rich-client.

How It Works

Mesmo os sistemas completamente orientados ao servidor têm que ter um pequeno cliente de manutenção de estado de sessão, nem que seja para manter um identificador de sessão. Em algumas aplicações, coloca-se toda a informação de sessão no cliente, e nesse caso é este o responsável por enviar essa informação ao servidor a cada pedido, e este por sua vez devolve essa informação com a

resposta. Este comportamento permite que o servidor seja completamente stateless, ou seja, não mantenha qualquer informação de estado.

Na maior parte das vezes, utiliza-se o padrão Data Transfer Object para tratar da transferência de dados. Este padrão serializa toda a informação na rede permitindo assim que mesmo informação com estrutura complexa possa ter transmitida.

Com uma interface Web, essa tarefa pode ser mais complicada. Existem três formas de implementar o estado da sessão no cliente:

- **Parâmetros no URL** – São a melhor forma se o volume de dados for pequeno. Essencialmente trata-se de incluir no URL um parâmetro com o estado da sessão. O único limite é precisamente o limite que o cliente possa impor para o tamanho de um URL. O seu uso mais comum é utilizá-lo para o identificador da sessão. Pode haver, no entanto, a desvantagem de o cliente fazer bookmark do URL com o ID da sessão, pelo que muitas empresas não o utilizam.
- **Campos escondidos** – Tratam-se de campos que estão numa página web mas que não são apresentados, apesar dos seus valores serem enviados ao servidor. O campo é sucessivamente enviado e retornado do servidor com informação do estado da sessão. Uma forma comum de representa esse estado é com o formato XML porque permite estruturar a informação numa única string. Não é um método muito seguro porque apesar de o conteúdo estar escondido na página web, é fácil ter acesso ao seu código fonte e visualizar a informação.
- **Cookies** – As cookies são ficheiros que são armazenados no cliente e que são automaticamente enviados para o servidor e recebidos deste. Actua um pouco como um campo escondido uma vez que cada cookie contém informação de estado actualizada. A limitação existe no tamanho máximo que uma cookie possa ter. Muitos utilizadores não gostam de os utilizar e desligam essa funcionalidade no browser, o que pode ter como consequência a não utilização de um site que deles dependa. No entanto, com a sua generalização, cada vez menos utilizadores o fazem. As cookies não oferecem mais segurança que os outros métodos e estão limitados a apenas um nome de domínio. Se o site estiver distribuído por vários nomes de domínio as cookies não os atravessarão.

When to Use It

O padrão Client Session State é uma boa opção para suportar servidores sem objectos com estado pois se a aplicação falhar do lado cliente toda a informação é perdida, mas tal não é grave, porque o cliente está à espera disso.

Os argumentos que desaconselham o padrão variam exponencialmente com a quantidade de dados envolvidos. Com uma pequena quantidade de campos tudo funciona bem, no entanto, quando essa quantidade aumenta o custo de transferência torna-se significativo podendo ser até proibitivo a partir de certo ponto. Em ambientes web tal torna-se particularmente crítico.

Levanta-se também a questão da segurança. Os dados transferidos são passíveis de ser observados e alterados. Uma forma de combater essa possibilidade é usar a encriptação, mas decorre sempre daí um aumento de custo de desempenho já que surgem duas tarefas adicionais (encriptação e desencriptação). Se os dados podem ser alterados, então terão que ser sempre re-validados pelo servidor para cada envio.

A recomendação para o uso deste padrão é que trate apenas da identificação da sessão, o que normalmente é um único número. No entanto, a preocupação com a segurança tem que estar sempre presente, já que qualquer utilizador pode alterar o identificador da sua sessão e aceder à de outro. Muitas plataformas geram identificadores aleatórios para reduzir este risco.

6.1.2. Server Session State

Este padrão pode ser tão simples como guardar os dados das sessões em memória entre pedidos HTTP. Normalmente existe um mecanismo para guardar os dados da sessão num local mais durável, como por exemplo, um objecto

serializável. O objecto pode ser guardado no sistema de ficheiros do servidor da aplicação ou pode ser guardado numa fonte de dados partilhada (uma tabela de base de dados com um ID da sessão como chave e o objecto serializado como valor).

How It Works

Na sua forma mais simples, este padrão guarda em memória um objecto de sessão num servidor aplicacional. Pode usar esquemas de mapeamento em memória que armazena estes objectos para cada identificador de sessão. Tudo o que cliente tem que fornecer é o seu identificador de sessão e o objecto ser identificado no mapa e processar o pedido.

Este cenário assume que o servidor aplicacional tem a memória necessária para realizar a tarefa e que existe só um servidor aplicacional, ou seja, não há clusters. Se o servidor aplicacional falhar, todas as sessões têm que ser abortadas.

Para muitas aplicações, estas assunções não constituem realmente um problema, no entanto, para outras pode ser catastrófico. Há formas de lidar com esses casos que introduzem sempre variações que introduzem complexidade ao que seria inicialmente um padrão simples.

A primeira questão é como lidar com os recursos de memória utilizados pelos objectos de sessão, sendo este o ponto mais controverso sobre a utilização do padrão Server Session State. A resposta assenta em não ter os recursos em memória e em vez disso usar o padrão Memento (Gang of Four) para armazenamento persistente. Tal apresenta duas questões: de que forma se efectua a persistência do estado das sessões e para onde deve ser efectuada a persistência. Muitas plataformas incluem mecanismos de serialização em binários que permitem serializar objecto de forma simples. Esta forma tem a vantagem de ocupar menos espaço em disco (embora hoje em dia isso seja menos significativo). Outra forma é serializar um texto, sendo o formato mais recomendado o ficheiro XML. Quanto ao local onde se persistem os objectos, surgem duas possibilidades: ou no sistema de ficheiros do servidor ou numa base de dados local. A última opção implicaria uma tabela cuja chave é o identificador da sessão o uma coluna Large Object (usando o padrão Serialized LOB) para armazenar a informação de estado da sessão. O desempenho das bases de dados varia muito no que diz respeito à forma como lidam com Large Objects.

A diferença entre o padrão Server Session State e o Database Sesseion State parece não existir se o primeiro usar bases de dados. De facto a diferença não é facilmente perceptível mas assenta no ponto em que o Server Session State converte a informação de sessão para a forma tabular. Se o padrão Server Session State é usado em conjunção com uma base de dados, é necessário lidar com a possibilidade de as sessões desaparecerem. Uma forma será instalar processos deamon que procuram sessões antigas e as terminam. Esta abordagem pode levar a muita contenção na tabela de sessões, pelo que esta deve ser particionada.

When to Use It

A grande vantagem do padrão Server Session State é a sua simplicidade. Na maioria dos casos não é necessário efectuar qualquer programação para o pôr em prática, uma vez que muitos dos mecanismos já vêm incluídos nos servidores aplicacionais. Mesmo a serialização e inserção de um LargeOBject numa base de dados é uma tarefa simples. A maior potencial esforço de programação surge na manutenção da sessão, principalmente se envolver clustering e recuperação de sessões.

6.1.3. Database Session State

Este padrão define também um armazenamento da sessão do lado do servidor mas, neste caso, envolve dividir os dados da sessão em tabelas e campos e guardá-los na base de dados tal como se guardam dados mais duráveis.

How It Works

Quando é feito um pedido do cliente para o servidor, o objecto que trata esse pedido obtém os dados da base de dados necessários para atender ao pedido. De seguida, efectua o processamento necessário e depois guarda novamente os dados na base de dados. Para obter a informação da base de dados o objecto no servidor precisa de informação sobre a sessão, o que no caso mais simples consiste no identificador de sessão mantido pelo cliente. Esta informação não é mais do que a chave que identifica os dados necessários na base de dados.

A informação envolvida é tipicamente uma mistura de dados da sessão específicos da interacção em causa e dados cometidos que são relevantes para todas as interacções.

Um dos pontos-chave a considerar é o facto de que a informação de sessão é considerada local à sessão e não deve afectar outras partes do sistema até que a sessão como um todo seja cometida. Assim, se se estiver a trabalhar numa encomenda numa sessão e se pretender guardar o seu estado intermédio na base de dados, tal terá que ser tratado de forma diferente do que o é quando a encomenda é confirmada no fim da sessão. O motivo para tal é que não se pretende que as encomendas pendentes surjam com as confirmadas, podendo afectar as estimativas da disponibilidade dos produtos que incluam.

Surge então a questão de como separar estes diferentes tipos de dados de sessão. Uma possibilidade é adicionar uma coluna boleana à tabela de sessões que apenas indica se a encomenda é pendente ou não. Outra abordagem é ter o ID da sessão na encomenda, o que facilita a pesquisa de todos os dados associados a uma sessão em particular. Daí advém também a vantagem de ser muito fácil distinguir as encomendas de sessões pendentes (tem a sessão ID preenchida) das já finalizadas (têm a sessão ID nula).

Utilizar o identificador da sessão é uma solução intrusiva porque todas as aplicações que precisem de consultar a base de dados têm de conhecer o significado dessa coluna para impedir que consultem informação de sessão. Podem utilizar-se vistas para acesso à tabela de forma a ocultar informação mas muitas vezes impõe custos no acesso à base de dados.

Uma segunda abordagem é utilizar um conjunto de tabelas separado. Se existem encomendas e linhas de encomenda na base de dados, criam-se tabelas para encomendas pendentes e as duas linhas correspondentes. Desta forma, guarda-se informação de encomendas pendentes nas tabelas encomendas pendentes e informação de encomendas confirmadas na tabela de encomendas. Esta abordagem elimina muita da intrusão da solução anterior, no entanto, duplica as tabelas envolvidas e cria lógica de selecção de tabelas para cada caso.

A informação “final” tem regras de integridade e validação que não precisam de ser aplicadas à informação pendente. As tabelas pendentes têm que se clones das tabelas reais para manter a lógica de mapeamento o mais simples possível, implicando usar os mesmos nomes para as colunas e tamanhos. A única diferença é que as tabelas pendentes têm uma coluna adicional de identificador de sessão.

É necessário também implementar um mecanismo para limpeza de sessões que tenham sido canceladas ou abandonadas. Se for conhecido o identificador de sessão a sua remoção é simples, caso contrário, utiliza-se um mecanismo de timeout que elimine sessões ao fim de um certo tempo. Para tal é necessário incluir um campo adicional que indique quando foi criada a sessão.

Se forem utilizadas tabelas pendentes que só serão utilizadas pelos objectos que tratam das sessões, pode não fazer sentido ter esses dados em tabelas. Será preferível utilizar um Serialized LOB, o que transforma toda a abordagem num Server Session State.

When to Use It

Database Session State é apenas mais uma alternativa para lidar com informação de estado de uma sessão, pelo que deve ser sempre comparado com a Server Session State e a Client Session State.

O principal aspecto a analisar neste padrão é o desempenho. Há ganhos em utilizar objectos sem estado no servidor, permitindo efectuar clustering. O compromisso é o esforço e tempo necessários para obter a informação da base de dados para cada pedido. Este custo pode ser reduzido utilizando caches nos objectos do servidor de forma a não ser necessário ler sempre todos os dados da base de dados. As operações de escrita, no entanto, terão que ser sempre efectuadas.

6.1.4. Escolher o padrão para as sessões

Existem bastantes questões relativamente à escolha do padrão para armazenamento dos dados das sessões.

Quanto às necessidades de largura de banda entre o cliente e o servidor, a utilização do padrão Client Session State implica a passagem dos dados da sessão juntamente com todos os pedidos HTTP. A não ser que os dados a guardar da sessão sejam muito reduzidos, o padrão Client Session State não deve ser usado.

Quanto à segurança e integridade, usando o padrão Client Session State, a não ser que os dados da sessão sejam encriptados, deve ser assumido que qualquer utilizador poderá editar os dados da sessão guardados no seu computador.

Um dos problemas comuns em muitos sistemas são as sessões-fantasma que deixam de ter actividade. O padrão Client Session State oferece o melhor tratamento, porque é o cliente quem mantém a informação de sessão, o servidor apenas atende a pedidos. Nas outras abordagens, é necessário implementar mecanismos de limpeza de sessões quando se conclui que foram canceladas por qualquer motivo.

Da mesma forma que o utilizador pode cancelar sessões, por vezes é o servidor que o faz, por exemplo, por falha de rede. O padrão Database Session State lida bem com esta situação uma vez que é o servidor que retém toda a informação da sessão na base de dados. Com o padrão Server Session State a informação pode ou não continuar a existir dependendo de se usar um mecanismo não volátil para armazenar a informação da sessão.

Em termos de esforço de desenvolvimento, o padrão Server Session State é o menos exigente. Os outros dois padrões apresentados envolvem normalmente a utilização de código para transformar os dados da base de dados para o formato que os objectos de sessão tratam. Este tempo adicional implica que não seja possível ter tantas características tão rapidamente como se consegue com o Server Session State.

As três abordagens não são mutuamente exclusivas. É vulgar encontrarem-se as três em partes diferentes da manutenção do estado da sessão. Esta abordagem pode tornar a estrutura mais complicada, quando se torna difícil perceber que parte do estado está em que parte do sistema. Será sempre necessário ter o padrão Client Session State nem que seja só para manter o identificador da sessão mesmo que todo o resto do sistema utilize qualquer dos outros padrões.

7. Padrões de Distribuição (interfaces remotas/locais)

No contexto da distribuição existem também vários padrões dos quais apresentamos apenas dois relacionados com a problemática das interfaces remotas/locais.

A interface de um objecto que vai ser usado remotamente deve ser diferente da interface de um objecto que vai ser usado localmente no mesmo processo.

Uma interface local deve ser de granularidade fina. Por exemplo, para uma classe Endereço, uma boa interface deverá conter métodos separados para obter a cidade, o estado, etc. Uma interface de granularidade fina é boa porque segue o princípio geral OO de muitas pequenas peças que podem ser combinadas e reescritas de vários formas para estender o desenho para o futuro.

Uma interface de granularidade fina não funciona bem quando é usada remotamente. Quando as chamadas a métodos são lentas a obtenção ou alteração dos atributos de uma classe deve ser feita numa só chamada e não em várias. Uma interface que permite tal é uma interface de granularidade grossa desenhada não para a flexibilidade e extensibilidade mas para a minimização de chamadas. Aqui veríamos uma interface do género *get endereço* e *update endereço*.

7.1. Remote Facade

A solução para estes problemas de distribuição é usar as interfaces de granularidade fina internamente e colocar interfaces de granularidade grossa nas fronteiras de distribuição, cujo único papel é fornecer uma interface remota para um objecto de granularidade fina. Os objectos de granularidade grossa não fazem nada a não ser agirem como uma fachada para os objectos de granularidade fina. O nome Remote Facade vem do facto de esta fachada existir apenas por motivos de distribuição.

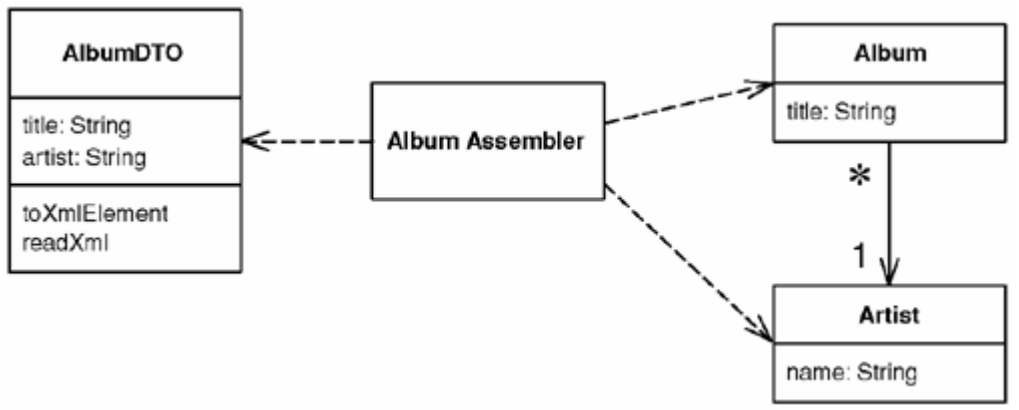
Usar o padrão Remote Facade ajuda a minimizar as dificuldades que as interfaces de granularidade grossa introduzem.

Desta forma apenas objectos que realmente necessitam de serviços remotos acedem aos métodos de granularidade grossa e fica óbvio para os programadores que estão a pagar esse custo.

7.2. Data Tranfer Object

O contexto deste padrão é a transferência de objectos de granularidade grossa.

O Data Transfer Object pode ser definido como um objecto que “guarda” os dados entre processos para reduzir o número de chamadas aos métodos.



Quando se trabalha com chamadas remotas, cada uma pode tornar-se demasiado "cara". É necessário reduzir o número de chamadas o que significa que tem que se transmitir maior quantidade de dados em cada chamada. Uma das formas de o fazer é utilizar grandes quantidades de parâmetros, o que por vezes se torna impossível de fazer em linguagens como o Java pois só retornam valores simples.

A solução passa por criar um Data Transfer Object (DTO) que pode guardar os dados e os pode enviar, serializados, de uma só vez através de uma ligação única.

How It Works

Uma das grandes vantagens na utilização de um DTO passa pela possibilidade de podermos transferir grandes quantidades de informação num só pedido. Um DTO transfere sempre mais dados do que o objecto que o pediu estava à espera. Contudo, é sempre preferível enviar mais do que o necessário em vez de efectuar múltiplos pedidos. É o caso, por exemplo, de um pedido de informações sobre uma determinada compra de um cliente. O DTO irá retornar mais informação do que a necessária, pois na base da sua construção foram considerados todos os parâmetros possíveis. Os tipos de campos transferidos são, regra geral, tipos simples pois são serializados para que possam ser entendidos dos dois lados, para quem pede e para quem envia.

Um DTO é desenhado para um determinado pedido. Daí que muitos DTO respondam exactamente ao que um determinado objecto necessite. Contudo, se diferentes tipos de apresentações requerem dados muito similares, é natural que num só DTO se insira toda essa informação. O critério para desenhar um DTO deve ficar à consideração e necessidade de cada um. Usar o mesmo para trocar informação de um lado para o outro deve ser uma decisão conforme as necessidades de cada objecto.

Um modelo típico de um DTO é um RecordSet, ou seja, um conjunto de registos tabelados, que é exactamente aquilo que se obtém num pedido à base de dados. Isto pode ser bastante útil se o cliente tiver a possibilidade de lidar com a estrutura de um RecordSet. As colecções de dados estruturadas são outra forma de DTO. Há quem utilize arrays, o que não é muito prático pois os índices do array tornam o código mais difícil de ser interpretado. A melhor colecção de dados, é um dicionário pois podemos atribuir às keys strings que permitem uma leitura mais fácil e compreensiva.

Serializar o Data Transfer Object

Além dos gets e sets, um DTO tem também a responsabilidade pela serialização dos dados num formato que possa circular na rede: qual o tipo de formato suportado em cada um dos lados da rede; o que pode circular na ligação. Algumas plataformas como o Java e o .NET já contém objectos específicos para a serialização de objectos. Contudo, este tido de objectos de serialização podem ser construídos de forma simples para linguagens especificas. Alguns cuidados devem

ser tidos em consideração, como por exemplo que ambos os lados da ligação consigam serializar e des-serializar.

Uma das grandes questões quanto à serialização de objectos, prende-se com a utilização do tipo de dados a utilizar para a serialização: texto ou binário? A serialização em texto é mais fácil de ler. O XML é um exemplo disso, mas também tem as suas desvantagens como é o caso de necessitar maior largura de banda penalizando a performance da ligação.

Outro factor a ter em conta na serialização de objectos, prende-se com a sincronização de dados em ambos os lados da ligação. Em teoria, sempre que o servidor altera a definição do DTO o cliente também altera essa estrutura, mas na prática isso nem sempre acontece. A Serialização em XML pode muitas vezes facilitar este tipo de dessincronizações embora a serialização com o apoio de um dicionário também ajude no tratamento deste tipo de erros.

When to Use It

Um DTO deve ser utilizado sempre que seja necessário trocar grandes quantidades de dados através de uma ligação. Existem alternativas aos DTO embora não sejam muito aconselhadas, como é o caso da utilização de uma string onde se represente os valores sem ter a interacção de um objecto. Contudo, é sempre preferível usar um DTO quando se lidar com componentes que utilizem XML.

8. Conclusões

8.1. Resumo

Ao longo deste documento foi possível analisar vários padrões para aplicações empresariais, desde padrões de implementação para as três camadas base como padrões mais específicos para a gestão das sessões.

Camada da Lógica de Negócio

Quanto à camada da lógica de negócio a opção é entre os três padrões apresentados: Transaction Script, Table Module e Domain Model. No ponto 2.4 foi feita uma análise dos três.

Camada da Fonte de Dados

Quanto à camada da fonte de dados, as opções dependem da opção da camada anterior.

Se usarmos o padrão Transaction Script podemos optar pelos padrões Row Data Gateway ou Table Data Gateway (talvez recorrendo a Record Sets ou não).

Se usarmos o padrão Table Module podemos usar o padrão Table Data Gateway com recurso aos Result Sets.

Com o padrão Domain Model a opção é entre vários padrões: Active Record, Row Data Gateway, Table Data Gateway e Data Mapper. Neste caso podemos usar todos os outros padrões de mapeamento objecto-relacional, por exemplo, o Association Table Mapping, Foreign Key Mapping, etc.

Em qualquer um deles poder-se-á usar os padrões Unit of Work e o Optimistic Offline Lock.

Camada de Apresentação

Nesta camada aconselha-se o uso do padrão MVC em todas as situações.

Para a vista foram apresentados os padrões Template View e Transform View. Ainda na vista aconselhou-se a utilização do Two Step View para casos em que se apresenta uma interface comum mas com variantes diferentes em termos de *look and feel*.

Para o controlador foram apresentados os padrões Page Controller e Front Controller.

Distribuição

Em termos de distribuição oram apresentados sucintamente os padrões Remote Facade e Data Transfer Object.

Sessões

No ponto 6 foram apresentados 3 formas de lidar com o problema das sessões com estado.

À medida que os padrões foram sendo apresentados foram também surgindo exemplos nas plataformas J2EE e .NET. Foram apresentadas também algumas comparações entre as plataformas.

8.2. Considerações Finais

Neste estudo foi feita uma visita geral a todos os padrões e, consoante o interesse encontrado e tempo disponível, foi aprofundado o estudo de cada padrão (ou alternativas de padrões). O estudo passou por tentar identificar a implementação (ou outro nome do padrão) na bibliografia associada a cada plataforma e tentar documentar a especificidade das implementações de cada padrão nas plataformas.

Este método de documentação de padrões mostrou-se extremamente eficaz para entrar em contacto com as plataformas dado que não só se absorvem as características específicas de cada plataforma como se aprende as "soluções comprovadamente boas". Dos padrões apresentados por Fowler todos são aplicáveis em ambas as plataformas estudadas.

De facto, a quantidade de padrões documentados na bibliografia é muito elevada e o estudo teve de se cingir aos mais importantes. Não foi mantido o nível de profundidade de estudo de cada padrão para que em alguns se pudesse mostrar exemplos e diferenças entre plataformas e noutros apenas mostrar descrições gerais dos padrões.

Podemos concluir que o conhecimento destes padrões é, de facto, uma mais valia para o desenvolvimento de aplicações empresariais.

Trabalho Futuro

Como trabalho futuro apresentamos a possibilidade de aprofundamento do estudo feito até agora, tanto a nível de número de padrões (especialmente os mais específicos de cada plataforma) como a nível de profundidade das comparações efectuadas.

Outra hipótese de continuação do trabalho poderia ser o seguimento do objectivo inicial do trabalho (posteriormente alterado) de construir uma aplicação única demonstrativa, nas duas plataformas, que permitisse demonstrar mais facilmente as diferenças entre as plataformas.

Outra sugestão seria o alargamento da comparação a outras plataformas.

9. Referências

[1] Patterns Of Enterprise Application Architecture, Martin Fowler, 2002

[2] Enterprise Solution Patterns using Microsoft.NET version 2, Patterns and Practices, Microsoft Corporation, 2003

[3] Oracle Application Server - Application Developer's Guide 10gr2, Oracle, 2004

[4] http://java.sun.com/blueprints/corej2eepatterns/

[5] http://java.sun.com/blueprints/patterns/catalog.html

[6] http://danagonistes.blogspot.com/2004/11/organizing-domain-logic.html

[7] http://www.builderau.com.au/architect/sdi/0,39024602,20281590,00.htm

[8].NET Patterns Architecture Design and Process, Microsoft Corporation, 2003

[9] Design Patterns - Elements of Reusable Object-Oriented Software, Erich Gamma et al, 1995, ISBN 0-201-63361-2.

[10] A Pattern Language: Towns, Buildings, Construction, Christopher Alexander, 1977