Universidade do Minho
Escola de Engenharia

Manuel José Torres Sousa da Cunha

Metodologias para Abstracção de *stored procedures* em Aplicações *web* Orientadas à Acção

Outubro de 2009

Metodologias para Abstracção de *stored procedures* em Aplicações *web* Orientadas à Acção
Manuel José Torres Sousa da Cunha
UMinho | 2009

Universidade do Minho
Escola de Engenharia

Manuel José Torres Sousa da Cunha

Metodologias para Abstracção de *stored procedures* em Aplicações *web* Orientadas à Acção

Tese de Mestrado em Informática

Trabalho efectuado sob a orientação do
Professor Doutor Victor Alves

Outubro de 2009

Aos meus pais e à minha irmã

“Deus não joga aos dados com o universo.”

Albert Einstein

Agradecimentos

É impossível expressar o meu agradecimento a todos os que estiveram comigo nesta caminhada, por vezes difícil, mas muitas vezes tornada mais fácil pela ajuda dessas pessoas. Em primeiro lugar, gostaria de expressar a minha profunda gratidão, o meu profundo agradecimento e todo o sentimento que lhes nutro, à minha família, os meus pais e à minha irmã Estefânia, sem eles este documento nunca seria produzido. À Sandra, pela enorme paciência. Sem o devido apoio de alguém do Departamento de Informática ser-me-ia mais difícil adquirir todos os novos conceitos e novas tecnologias abordadas durante a duração deste trabalho, assim agradeço ao Professor Victor Alves a disponibilidade total em me auxiliar sempre que necessitei e me dar sempre uma perspectiva fresca sobre o assunto e me incentivou a atingir mais e melhores resultados. Não posso esquecer quem me fez ver novas perspectivas, assim endereço o meu agradecimento a toda a equipa do projecto onde me encontro inserido, especialmente ao meu supervisor na empresa, Pedro Viterbo, que se mostrou sempre disponível para ajudar, ao Hugo, ao Ricardo, ao André, à Susana, ao Eduardo, à Cláudia, ao Rocha e ao meu gestor de projecto, Engenheiro Paulo Ferro que sempre viu este trabalho com aplicação prática na engenharia de *software*, todos eles, sem excepção estiveram presentes com a sua análise crítica. Ao Engenheiro Rui Lopes por ter acreditado em mim e me ter dado a oportunidade e apoio necessários para atingir este objectivo e à PontoC pelo espírito inovador, jovem, que incute aos seus colaboradores. Por último, mas não em último, expresso o meu agradecimento a todos os meus amigos que estiveram sempre presentes e àqueles que não foram mencionados, mas que estão presentes no meu pensamento.

Resumo

A persistência dos dados e a forma como um servidor *web* comunica com um Sistema de Gestão de Bases de Dados pode ser um problema se não for devidamente arquitectado e estruturado. O desenvolvimento de aplicações *web* não é recente e o seu contínuo crescimento e evolução obriga a que a informação seja disponibilizada o mais rápida e eficientemente possível, fazendo com que os seus processos de desenvolvimento tenham ciclos cada vez mais curtos e, consecutivamente, os programadores devem responder a esses processos de uma forma mais eficiente e rápida. A forma de armazenar os dados pode ser feita de várias formas. Uma das quais pode ser o recurso a *stored procedures* para colocar, de forma persistente, a informação numa base de dados. Habitualmente o recurso a esta metodologia implica a codificação de um objecto equivalente no servidor aplicacional que servirá de ponte entre a aplicação *web* e a base de dados. Esses objectos são, naturalmente, referenciados como *Data Acces Objects* (DAO) e a sua abstracção é objecto de análise neste trabalho. Estes objectos recorrem, normalmente, a objectos Java muito simples (POJO) como sendo os objectos de transferência de dados. É feita uma análise à possibilidade de usar objectos JSON como complemento dos POJO para a transferência de dados entre uma base de dados e um servidor aplicacional J2EE. Simultaneamente definem-se metodologias de *cache* que possibilitem rápidas alterações na aplicação *web* sem ter de se recorrer ao habitual *deploy* da aplicação, respondendo também mais rapidamente a novos requisitos propostos pelo cliente. Arquitectando uma camada de persistência que sirva de intermediário entre o servidor aplicacional e a base de dados aumenta-se o desempenho dos métodos de desenvolvimento *web*, assim como a própria eficiência da aplicação, tornado-a mais robusta, mais portável e fazendo com que os programadores se abstraiam das especificações de baixo-nível existentes na comunicação e obtenção de dados entre o servidor e a base de dados.

Palavras-chave: JAVA Web Framework, Persistence Layer, JSON, AJAX, Webwork, JSP, Design Patterns, DAO, MVC, Stored Procedures

Abstract

The data persistence and how a web server communicates with a Database Management System can be a problem if not properly engineered and structured. The development of web applications is not new and its continued growth and evolution requires that the information is made available as quickly and efficiently as possible, making their development process cycles shorter and, thereafter, developers must meet these processes more efficiently and quickly. The way of storing data can be done in several ways, one of which may be the use of stored procedures to place, persistently, information in a database. Usually the use of this methodology involves the encoding of an object equivalent in a application server that will serve as a bridge between the web application and a database. These objects are, naturaly, referred to as Data Acces Objects (DAO) and its abstraction is examined in this work. These objects use usually the simplest Java objects (POJO) as objects of data transfer. We analyze the possibility of using JSON objects in addition to the POJO as the transfer of data between a database and a J2EE application server. At the same time we define cache methodologies that enable systems rapid changes in the web application without having to resort to the usual application deployment, and responds quickly to new requirements proposed by the client. Hatched a persistence layer that serves as an intermediary between the application server and database increases the performance of the methods of web development, as well as the very efficiency of the application, making it more robust, more portable and making developers to disregard the specifications of existing low-level communication and data collection between the server and database.

Keywords: JAVA Web Framework, Persistence Layer, JSON, AJAX, Webwork, JSP, Design Patterns, DAO, MVC, Stored Procedures

Índice

Índice de Figuras

Índice de Tabelas

Notação e Terminologia

Notação Geral

Ao longo de todo o documento é utilizado *itálico*, para palavras apresentadas em língua estrangeira, equações e fórmulas matemáticas. O texto utilizado com cor exemplifica extractos de código utilizado. De referir ainda a existência de termos que dado a sua universalidade não foram traduzidos.

Acrónimos

API	*Application Programming Interface*
AJAX	***A**synchronous **J**avaScript **A**nd **X**ML*
ASF	*Apache Software Foundation*
BD	*Base de Dados*
CGI	*Common Gateway Interface*
CSS	*Cascading Style Sheets*
DAO	*Data Access Object*
DTD	*Document Type Definitions*
EL	*Expression Language*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
IDE	*Integrated Development Environment*
Java EE	*Java Enterprise Edition*
JAR	*Java ARchive*
JSP	*JavaServer Pages*

JSF	*JavaServer Faces*
jwcid	*Java Web Component ID*
MVC	*Model-View-Controller*
OGNL	*Object-Graph Navigation Language*
POJO	*Plain Old Java Object*
POO	*Programação Orientada aos Objectos*
SI	*Sistemas de Informação*
SGBD	*Sistema de Gestão de Bases de Dados*
TI	*Tecnologias da Informação*
UEL	*Unified Expression Language*
UI	*User Interface*
URL	*Uniform Resource Locator*
WW	*Webwork*
WWW	*World Wide Web*
XML	*e**X**tensible **M**arkup **L**anguage*

Capítulo 1

Introdução

Ideias chave:
- Aplicações *web*;
- MVC;
- Persistência de dados;
- Frameworks orientadas à acção;
- *Stored Procedures;*
- Problema no desenvolvimento de aplicações *web* sem camada de persistência.

A Informática encontra-se, constantemente, numa evolução tecnológica, sendo um aliciante para melhorar a qualidade de vida, aumentar a produtividade laboral, proporcionar novos desafios. No domínio da web esta evolução também se verifica. Hoje, mais que nunca, as aplicações neste domínio chegam a um elevado número de utilizadores e a forma de interacção é crucial para uma melhor compreensão da aplicação, assim como facilitar o seu uso. O desenvolvimento de interfaces gráficos para o utilizador é, portanto, um factor preponderante quando se produz um produto em larga escala, sendo a parte visual a forma do utilizador usar a aplicação. No desenvolvimento de uma aplicação para web têm de ser consideradas várias possibilidades, desde a escolha da base de dados, passando pela lógica de negócio que lhe é inerente e chegando à parte visual. Desde a camada inferior até à camada superior existem ferramentas que tornam cada nível mais transparente, mais modular. No desenvolvimento da interface gráfica com o utilizador torna-se relevante a escolha de uma ferramenta que permita a uma equipa potenciar a sua produção num

determinado intervalo de tempo referente a um projecto, com uma curva de aprendizagem baixa, esperando uma boa resposta da comunidade (em caso de dúvidas) não esquecendo também a potencial evolução tecnológica que essa ferramenta possa vir a surtir. Deste modo, a escolha de uma ferramenta que nos permita atingir estes objectivos revela-se importante, tanto no projecto actual, como nos projectos futuros.

A forma como se obtêm os dados a partir de uma base de dados e se colocam numa página web é algo que não pode ser deixado de parte, e o desenvolvimento de tal interacção pode ser feita de diversas formas, recorrendo a ferramentas que garantam a fiel comunicação o entre a base de dados e a aplicação web propriamente dita. Esta comunicação tem de proporcionar, a quem desenvolve, uma facilidade de compreensão, não havendo a necessidade de obter conhecimentos tecnológicos além dos já aprendidos, e devendo ser transparente em relação a outras tecnologias paralelas no mundo da web. É de salientar que neste universo, estas ferramentas, conhecidas por *frameworks*, são criadas, à partida, de acordo com as necessidades específicas de um determinado projecto, sendo, posteriormente, difundidas através de uma comunidade que lhe dá o suporte necessário e que a mantém e evolui. Torna-se então fácil perceber porque foram desenvolvidas inúmeras *frameworks* orientadas a interfaces com o utilizador, assim como para a comunicação entre uma base de dados e uma aplicação *web*.

1.1 Desenvolvimento *web*

Desenvolver uma aplicação empresarial de grande porte não é uma tarefa fácil, nem rápida. Através da devida estruturação da mesma é possível tornar essa tarefa menos lenta e menos complicada. Recorrendo a fórmulas já acreditadas como verdadeiras soluções para diversos problemas podemos melhorar ainda mais esse desenvolvimento, tornando-o mais eficiente, maximizando os lucros de colocar a aplicação no mercado dentro do prazo previsto, minimizando os custos do processo de criação e até da sua manutenção. Considerando que muitos dos projectos de *software* não chegam a ser concluídos, torna-se necessário melhorar os processos de desenvolvimento de um produto,

desde a sua concepção, passando pela análise de requisitos, a codificação, os testes e a instalação final do produto [DSI, 2007]. Já diz o velho ditado: "tempo é dinheiro" e, como tal, todas as formas que permitam a redução do tempo de desenvolvimento são potenciadoras de gerar riqueza.

Olhando para uma perspectiva mais conceptual percebe-se que as primeiras fases de um projecto são as mais importantes: qual o sistema de armazenamento a usar, qual a forma de transacção de dados pretendida, como aplicar as regras de negócio, entre outros. Mas a forma como se irão abordar determinados problemas na implementação do produto é também um factor preponderante durante as várias fases de um projecto. Como já muitos programadores, engenheiros e arquitectos de software já se depararam com diversos problemas, naturalmente começaram a surgir propostas de soluções sobre a forma como abordar esses problemas. Através da identificação de padrões torna-se possível inferir um determinado padrão como sendo capaz de resolver vários problemas que se encontrem nesse padrão de solução. [Gamma et al, 1994] No mundo do desenvolvimento de software, recorrendo ao paradigma orientado a objectos [Martins, 2000] , estes padrões, quando agrupados, podem definir uma arquitectura padrão capaz de resolver um determinado problema. Um desses problemas foi qual a arquitectura mais adequada para o desenvolvimento de aplicações *web*. No universo das aplicações *web* desenvolvidas através da linguagem de programação *Java* (podendo mesmo ser possível considerar no âmbito das aplicações *web* orientadas ao objecto) uma das arquitecturas mais usadas no dia-a-dia de programadores é o *Model-View-Controller* (MVC) (Figura 1).

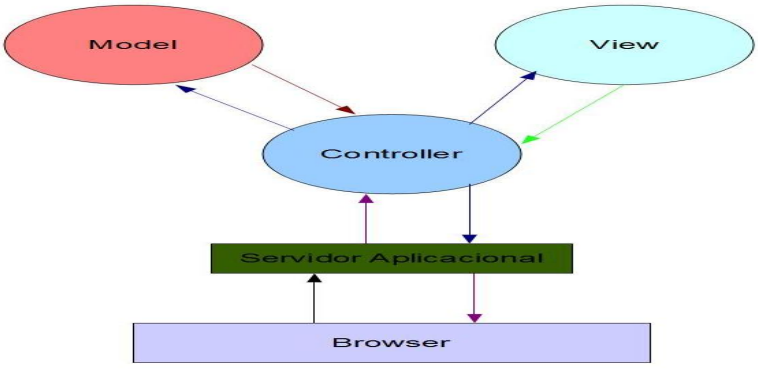

Figura 1: MVC e a *web*

Esta arquitectura permite a separação de conceitos, assim como também a organização do desenvolvimento através de equipas distintas, potenciando todo o processo de construção de um produto. Uma das vantagens de uma aplicação *web*, contrapondo com as aplicações "desktop", é a sua instalação (*deploy*). A instalação de uma aplicação *web* é, de um modo geral, a configuração dos componentes no servidor aplicacional, não sendo necessária, também de um modo geral, nenhuma configuração, nem *software* especial do lado do cliente, a não ser o *browser.* A natureza da comunicação entre uma aplicação *web* e um servidor é um dos pontos principais no seu sucesso. O *HyperText Transfer Protocol* (HTTP) é o principal protocolo de comunicação nas aplicações *web*, que devido à inexistência de estado nas conexões torna-se muito robusto e com tolerança a faltas. Não pensemos que uma aplicação *web* anda em volta da comunicação entre o cliente o servidor, mas sim através da navegação de páginas *web*. Podemos considerar, até um certo nível de abstracção, que toda comunicação numa aplicação *web* é descrita como o pedido e a recepção de entidades de uma página *web*. [Conallen, 1999]

Muitas aplicações desenvolvidas recorrem a um *Uniform Resource Locator* (URL) para criar estados na aplicação, através de parâmetros enviados pelo cliente ao servidor. Esses parâmetros podem ser introduzidos pelo utilizador (num formulário de inserção/alteração de dados) ou mesmo pré-programados para serem enviados em determinada altura.

Para esse tipo de desenvolvimento o programador pode recorrer a várias estratégias como a colocação de campos invisíveis ao utilizador que servem para controlar as interacções do cliente, a validação dos dados inseridos/alterados feita apenas do lado do cliente (comum para verificar se um determinado campo está correctamente preenchido, como a inserção de texto em campos que são de índole numérica). Deste modo e devido à grande proliferação do desenvolvimento orientado à *web* foram surgindo também soluções capazes de abstrair determinados componentes do processo de construção da aplicação. Assim apareceram estruturas "prontas-a-usar" (*ready-to-use*) para as diversas áreas associadas à aplicação, normalmente essas estruturas são designadas de *frameworks*. Desde o tratamento de eventos no cliente, passando pela abstracção do tratamento

dos pedidos/repostas efectuados, a geração de componentes *HyperText Markup Language* (HTML) e chegando à forma como os dados são armazenados, existentes estruturas capazes de abstrair as várias implementações.

Ao longo deste trabalho é feita uma análise a algumas das estruturas e também a apresentação de implementações existentes para essas estruturas.

De acordo com os pressupostos deste trabalho, é feita a análise de arquitecturas capazes de abstrair a obtenção de dados de uma base de dados (BD), a persistência de dados.

1.2 Persistência de dados

A *World Wide Web* (WWW) tem tido um largo crescimento desde o seu aparecimento e desde o início que o desenvolvimento de aplicações para web tem sofrido uma constante evolução e alteração. Nos primeiros tempos os conteúdos eram estáticos, ou seja, não havia alteração dos mesmos em tempo de execução. A evolução de novas tecnologias permitiu o surgimento de conteúdos dinâmicos, de aplicações mais robustas e aproximou o utilizador de aplicações em *web* para um universo onde essas mesmas aplicações se assemelham às desenvolvidas para *desktop*, sendo esse um dos objectivos finais de quem apresenta um produto para *web*. Através da devida estruturação de uma aplicação é possível que esta seja desenvolvida e, posteriormente, mantida por várias equipas. Recorrendo a determinadas arquitecturas é possível dividir as tarefas entre quem ficará responsável por implementar o aspecto gráfico da aplicação, quem ficará responsável pela persistência dos dados, quem fará o desenvolvimento dos interfaces gráficos com o utilizador e como irá apresentar a informação ao mesmo.

Com a constante evolução de tecnologias, o exigente mercado de trabalho e a necessidade de produzir mais em menos tempo, surgem ferramentas para resolver os diversos problemas associados ao desenvolvimento *web*. Essas ferramentas são, normalmente, designadas de *frameworks*, uma estrutura que tem soluções prontas a usar para um determinado problema.

As aplicações *web* recorrem normalmente ao HTTP [HTTP, 1999], e como se trata de um protocolo sem estado [Fielding et al, 1999] muitos programadores optam por criar a noção de estado através do recurso aos parâmetros enviados pelo cliente ao servidor, conseguindo, desta forma, saber em que ponto da aplicação se encontra o utilizador em determinado espectro temporal.

Podemos ver o desenvolvimento de aplicações *web* como uma actividade multi-facetada, não sendo apenas exclusiva da componente mais técnica, mas incluindo também questões organizativas, de gestão e até mesmo de cariz mais social e artístico. [Fraternali, 1999]

1.3 Enquadramento

Normalmente as decisões das tecnologias a usar em determinado projecto parte de decisões ainda no âmbito da arquitectura do sistema a criar. O mestrando elaborou a sua dissertação em ambiente empresarial e, como tal, foi observando algumas das formas de programar/desenvolver colocadas em prática diariamente. Essas metodologias não são objecto de análise deste trabalho, mas são consideradas na generalização da metodologia apresentada. Assim deve ser apresentado o enquadramento deste trabalho de uma forma mais universal, possibilitando que situações idênticas possa ser resolvidas através do mesmo método apresentado.

Para o desenvolvimento é considerado como método o modelo em espiral [Boehm, 1988], onde se verifica a necessidade de fazer várias iterações até chegar ao produto final. Estas iterações são repartidas em quatro fases (Figura 2): “Analisar Riscos e Planear” , “Analisar Requisitos”, “Especificar, Implementar, Testar” e “Avaliar”. Assim, existem diversos ciclos de iteração sobre as quatro fases, levando ao refinamento do produto desenvolvimento até atingir um grau de satisfação junto do cliente. Estes ciclos, neste caso, tendem a ser muito curtos, deste modo as tecnologias a usar no desenvolvimento devem proporcionar aos programadores a facilidade de programar o que está especificado de uma forma mais prática e sem problemas na implementação dos requisitos analisados na fase anterior. Muitas vezes estas iterações são tão rápidas que podem gerar confusão entre as distintas fases, isto é, se as ferramentas usadas criarem entropia na implementação pode dar-se o caso da fase de especificação, implementação e teste ser bastante mais demorada e, como

é senso comum, o desenvolvimento está sempre a mudar, muitas vezes devido a novos requisitos do cliente, o que pode levar a que seja necessário recuar nalguma iteração e refazê-la para ir de encontro aos desejos de quem irá adquirir o produto.

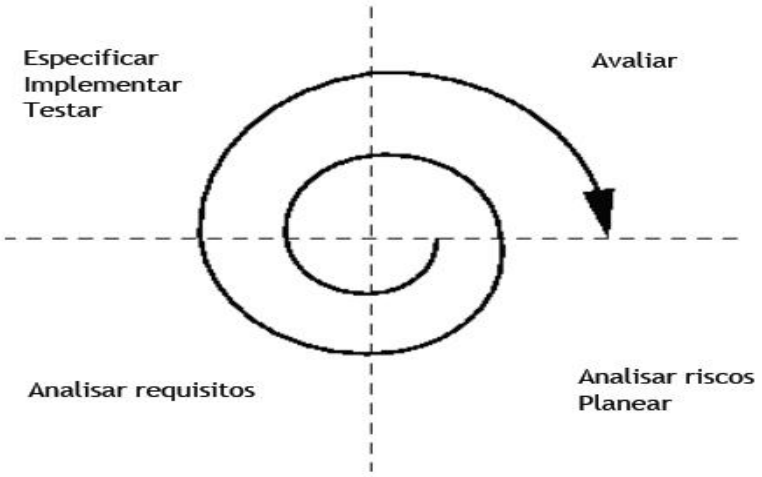

Figura 2: Modelo de desenvolvimento em espiral – Retirado de [DSI, 2007]

Quando nos encontramos num desenvolvimento de uma aplicação *web* os riscos analisados e planeados têm um aspecto central no início do processo, pois é nesta fase que se definem as tecnologias a usar e as abordagens a considerar para o correcto desenvolvimento da aplicação. No contexto deste trabalho, consideramos que foi escolhida, uma estrutura para o desenvolvimento de interfaces gráficos com o utilizador (uma *framework*) orientada à acção, sendo esta ferramenta responsável pela abstracção do tratamento dos pedidos do cliente e as respostas enviadas do servidor. Foi considerada, a *framework Webwork* 2.1.7, sendo que as metodologias propostas têm como pressuposto adequarem-se a qualquer tipo de *framework* usada no desenvolvimento *web*.
Considerando que, normalmente, os programadores recorrem a estas estruturas para que o seu trabalho de codificação contenha o menor número de erros possível, devemos ter em conta também a forma como essas ferramentas são usadas. Para a apresentação do problema analisado neste trabalho é necessária a apresentação da arquitectura base de uma *framework* orientada à acção e também perceber as dificuldades que os programadores encontram quando o

desenvolvimento da aplicação tem iterações muito rápidas e a própria aplicação se encontra em constante modificação/implementação. Normalmente este tipo de aplicações tem de possuir, no seu desenvolvimento, de metodologias que reduzam o tempo que os programadores passam a codificar os requisitos, para assim responder mais rápida e eficientemente a novas implementações e alterações de requisitos. Para este trabalho foi considerado que a equipa de desenvolvimento recorre a uma *framework* orientada à acção para o desenvolvimento dos interfaces gráficos com o utilizador e abstracção de pedidos/respostas. A arquitectura é apresentada na Figura 3 e tem como pressuposto o recurso a *Webwork*, podendo ser generalizada a qualquer tipo de *framework* orientada à acção.

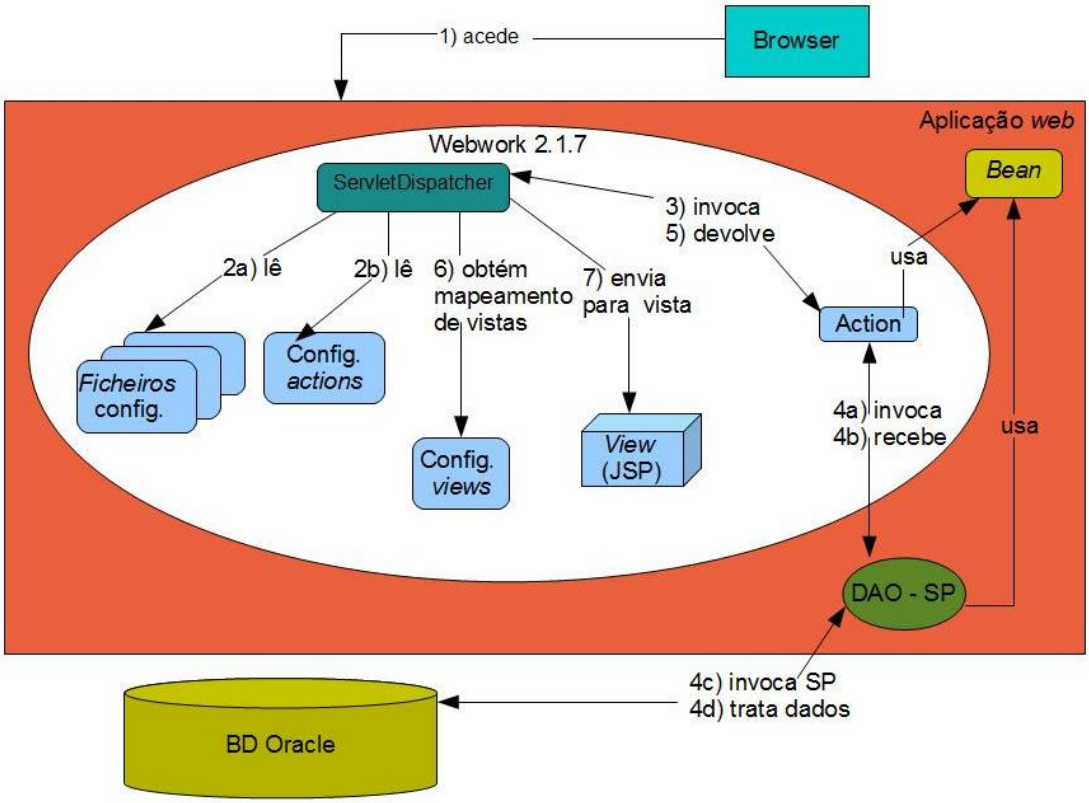

Figura 3: Arquitectura orientada à acção

Antes da análise da arquitectura é apresentado um exemplo de um pedido de um cliente a um servidor aplicacional, sendo, de seguida feita a devida análise de acordo com esta arquitectura.

Imagine-se uma simples aplicação onde o utilizador insere os seus dados pessoais e depois envia para a aplicação (não são consideradas quaisquer validações neste exemplo) (Figura 4):

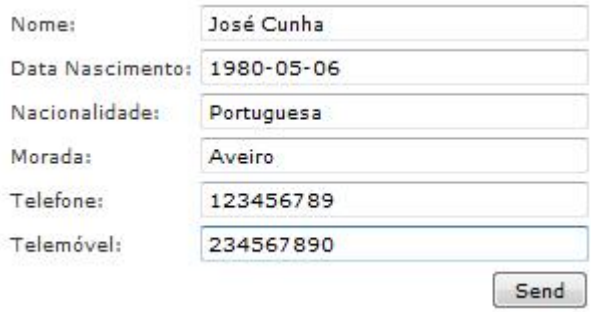

Figura 4: Exemplo de formulário

Deste modo, o pedido efectuado pelo utilizador passa pela introdução dos dados nos respectivos campos e posterior envio.

Neste exemplo podemos também que o utilizador irá ter a possibilidade de consultar os dados de outros utilizadores existentes no sistema, efectuando também um pedido para esse caso.

Nome	Idade	País
José Cunha	29	Portugal
John Doe	29	USA
Jane Doe	19	Italy

Figura 5: Exemplo de tabela de consulta

Observando a tabela vemos que existem apenas três colunas de informação. Consideramos que os dados, no preenchimento do formulário, são enviados para uma base de dados através da invocação de um *stored procedure* da mesma, sendo que a leitura dos dados respeitantes à consulta serão feitos através do mesmo mecanismo. Esta consideração é feita de acordo com os pressupostos do trabalho, pois baseiam-se em aplicações *web* onde o armazenamento/leitura de dados de uma BD é feito através da invocação de *stored procedures*.

Analisando a arquitectura é possível identificar e mapear as várias fases de acordo com o exemplo.

1. o *browser* acede à aplicação *web* recorrendo a um URL que lhe irá apresentar o formulário de inserção de dados, o utilizador preenche os dados e envia para o servidor;
2. o pedido efectuado pelo *browser* é interceptado por um *servlet* encarregado dessa função:
   - o *servlet* é declarado num ficheiro de configuração da aplicação *web* (web.xml) , através da inclusão da seguinte declaração:

```
<servlet-name>webwork</servlet-name>
<servlet-class>
com.opensymphony.webwork.dispatcher.ServletDispatcher</servlet-class>
</servlet>
```

Os pedidos do *browser* são feitos através da invocação de um URL padrão que se encontra devidamente mapeado no ficheiro *web.xml*:

```
  <servlet-mapping>
    <servlet-name>webwork</servlet-name>
    <url-pattern>*.action</url-pattern>
  </servlet-mapping>
```

   a) O *servlet* é responsável pela leitura de ficheiros de configuração, onde se encontram as propriedades definidas para a aplicação, assim como para as *actions* desencadeadas pelo cliente.
   b) É feita a leitura do ficheiro de configuração das *actions*. Neste ficheiro encontram-se declaradas as *actions* que devem ser interceptadas pelo *servlet* e também qual a classe que se encontra associada a cada *action*, o tipo de resultado da mesma, se possui validações a serem efectuadas e quais (se existirem) os outros mecanismos de intercepção disponíveis. No caso do exemplo existe uma *action* declarada na classe *PessoaAction* (ver Anexo B. Action inseredados()).

2. Após a leitura da configuração de uma *action*, o *servlet* então invoca a *action* pedida. A *action* é então responsável por toda a lógica associada ao pedido efectuado pelo cliente. É muito comum o recurso a *beans* para a alocação dos parâmetros enviados pelo cliente, permitindo que os parâmetros sejam devidamente alocados como atributos da classe.

Através do recurso à leitura dos parâmetros e de *Java Reflection* esses parâmetros são colocados ao dispor da *action* através dos métodos *setXXX*, onde *XXX* será o nome do parâmetro e o respectivo mapeamento na classe. No exemplo acima referido, quando o utilizador clica no botão de envio é despoletado um evento, no *browser*, que irá reunir a informação colocada pelo utilizador e enviar através do URL esse pedido para o servidor:

```
http://exemplo.com/insereDados.action?nome=José%20Cunha&dataNasc=1980-
05-
06&nacionalidade=Portuguesa&morada=Aveiro&telefone=123456789&telemovel=2
34567890
```

Chegando ao servidor e já depois do *servlet* ter feito o devido encaminhamento para a *action* pretendida os parâmetros são colocados ao dispor da acção. A acção fica então responsável por enviar os dados para a BD. Para tal, invoca um objecto que lhe permite aceder aos dados (*DAO*).

3. Para o armazenamento/leitura dos dados recorre-se a uma camada intermédia que é responsável pela comunicação com a BD, assim como pela invocação de procedimentos na mesma de forma a que os dados sejam gravados/lidos.
   a) A invocação do *stored procedure* existente na BD é feita através de um objecto que usa a conexão com a BD (Oracle) através do driver *Open Database Connectivity* (ODBC), registando os atributos do objecto como parâmetros do *stored procedure* (ver Anexo D. PessoaDAO). É uma metodologia comum, para a construção do código de classes deste tipo, proceder ao recurso de um utilitário que, através de uma *query*, obtém a forma de construir o código para determinada invocação. Bastando para tal recorrer a esse utilitário para se obter um DAO para cada *stored procedure* a invocar.
   b) c) e d) O *stored procedure* é executado e o seu resultado enviado para a acção. Nos casos em que existam dados devolvidos, é prática comum usar *beans* para alocar os dados resultantes. Deste modo, seria necessário percorrer o *ResultSet* e, para cada linha do mesmo, colocar os dados como atributos do *bean*.
4. A *action*, após todo o seu processo de execução, devolve a informação/dados ao *servlet* .

5. Através da leitura dos ficheiros de configuração, o *servlet* sabe para onde terá de direccionar o resultado da invocação da acção.
6. Os dados são enviados para a "vista" onde serão disponibilizados, através de *Object-Graph Navigation Language* (OGNL) [OGNL, 2006], os *beans* para a construção do HTML que será enviado para o cliente que fez o pedido.

Este mesmo processo poderia ser descrito de forma semelhante para o caso exemplo em que o utilizador obtém uma listagem dos dados dos outros utilizadores (Figura 5). Mas, para este exemplo, os programadores iriam implementar um *ListaPessoaDAO* (ver Anexo D. ListaPessoaDAO) que iria obter os dados necessários para apresentar a informação ao utilizador.

Todo este processo é relativamente simples de perceber e bastante fácil de implementar. Esta forma de desenvolvimento de uma aplicação *web* revela-se bastante prática quando existem equipas distintas na sua implementação:

- uma equipa responsável pela aplicação das regras de negócio no Sistema de Gestão de Base de Dados (SGBD) que disponibiliza uma *Application Programming Interface* (API) baseada em *stored procedures* para a equipa de desenvolvimento *web;*
- uma equipa de desenvolvimento *web* que é responsável pela implementação da arquitectura apresentada na Figura 3.

Mas podem sempre surgir algumas questões com a sua prática. Em projectos onde se recorra a esta arquitectura e que estejam em constante desenvolvimento espiral, com iterações rápidas, pode existir a necessidade de tornar o processo, ou partes do mesmo, mais generalizadas, de forma a potenciar os recursos humanos disponíveis para a sua implementação. Senão vejamos, de acordo com o exemplo:

- um novo requisito exige que seja armazenada a cor preferida do utilizador e também seja apresentado na listagem de utilizadores.
    - Na equipa de BD é codificada essa inclusão, assim como a colocação de mais um argumento na invocação do mesmo *stored procedure*. (Neste caso não é feita

nenhuma análise à forma como a entidade *Pessoa* é armazenada na BD, mas podemos, de uma forma simplista, supor a existência de uma tabela onde se encontram, como colunas, os campos anteriormente enunciados).

- Na equipa *web* este novo requisito exige várias alterações, desde a inclusão de um novo campo de introdução de dados, até à alteração da *action* invocada, assim como na invocação do *stored procedure* da BD, sendo ainda necessário alterar o *bean* para ir de encontro ao novo requisito.

Como se pode observar, a adição de "apenas" um novo "campo" pode tornar uma tarefa de programação propícia a erros, pois o programador *web* irá fazer diversas alterações para satisfazer o novo pedido. Neste exemplo, o programador irá colocar um novo atributo no *bean* "Pessoa" (cor), adicionar um novo `setter` e `getter` na acção `inserirDados`, colocar mais um argumento na invocação do SP e adicionar a colocação desse argumento no *statement* (na classe *PessoaDAO*) a enviar à BD, sendo também necessário proceder a alterações na classe *ListaPessoaDAO.* Após este processo de desenvolvimento é ainda necessário fazer o *deploy* da aplicação para o servidor. Mesmo que se use uma estrutura de directórios estendida (*WAR* estendido [Jamae & Johnson, 2008]) para colocar as alterações implementadas no servidor, há sempre a necessidade de "reiniciar" a aplicação *web*. Isto deve-se ao facto das classes necessitarem de ser compiladas e de o servidor precisar de voltar a instanciá-las, deveria ser necessário fazer apenas as alterações na *view* para corresponder ao novo requisito, mas, como vimos, tal não acontece. O exemplo acima descrito pode perfeitamente enquadrar-se com os problemas diários de uma qualquer aplicação desenvolvida. Existem inúmeras aplicações, hoje em dia, e algumas delas podem ser de tal forma críticas que uma paragem pode trazer implicações nefastas para o seu uso, privando os utilizadores do seu acesso durante um período de tempo.

Esta prática de desenvolvimento é muito comum no dia-a-dia da construção de aplicações *web*, principalmente naquelas em que existe uma elevada interacção com o cliente para a implementação dos requisitos.

Este trabalho faz uma análise da comunicação entre o servidor *web* e o SGBD e a forma como se pode melhorar o processo de desenvolvimento descrito a título de exemplo. Naturalmente, em aplicações com um grande volume de dados uma "pequena alteração" pode revelar-se uma enorme dor de cabeça para os programadores quando têm de fazer inúmeras alterações ao código já implementado.

O trabalho desenvolvido pelo mestrando concentra-se na forma como os dados são enviados/obtidos da BD, fazendo uma análise desta forma (enunciada como exemplo) de desenvolvimento *web* e providenciando uma solução que abstraia os programadores dos objectos que acedem aos dados existentes na BD.

1.4 Objectivos

Este trabalho tem como objectivo principal o possibilitar e facilitar a fácil integração entre duas equipas de desenvolvimento, *web* e BD, providenciando uma solução de interoperabilidade transparente entre as duas equipas, reduzindo, desta forma, o tempo necessário ao desenvolvimento de aplicações *web* que cumpram estes pressupostos. Em termos de sub-objectivos pretende-se:

- Desenvolver uma arquitectura que permita a comunicação entre um servidor *web* e um SGBD, de forma transparente aos programadores, numa aplicação *web* desenvolvida com recurso a uma *framework Java* para *web* que invoca *stored-procedures* de uma BD Oracle para implementar as regras de negócio do lado desta última e Java *beans* para mapear objectos provenientes da base de dados e/ou de pedidos ao servidor *web*;
- Colocar, junta e paralelamente, uma arquitectura que não obrigue o servidor aplicacional a fazer *re-deploy* da aplicação quando as alterações não o justificam;
- Avaliar o desenvolvimento de aplicações *web* com recurso a *frameworks* Java;
- Propor métodos para um melhor desenvolvimento de interfaces gráficas com o utilizador quando se recorre a uma *framework Java* orientada à acção;

- Avaliar o desenvolvimento de interfaces gráficas com o utilizador, a criação de eventos e as metodologias associadas a *frameworks Javascript* no contexto da metodologia abordada.

Para a persecução deste objectivo será necessário o recurso a diferentes áreas de conhecimento e ao recurso de normas abertas à comunidade. Em suma, este trabalho disserta sobre uma Metodologia para persistência de dados em *frameworks Java* para *web* orientadas à acção através de JSON recorrendo à *framework Webwork 2.1.7* [Webwork, 2005] e à chamada de *stored procedures* de *Procedural Language/Structured Query Language* (PL/SQL) através de ODBC.

1.5 Estrutura do documento

No Capitulo 2 apresentam-se normas e tecnologias usadas neste trabalho que são consideradas estado da arte nas suas áreas de aplicação. O Capítulo 3, abrange a problemática da interoperabilidade entre um servidor *web* e um servidor de BD quando existem duas equipas de desenvolvimento distintas, mas que necessitam de um ponto de comunicação, transparente a ambos, capaz de maximizar a produtividade no desenvolvimento de código, assim como facilitar a interacção entre a camada de Controle e o Modelo de Dados, em aplicações *web* desenvolvidas de acordo com o padrão "orientada à acção". No Capítulo 4, é apresentada a problemática do desenvolvimento de interfaces gráficos com o utilizador e como algumas ferramentas no mercado procuram solucionar esse problema através da reutilização e modularização de componentes HTML. O Capítulo 5, apresenta as metodologias que abrangem e desenvolvem os capítulos anteriores apresentando vantagens, desvantagens e problemas encontrados na sua aplicação.

No capítulo 6, é efectuada uma análise dos resultados obtidos neste trabalho e as conclusões que dele se retiraram bem como possíveis soluções e melhoramentos em trabalhos futuros.

Capítulo 2

2 Arquitecturas, Padrões e a Web

Ideias chave:
- Programação orientada a objectos
- Arquitecturas e padrões de desenvolvimento;
- DAO;
- POJO;
- Persistência de dados;
- J2EE;
- JSON e AJAX

Neste capítulo são abordadas as formas já encontradas, noutros trabalhos, de responder ao problema apresentado neste trabalho, assim como são apresentados alguns dos padrões mais comuns em desenvolvimento de aplicações e como o agrupamento desses padrões pode originar uma arquitectura, sendo analisada uma das arquitecturas com mais sucesso no desenvolvimento de aplicações *web* – MVC. As arquitecturas padrão, assim como os desenhos padrão podem possibilitar a abstracção de uma camada de persistência entre um servidor aplicacional e uma base de dados, sendo apresentadas formas que possibilitam essa abstracção. É apresentada a arquitectura que compõem o *Java Platform*, *Enterprise Edition* ou *Java EE* (J2EE), assim como as suas principais características e modo de funcionamento.

2.1 Programação Orientada aos Objectos

A Programação Orientada aos Objectos (POO) tem como base fundamental a noção que lhe proporciona o nome – o Objecto. Cada objecto é uma unidade de software que tem a capacidade de armazenar estados, através de atributos que lhe são característicos, responder a pedidos que lhe sejam efectuados, interagir com outros objectos [DSI, 2007]. Cada objecto é uma instância de uma classe, assim é possível afirmar que a classe é a entidade que gera os objectos, de acordo com determinadas características estabelecidas na classe, sendo os atributos dos objectos. As vantagens proclamada pelos defensores deste paradigma referem-no como uma forma natural e compreensível de pensar, programar. A sua robustez e escalabilidade são apontados, também, como algumas das virtudes, assim como a facilidade que proporcionam na manutenção dos sistemas desenvolvidos e até no aumento de produtividade. Na POO temos vários aspectos fundamentais [DSI, 2007]:

- Identidade (dos objectos): cada objecto é único, mesmo existindo objectos iguais.
- Classificação: um objecto será uma instância da sua classe, assim uma classe é um aglomerado de objectos, que possuem o mesmo conjunto de propriedades, os mesmos métodos e as mesmas relações com outros objectos.
- Polimorfismo: um método poderá ter comportamento diferente em objectos de classes diferentes; permite uma forma de abstracção.
- Herança: cada objecto herda as propriedades da classe que lhe é superior, não havendo a necessidade de redefinir esses atributos, existindo uma organização nas classes.

Para a representação deste paradigma é usada, no desenvolvimento, a linguagem de programação JAVA. Esta linguagem possui muitas particularidades, não sendo o âmbito deste projecto explorar essas mesmas particularidades, para tal é sugerida a leitura de [Martins, 2000], havendo, sempre que necessário, a devida contextualização com o paradigma. Convém salientar, e referir, a inclusão do conceito de anotações, na versão 1.5, nesta linguagem de programação, sendo que tornou-se,

de imediato, bastante utilizado desde a sua inclusão nas suas especificações [GoF, 2006].
De acordo com os pressupostos do projecto, toda a análise, desenvolvimento e implementação é feita tendo como base a programação orientada aos objectos.

2.2 Arquitecturas e Padrões

O que são em concreto os desenhos padrão ou padrões de projecto, conhecidos também por *design patterns*? A tradução de *design pattern* é "desenho padrão" e este conceito surgiu no âmbito da arquitectura através do arquitecto Christopher Alexander [Alexander et al, 1977]:

"Os elementos desta linguagem são entidades denominadas de padrões. Cada padrão descreve um problema que ocorre várias vezes no nosso ambiente e assim descreve o núcleo da solução para esse mesmo problema, de tal forma que é possível usar esta solução um milhão de vezes, sem ter de voltar a repeti-la."

Este conceito foi aproveitado e trazido para a informática por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides (também conhecidos pela alcunha "*Gang of Four*" ou pelo acrónimo *GoF* [GoF, 2006] em 1994 [Gamma et al, 1994]. O seu trabalho visa descrever soluções para problemas comuns no desenvolvimento de *software*. Sendo assim, um *design pattern* pode ser considerado como uma descrição geral sobre como usar a solução de um problema em vários aspectos do desenvolvimento de aplicações informáticas. Isto deveu-se, principalmente, ao empirismo aplicado, pelos autores do livro, no dia-a-dia, nas suas tarefas enquanto programadores, verificando que alguns problemas tinham uma solução comum. Numa perspectiva do paradigma de programação orientada aos objectos, podemos ver os *design patterns* como a abstracção das interacções e relações entre objectos, sem conhecimento de qual o resultado final dessas mesmas interacções e relações. Estas soluções podem ser vistas como uma forma de acelerar todo um processo de desenvolvimento de uma aplicação, considerando que se usam padrões já comprovados como solução de determinados problemas [PatternWiki, 2009]. As várias formas de desenvolvimento de sistemas da informação [DSI, 2007] podem ser potenciadas se estes padrões forem aplicados desde o início do processo de desenvolvimento, pois em fases mais avançadas da implementação

do projecto podem revelar-se bastante úteis na resolução de maiores problemas, providenciando aos programadores e arquitectos uma melhor análise de todo o código produzido até então. Ao agrupar vários desenhos padrão podemos considerar que nos encontramos perante uma arquitectura padrão, sendo esta arquitectura capaz de resolver um determinado conjunto de problemas através da soma da resolução das suas partes. Deste modo, podem existir soluções generalizadas para problemas comuns no quotidiano aquando do desenvolvimento de Sistemas de Informação (SI). [Gamma et al, 1994] - Gama et al, propõem um conjunto de vinte e três padrões, mas, ao longo do tempo, foram surgindo mais depois da abordagem inicial. No trabalho desenvolvido decidiram agrupar os padrões em categorias:

- **Padrões Comportamentais**
- **Padrões Creacionais**
- **Padrões Estruturais**

De seguida são apresentados os padrões propostos por Gama et al - [Gamma et al, 1994] - em três tabelas, uma por cada categoria de padrões. Para uma melhor compreensão e dado que os termos são globais não foi feita a tradução dos mesmos.

Padrões Comportamentais
Chain of responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template method
Visitor

Tabela 1: Padrões Comportamentais

Padrões Creacionais
Abstract Factory
Builder
Factory Method
Prototype
Singleton

Tabela 2: Padrões Creacionais

Padrões Estruturais
Adapter
Bridge
Composite
Decorator
Facade
Flyweight
Proxy

Tabela 3: Padrões Estruturais

Não é intuito deste trabalho elaborar uma análise exaustiva dos padrões apresentados, servindo estes como suporte à resolução de problemas comuns no desenvolvimento de aplicações, assim é feita uma clarificação dos padrões usados sempre que tal se justificar, principalmente aqueles mais directamente relacionados com o desenvolvimento de aplicações *web*.

Apresentado o contexto dos padrões no desenvolvimento de SI, deve ser introduzida definição de *"padrão arquitectural*" ou *"arquitectura padrão".* Comparativamente aos padrões apresentados, podemos ver uma "arquitectura padrão" como sendo maior em escala. Habitualmente, uma "arquitectura padrão" é vista como um consenso que foi atingido no âmbito do desenvolvimento arquitectural de SI. Mas encontrar e aplicar a arquitectura padrão adequada para um determinado problema tem se revelado uma tarefa realizada através de tentativa e erro e sem um sistema pré-estabelecido. Mesmo neste campo mais abstracto existem diferentes formas de pensar, de analisar as arquitecturas padrão. Em [Avgeriou & Zdun, 2005] podem ser identificadas duas formas distintas, duas escolas de pensamento, mas que têm como finalidade encontrar soluções que resolvam problemas já ao nível de desenho arquitectural, além de ser apresentada uma possível solução, um campo neutro para que as duas formas coexistam.

São do interesse, para este trabalho, as arquitecturas padrão para aplicação distribuídas, desenvolvidas tendo em mente a *web*. Neste particular podemos encontrar uma evolução nas arquitecturas até aos dias de hoje, não sendo de todo únicas, nem pretendem resolver todos os problemas encontrados no desenvolvimento de aplicações *web*. As arquitecturas de uma única camada não são aqui exploradas, devendo-se ao facto deste tipo de arquitectura centralizar uma aplicação numa máquina só, contendo toda a estrutura necessária para o seu funcionamento numa única camada. Um exemplo de uma aplicação desenvolvida através desta arquitectura seria o de um utilitário para ver o sistema de ficheiros de uma máquina. Deste modo, é perceptível que o interesse, para as aplicações, se centre em arquitecturas com mais de uma camada.

2.2.1 Arquitecturas de duas camadas

As arquitecturas de duas camadas são habitualmente conotadas à evolução operada durante a década de 1980 quando as aplicações distribuídas começaram a surgir, criando um novo paradigma na computação distribuída. Habitualmente, estas arquitecturas são também designadas de: ***modelo cliente-servidor***.

Resumidamente, um cliente conecta-se a um servidor (Figura 6) e pede dados [Ntier, 2009]. É, de uma forma simplista, a ligação terminal-servidor ou ainda *browser*-servidor. Neste podíamos ter um cliente "inteligente" que realiza grande parte do trabalho a comunicar com um servidor "burro" ou um cliente "burro" (mais comum) que "fala" com um servidor "inteligente".[Petersen, 2007]

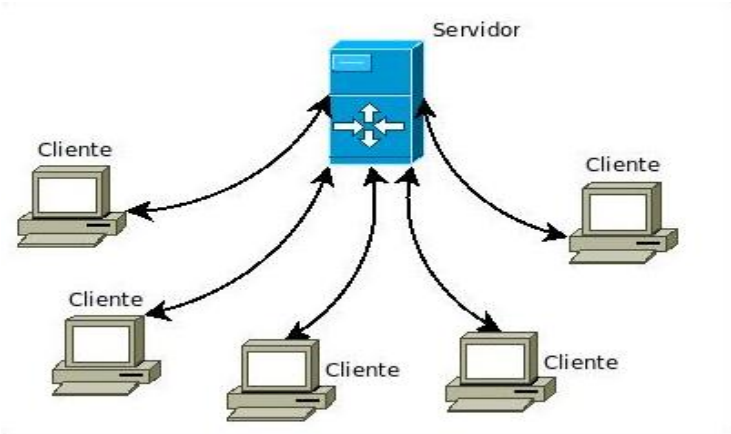

Figura 6: Modelo Cliente-Servidor

Este tipo de arquitectura permite ao cliente efectuar o pedido para obtenção de informação, guardar informação, ou enviar dados para serem tratados pelo servidor, por outro sistema. Comparativamente a aplicações de uma única camada podemos observar que existe um aumento da escalabilidade das aplicações desenvolvidas segundo este modelo arquitectural e apesar de já existir uma separação entre o que o cliente "vê" e a informação que lhe é disponibilizada, ainda não

é possível identificar uma separação mais funcional em mais camadas. Esta desvantagem levou a que muitas aplicações tenham optado por abandonar este modelo, passando para modelos/arquitecturas que disponibilizam uma maior separação das camadas, possibilitando a centralização e especialização de cada camada em módulos de desenvolvimento distintos, havendo mesmo, desde então, a procura de recursos habilitados para o desenvolvimento de cada camada.

2.2.2 Arquitecturas de três camadas

Para colmatar as desvantagens apresentadas pelo modelo cliente-servidor surgiu uma arquitectura de três camadas. Nesta arquitectura (Figura 7) existe uma clara separação entre a camada de apresentação e uma camada intermédia onde residem os servidores aplicacionais, contendo objectos representativos de entidades de negócio e as operações sobre estes e a última camada, a de dados, é responsável pelo armazenamento dos dados [Aarsten et al, 1996].

Esta camada vem aumentar a escalabilidade de um sistema de informação, pois ao separar a ligação directa que existia entre a camada superior e a inferior, faz com que seja possível adicionar novos componentes na camada intermédia. Desta forma, é completamente transparente para o utilizador os processos envolvidos no uso do sistema, não lhe sendo possível inferir sobre o tratamento dos dados enviados e mesmo existindo erros, ou até mesmo falhas, estes podem ser sempre tratados pela camada intermédia e caso não existam soluções (ou as regras de negócio assim o definam) será sempre possível informar o utilizador do erro em questão.

Esta arquitectura veio possibilitar uma forma mais estruturada de desenvolver SI distribuídos, com a devida separação das camadas torna-se também possível a existência de técnicos especializados em cada uma permitindo uma total separação de tarefas, havendo alguns mais propensos ao tratamento/criação de interfaces gráficos com o utilizador (a camada de apresentação) outros ainda mais direccionados para toda a lógica associada ao armazenamento de dados (a camada de dados) e também quem esteja mais orientado para o tratamento das interacções entre as duas camadas.

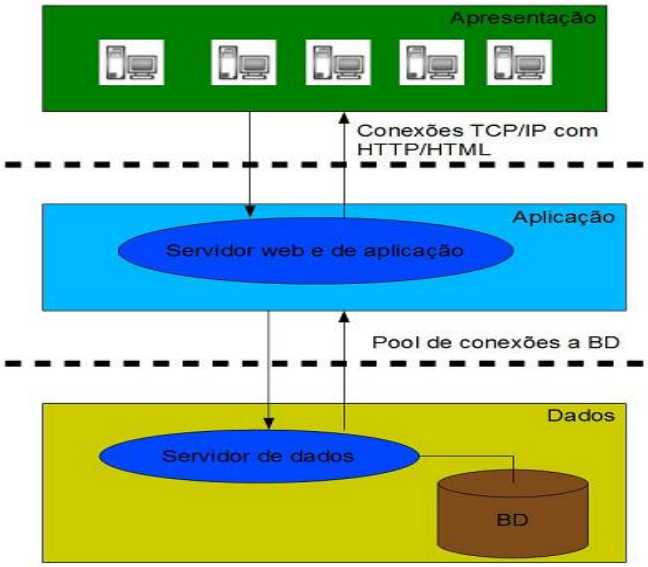

Figura 7: Arquitectura de 3 camadas

Com a evolução natural das arquitecturas surgiram, para o desenvolvimento de aplicações *web,* estruturas arquitecturais próprias para este universo. É o caso de uma das arquitecturas mais correntemente usadas e referenciadas neste campo: o *Model-View-Controller*, também conhecido pelo acrónimo MVC.

2.2.3 A arquitectura Padrão *Model-View-Controller*

Desde o início do desenvolvimento de aplicações que existiu uma necessidade de separar o código produzido. Nos primórdios do desenvolvimento de aplicações *web* todo o código era criado dentro da mesma estrutura, revelando assim uma enorme dificuldade na sua manutenção e também na sua compreensão. Foi então percepcionado que se deveria separar o código em diversas camadas, de forma a proporcionar uma melhor manutenção, compreensão e reutilização [Conallen, 1999]. Não existia a necessidade de colocar a lógica de negócio na apresentação de interfaces ao utilizador, algo que complicava o código de quem desenvolvia. Estruturando um projecto em diversas camadas melhora-se a produtividade e secciona-se, através de equipas, o seu desenvolvimento, obedecendo a regras claras de separação das camadas. Assim foi perceptível

que o que era respeitante a sistemas de armazenamento de dados (Bases de Dados, por exemplo) deveria ficar associado a uma das camadas, tratando da implementação de um Modelo de Dados. Verificou-se que existia a necessidade de efectuar alterações na implementação do Modelo de Dados, com nova informação ou através da remoção da mesma, esta camada, intermédia, é a camada aplicacional uma implementação de um Controle, sendo responsável por todas as alterações feitas à implementação do Modelo de Dados, é a camada que fará a transição entre a informação proveniente de um *user interface (UI)* e a actualização dos dados existentes. Deste modo, é compreensível a existência de uma última camada, a de Visualização, onde são apresentadas as formas do utilizador introduzir dados, de disponibilizar informação ao mesmo. A separação destas camadas é baseada numa arquitectura padrão [Burbeck, 1997] (*Model-View-Controller)* apresentada de seguida.

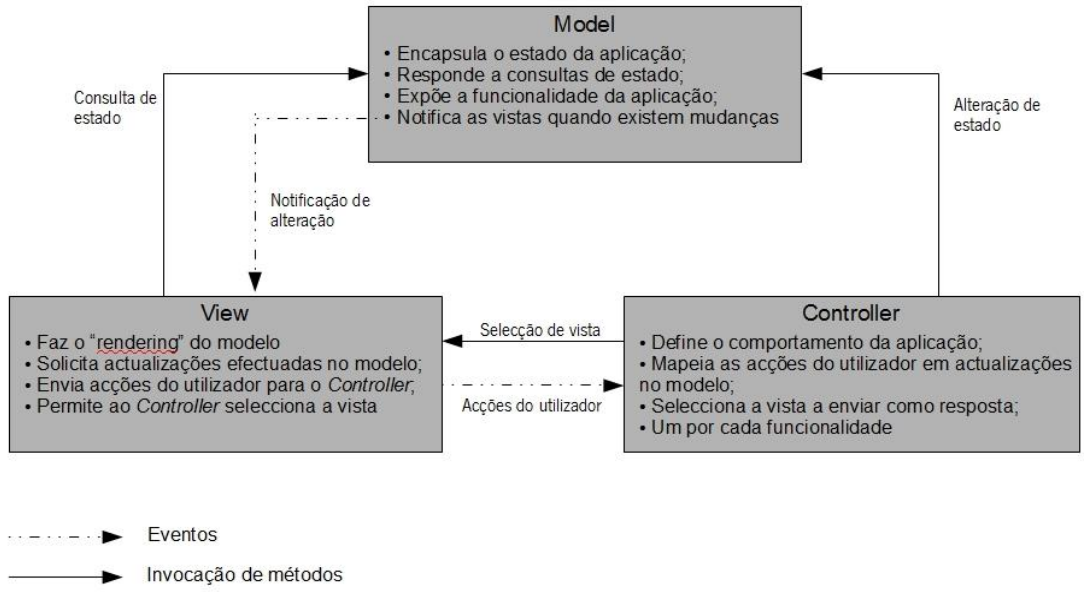

Figura 8 - Arquitectura MVC – Baseado em [DesigningJ2EE, 2005]

Definição

Model-View-Controller (MVC) é uma arquitectura padrão usada em Engenharia de Software. Compreende a separação da lógica de negócio dos interfaces com o utilizador e a sua implementação bem conseguida permite a existência dessa separação, sendo possível obter uma

aplicação onde se torna fácil fazer alterações na camada visual ou nas regras de negócio, sem interferir um com o outro.

Torna possível a modularidade da aplicação, possibilitando a colocação em máquinas distintas dos diversos módulos. A arquitectura permite que sejam feitas alteração na lógica de negócio sem afectar o interface com o utilizador e vice-versa.

Assim:

- ***Model****:* diz respeito à informação da aplicação e às regras de negócio que permitem manipular essa mesma informação.
- ***View***: encontra-se directamente relacionado com o que utilizador "vê" na aplicação e lhe permite usar.
- ***Controller***: faz a gestão da comunicação entre as acções desempenhadas pelo utilizador (e. g. clique no rato, carregar determinada tecla) e o modelo usado.

De acordo com a arquitectura MVC, as *frameworks* podem implementá-la através de uma de duas formas: ***Push*** ou ***Pull***, a saber [Push&Pull, 2003]:

- ***Push***: - as *frameworks* usam determinadas acções que fazem o processamento necessário e, de seguida, enviam a informação (*push*) para a *View* de forma a que o resultado seja apresentado.
- ***Pull***: - esta forma é também conhecida por ser "baseada no componente", ou seja, as *frameworks* que a implementam começam pela camada de visualização *(View)* que depois "retira" (*pull*) os resultados de vários controladores se necessário.

Esta arquitectura é também associada ao padrão *Observer* (Figura 9) de [Gamma et al, 1994], tendo sofrido evoluções ao longo do tempo. Uma dessas evoluções levou ao ao aparecimento de uma nova forma - o *Model 2* da arquitectura MVC. Neste caso, um pedido efectuado, pelo cliente, é

primeiro interceptado por um *servlet*, de controlo, que faz o processamento inicial do pedido e determina qual a vista a disponibilizar em seguida [Ozyurt, 2003].

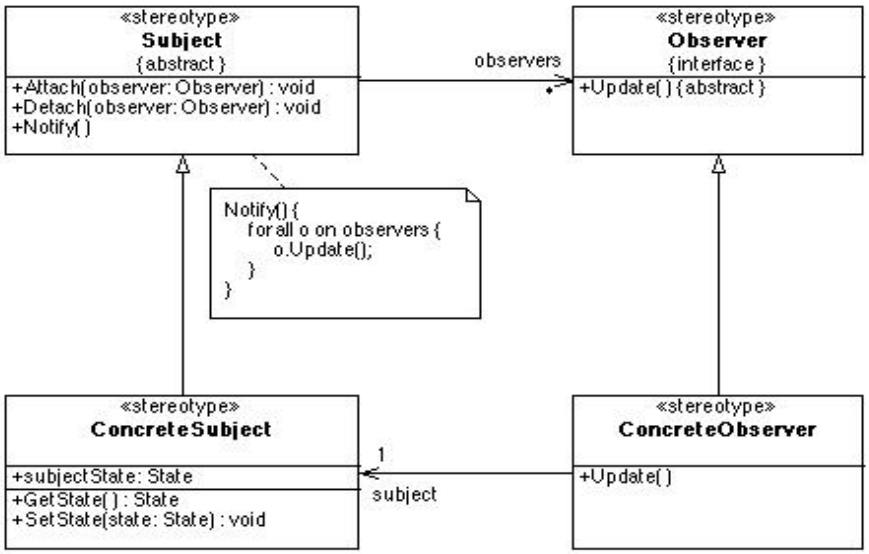

Figura 9 - *Observer pattern* - Retirado de [GoFPatterns, 2001]

2.2.4 Data Access Object (DAO)

Este padrão surge como um dos padrões nucleares da arquitectura J2EE [Alur et al, 2001][J2EE, 2007].

O padrão DAO visa:

- a separação da interface de acesso aos recursos de dados (uma base de dados, por exemplo) dos mecanismos de acesso a dados
- adapta uma API específica de acesso aos dados a interface genérico

Deste modo é perceptível a sua utilidade no desenvolvimento de aplicações que acedem a um sistema de armazenamento de dados. Assim o DAO é o responsável por gerir a conexão com o sistema de armazenamento para obter e guardar dados.

A estrutura é a apresentada pela figura 10 :

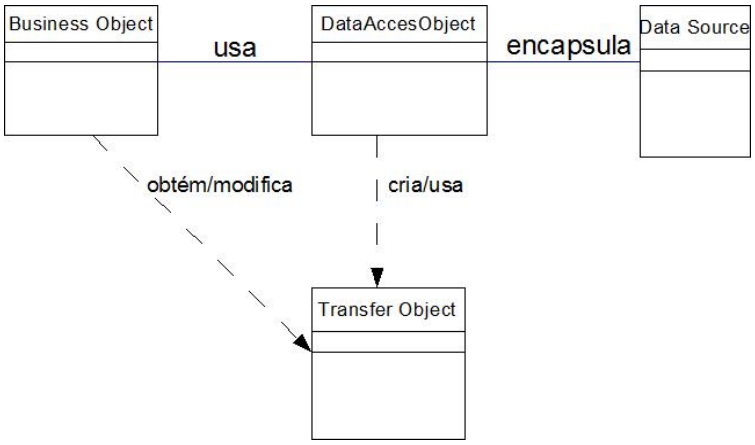

Figura 10 - *Data Access Object* – Figura baseada em [DesigningJ2EE, 2005]

Observando a estrutura é possível identificar quatro intervenientes no padrão [DAO, 2002]:

- ***BusinessObject***: este objecto representa os dados do cliente. Será este o objecto responsável por solicitar o acesso aos dados do *DataSource* para a manipulação dos mesmo. De acordo com a definição proposta, pode ser implementado através de um *session bean*, um *entity bean* ou qualquer outro objecto de acordo com um *servlet* ou *bean* auxiliar que aceda ao *DataSource*.
- ***DataAccessObject***: trata-se da razão de ser deste padrão, sendo o objecto principal. Faz com que o *BusinessObject* se abstraia por completo de toda a implementação que permite aceder aos dados, proporcionando uma forma transparente de aceder à informação/dados. Como se pode observar pela figura 10, o *BusinessObject* também delega as operações de carregamento e armazenamento dos dados neste objecto.
- ***DataSource***: representação da implementação do local de armazenamento dos dados/informação, podendo ser uma base de dados relacional, um repositório de ficheiros XML, um sistema de ficheiros, entre outros.
- ***TransferObject***: este objecto é representativo da transferência dos dados. O *DataAccessObject* pode usar um objecto deste tipo para enviar dados para o cliente, sendo também possível receber dados do cliente num objecto deste tipo para actualizar a informação/dados existente no local de armazenamento.

Estratégias no uso do padrão *DAO* [DAO, 2002]

Não existe uma forma única de implementar o padrão DAO. Dependendo da aplicação a desenvolver e do volume de dados contidos num local de armazenamento a estratégia escolhida pode sempre ser uma outra que não se encontre aqui enunciada. Para o problema em questão é importante referir que nos encontramos na primeira estratégia apresentada, onde os *TransferObject* são usados nos dois sentidos da comunicação.

- **Geração automática de código que implementa um DAO**:

Como cada *BusinessObject* corresponde a um DAO específico, é possível estabelecer relações entre o *BusinessObject*, o DAO e implementações que se encontrem em níveis inferiores (normalmente tabelas em bases de dados relacionais, mas também s*tored procedures).* A partir do momento em que essas relações estejam identificadas é possível criar uma pequena aplicação/utilitário responsável por gerar o código de todos os DAO da aplicação. Alternativamente, o gerador de código poderá analisar a base de dados e criar os DAO necessários para aceder à mesma.

- ***Fábrica (Factory)* para DAO**:

Esta estratégia é mais comum para situações em que não se espera mudar o sistema de armazenamento de dados. Aplicando os padrões *Abstract Factory* (Figura 11) e *Factory Method* (Figura 12) [Gamma et al, 1994] potencia-se uma maior flexibilidade no padrão DAO.

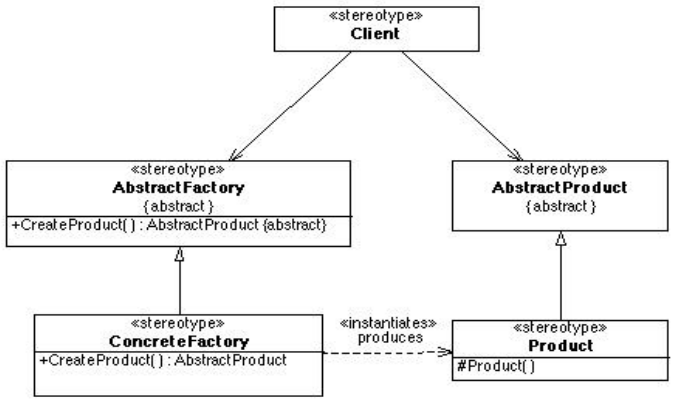

Figura 11 - Padrão *Abstract Factory* - Retirado de [GoFPatterns, 2001]

Assim e baseado nestes padrões, pode perceber-se esta estratégia através do diagrama de classes desta estratégia:

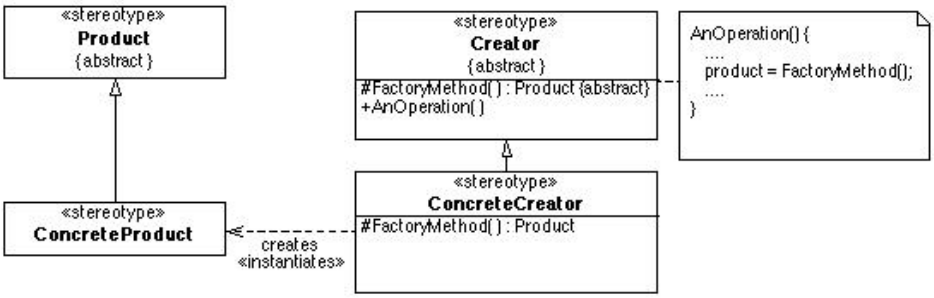

Figura 12 - Padrão *Factory Method* – Retirado de [GoFPatterns, 2001]

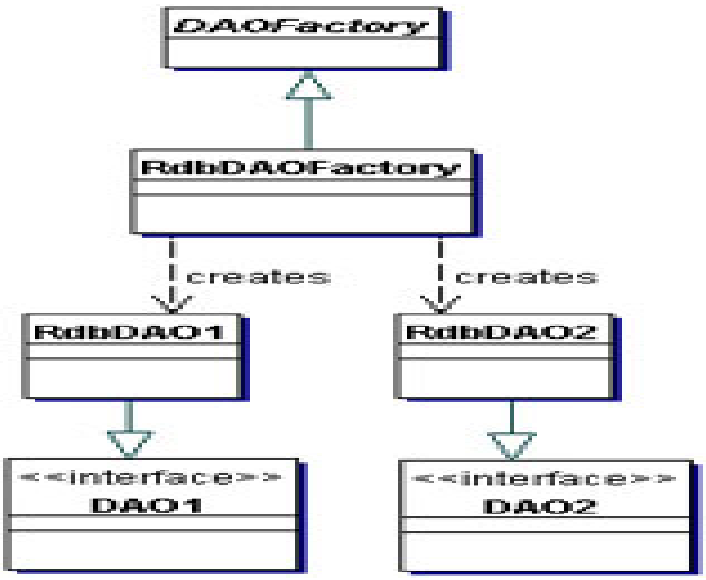

Figura 13 - Diagrama de classes para estratégia de *Factory* recorrendo ao padrão *Factory Method* [Gamma et al, 1994] - Retirado de [DesigningJ2EE, 2005]

Um dos objectos usados pelo padrão diz respeito ao "Transfer Object" , sendo responsável por fazer o transporte da informação entre o local de armazenamento dos dados e o "Business Object". Uma

das formas mais comuns de implementar o "Transfer Object" é recorrendo a *Plain Old Java Objects*, também conhecidos como POJO.

2.2.5 Plain Old Java Objects (POJO)

O termo *Plain Old Java Objects* foi introduzido em 1999 por Martin Fowler [Fowler, 1999] no intuito de a lógica de negócio ser codificada em objectos Java simples em vez de nos habituais *Entity* Beans. Os autores concluíram que os programadores não recorriam a este tipo de objectos meramente por não existir um nome "pomposo" com que pudessem tratá-los e, assim, atribuiu-lhes um nome mais apelativo. Estes objectos, para serem considerados como tal, não devem estar associados a nada mais do que aquilo que é especificado pela "Java Language Specification", deste modo não devem herdar métodos e propriedades de classes pré-especificadas, não devem implementar interfaces pré-especificados, nem conter anotações pré-especificadas. Basicamente, é um objecto *Java* da "velha guarda", sendo a estrutura básica da linguagem. Um exemplo muito simples de um POJO seria :

```
public class Pessoa implements java.io.Serializable {
        private String nome;
        private Integer idade;

        public Pessoa() {
        }

        public Pessoa(String nome, Integer cor) {
             this.nome = nome;
             this.idade = idade;
        }

        public String getIdade() {
             return idade;
        }
        public void setIdade(String idade) {
             this. idade = idade;
        }
        public String getNome() {
             return nome;
        }
        public void setNome(String nome) {
             this.nome = nome;
        }
}
```

Como se pode observar, o POJO é extremamente simples na sua implementação, sendo bastante útil para usar como "Transfer Object" em persistência de dados. De facto em aplicações empresariais é muito comum o recurso a POJOs e a *frameworks* não intrusivas, podendo, muitas vezes, o seu recurso melhorar o desenvolvimento [Richardson, 2006]:

- **melhoria da separação de conceitos**. Em vez de ser necessário pensar e fazer tudo ao mesmo tempo, assim concentram-se os esforços numa coisa de cada vez. Primeiro pode-se desenhar e implementar a lógica de negócio e então pode-se passar para a persistência e as transacções.
- **rápido desenvolvimento**. Assim é possível testar a lógica de negócio sem existir a preocupação com a persistência e a transacção de dados. Não há a necessidade de fazer a instalação dos componentes para testá-los, nem há a necessidade de manter um sincronismo entre o modelo de objectos e a BD. Assim os testes podem ser extremamente rápidos e o desenvolvimento seguir a mesma via.
- **melhorada portabilidade**. Não se fica preso a uma implementação particular de uma *framework* e o custo de mudar para a próxima geração de uma *framework* Java fica minimizado.

2.3 A lógica de negócio e a persistência de dados

Toda e qualquer aplicação desenvolvida tem em vista, de um modo geral, a satisfação de um modelo de negócio, seja na área médica, em redes sociais, telecomunicações, no comércio electrónico, entretenimento, entre outros. Assim a interacção entre um modelo de dados e a aplicação propriamente dita revela-se algo extremamente crucial no desenvolvimento de aplicações *web*. Existem inúmeras formas de desenvolver toda a lógica de negócio associada a uma aplicação, muitas vezes dependente da tecnologia actual, da vontade do cliente e até mesmo da administração de uma empresa.

Assim, um dos maiores desafios em desenvolver aplicações empresariais distribuídas, estruturadas numa arquitectura multi-camada, reside em saber qual a melhor forma de organizar os dados de

forma persistente, de disponibilizar uma arquitectura para a persistências dos dados na lógica de negócio. Nos últimos tempos o surgimento de bases de dados orientadas aos objectos (BDOO) surgem como um concepção apelativa para armazenar dados, mas as bases de dados relacionais (BDR) ainda prevalecem como uma das formas de garantir a persistência de dados, existindo, ainda, soluções baseadas em armazenamento em sistemas de ficheiros. [Korthaus & Merz, 2003]

Deste modo, é perceptível que a informação, os dados, são um factor crucial na vida de uma empresa. Muitas empresas, senão a maior parte delas, têm se apercebido que não é possível sobreviver num mercado competitivo sem a informação que existe nos seus SI.

Assim, podemos perceber perfeitamente a frase de Berry et al:

"Os dados de uma organização são, em muitos casos, a própria organização. Sem um consistente e bem definido conjunto de serviços para aceder à informação, por vezes heterogénea, dentro de uma organização, construir e integrar aplicações pode tornar-se um pesadelo." [Berry et al, 2002]

A forma como a informação é disponibilizada é então o factor preponderante no desenvolvimento de aplicações *web*. Percebe-se o porquê de terem surgido propostas para o desenvolvimento de aplicações multi-camada, deste modo é possível separar o desenvolvimento de aplicações *web* em multi-camadas, como visto anteriormente.

Existem muitas possibilidades quando se trata de implementar a lógica de negócio de uma aplicação, assim como as regras de negócio associadas. No âmbito deste trabalho, o estudo é feito considerando as aplicações *web* desenvolvidas numa arquitectura de três camadas (seguindo o Model2 da arquitectura padrão MVC) e considerando que o Modelo de Dados é implementado através de um SGBD Oracle, onde as regras de negócio e grande parte da lógica de negócio se encontram dentro do SGBD. A abordagem analisada tem em vista a disponibilização da informação (dos dados) do SGBD através de *stored procedures*. Esta metodologia é uma das abordagens para reduzir o número de acessos a um SGBD, minimizando o número de *queries SQL* necessárias.

2.3.1 Camada de persistência

Como vimos no início do capítulo, as aplicações com grande volume de dados têm necessidade de armazenar esses mesmos dados em algum local. Habitualmente é escolhido um SGBD relacional para armazenar os dados da aplicação e é neste ponto que nos iremos centrar. Quando se fala em persistência de dados no paradigma orientado aos objectos (em particular na linguagem Java) fala-se de como armazenar os dados numa base de dados relacional. Algumas das tecnologias mais proeminentes são analisadas no Capítulo 3, mas de momento centremo-nos na forma como essa persistência pode ser abordada no desenvolvimento de aplicações.

Em aplicações orientadas ao objecto, normalmente, encontramos uma ferramenta que faz a gestão dos pedidos de um cliente a um servidor e das respostas deste último ao primeiro. Revela-se importante perceber que, na arquitectura de três camadas, encontramo-nos no âmbito do *Controller*, responsável pela interacção entre a camada visual (*View*) e o modelo de dados (*Model*). Relembrando que o protocolo HTTP é caracterizado por não possuir estado [Fielding et al, 1999] umas das abordagens para criar estado em aplicações *web* passa por colocar na URL, no pedido ao servidor, os parâmetros que definem o estado da aplicação no cliente. Através de mecanismos de intercepção desses mesmo pedidos é possível colocar os parâmetros enviados em objectos apropriados, recorrendo ao conceito de OGNL [OGNL, 2006]. Estes mecanismos podem também ser usados para instanciar objectos provenientes de uma base de dados relacional. Recorrendo aos princípios de *Java Reflection* [Forman & Forman, 2005] é possível aceder às propriedades de cada objecto de uma forma simples.

2.4 Java Plataform, Enterprise Edition (Java EE)

O *Java Plataform, Enterprise Edition*, também conhecido por Java EE [J2EE, 2007] , é uma plataforma, conhecida mundialmente, que possibilita o desenvolvimento de aplicações, orientadas para o servidor, na linguagem de programação Java. Esta ferramenta disponibiliza funcionalidades para tolerância a faltas, distribuição de objectos e um software multi-camada, de acordo com componentes modulados que estão em execução num servidor de aplicações.

O Java EE possui várias especificações de API, nomeadamente JDBC, RMI, *web services*, entre outros. Além destas especificações "mais comuns" existem outras inerentes à própria tecnologia, como *servlets*, portlets, *Entreprise JavaBeans* (EJB), tecnologias próprias para o desenvolvimento de *web services* e, para a camada visual (não apenas) *JavaServer Pages* (JSP). As noções de *servlets* e JSP serão apresentadas posteriormente, pois encontram-se intrinsecamente relacionadas com o pressuposto deste trabalho.

É assim possível, através do Java EE, criar aplicações empresariais escaláveis, portáveis e reutilizáveis. Através de mecanismos próprios, um servidor Java EE é capaz de garantir segurança e concorrência, sendo também habilitado de lidar com transacções e gestão de componentes. Deste modo, quem desenvolve abstrai-se de toda a infra-estrutura que se encontra a usar e também de todas as tarefas de integração necessárias para a aplicação ser suportada, permitindo-lhe focalizar-se apenas no desenvolvimento da lógica de negócio.

É apresentada, de seguida, a estrutura das API para o Java EE 5:

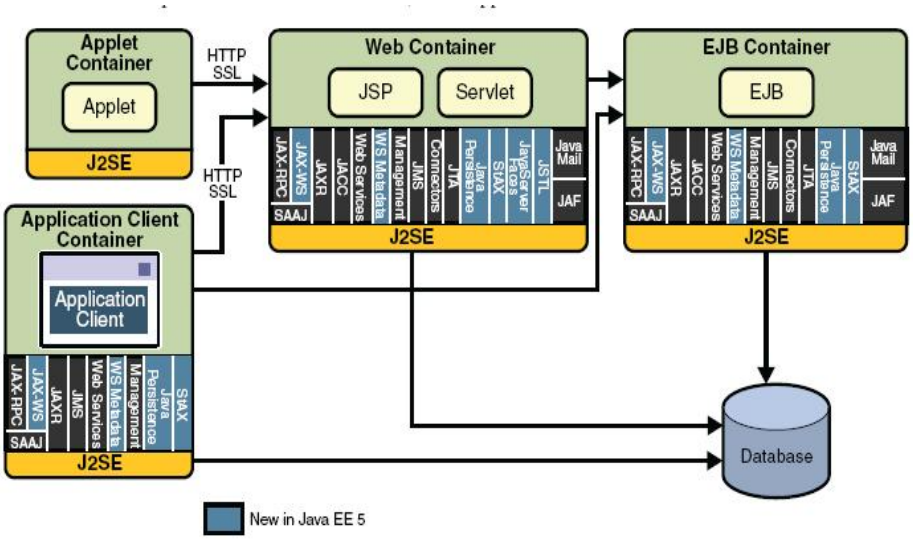

Figura 14: API do Java EE 5 - Imagem retirada de [J2EE, 2007]

2.4.1 Containers, Servlets e JSP

É importante para um programador *web* poder abstrair-se de todo o tratamento de pedido e resposta feito através do protocolo HTTP. Centrando-se no desenvolvimento da aplicação a sua produtividade aumentará e haverá também uma redução no tempo necessário para o desenvolvimento do projecto em questão. Desta forma torna-se evidente a introdução de um dispositivo que funcione como isolador e seja um interface entre um componente e as especificações de baixo-nível da plataforma. Essa interface será o ***container*** [J2EE, 2007] onde estarão os dispositivos que permitirão a abstracção do protocolo HTTP. O Java EE possui vários *containers*, de seguida apresenta-se um esquema de um servidor Java EE e os seus containers. De realçar a relevância para qualquer aplicação *web* do *web container*.

Desta forma, de acordo com a tecnologia anteriormente referida, surge a noção de ***servlet*** [J2EE, 2007]. Resumidamente, um *servlet* não é mais do que um objecto que recebe um pedido e produz uma resposta baseada nesse pedido. De acordo com as especificações, os *packages javax.servlet* e *javax.servlet.http* disponibilizam *interfaces* e classes para o desenvolvimento de *servlets*. Todos os *servlets* criados têm, obrigatoriamente, de implementar o *interface Servlet*, sendo estabelecidos vários métodos para o ciclo de vida do *servlet*.

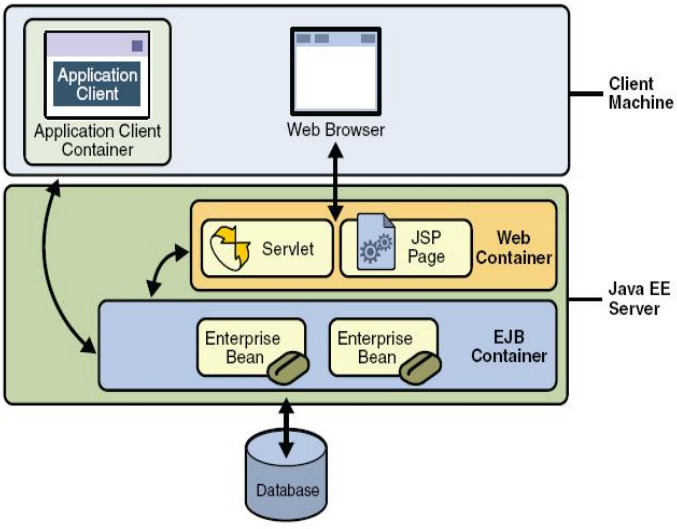

Figura 15: Servidor Java EE e *Containers* - Imagem retirada de [J2EE, 2007]

O ciclo de vida [J2EE, 2007] de um *servlet* é controlado pelo *container* onde se encontra inserido. Quando é efectuado um pedido, a um *servlet*, o *container* executa os seguintes passos:

1. se não existe uma instância do *servlet*, o *web* container:
    1. carrega a classe *servlet*;
    2. cria uma instância dessa classe;
    3. inicializa a instância do *servlet* através da invocação do método **init**.
2. É invocado o método do serviço, passando os objectos de pedido e resposta.

Se existe a necessidade do *container* remover o *servlet* então é invocado o método *destroy* do *servlet*. A tecnologia de *servlets* apesar de intrinsecamente ligada ao trabalho em si, não é do âmbito deste documento descrever as características da mesma, sendo apresentada, em secção própria, recursos onde é possível encontrar essa informação.

É perceptível toda a abstracção criada por este processo, mas ainda é possível ir mais longe, ou seja, atingir uma abstracção de alto nível em relação aos servlets de forma a que o código de visualização (HTML ou outro formato) seja gerado automaticamente. Assim estendeu-se o conceito de *servlet* sendo criada a tecnologia *JavaServer Pages* (JSP) [J2EE, 2007]. Esta abstracção permite a criação de conteúdos para web que possuem componentes dinâmicos e/ou estáticos.

As principais características desta tecnologia são apresentadas de seguida:

A. uma linguagem própria para desenvolver páginas JSP, que são documentos baseados em texto, onde se encontra descrita a forma de processar um pedido efectuado e a construção de uma resposta.
B. Uma linguagem de expressões (*Expression Language* – EL) que permite aceder aos objectos que se encontram no lado do servidor.
C. Mecanismos que permitam a definição de extensões à linguagem JSP.

2.5 Tecnologias na web

2.5.1 Javascript Object Notation – JSON

JSON é um formato de texto completamente independente de linguagem, pois usa convenções que são familiares às linguagens C e familiares, incluindo C++, C#, Java, Javascript, Perl, Python e muitas outras. Estas propriedades fazem com que JSON seja um formato ideal de serialização de dados estruturados. A sua origem encontra-se definida num sub-conjunto da linguagem *Javascript*, derivando dos *object literals* da mesma linguagem, de acordo com a definição *standard ECMAScript*. [ECMA, 1999][Fonseca & Simões, 2007][JSON, 2002] A sua arquitectura é extremamente simples e é caracterizada pela facilidade que os humanos têm em ler a sua estrutura e também pela forma rápida que as máquinas têm de a interpretar e processar.

No JSON existem duas estruturas de dados, a saber:

1. Um conjunto de pares chave/valor. Designado por *JSON Object*.
2. Uma lista ordenada de valores. Designado por *JSON Array*.

Podemos aceitar estas estruturas como universais, uma vez que, hoje em dia, as inúmeras linguagens de programação existentes as têm implementadas e as suportam.

A apresentação de dados, no caso do *JSON Object*, é feita de uma forma muito simples (como se pode observar na figura 3):

- Um *JSON Object* começa com `{` e termina com `}`.
- Cada chave é seguida por `:` e cada par chave/valor é separado por `,`.

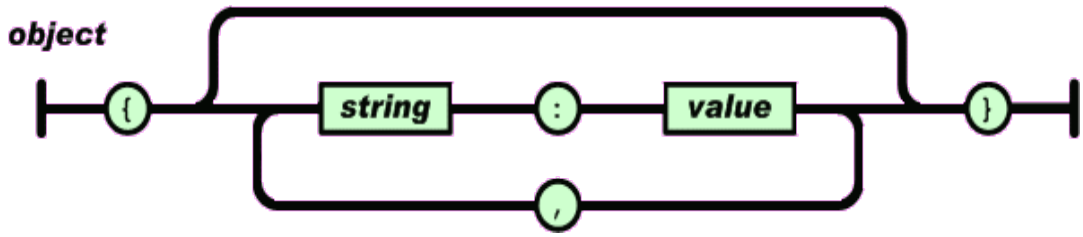

Figura 16: *JSON Object* - Retirada de [JSON, 2002]

Um exemplo de uma implementação desta estrutura poderia ser semelhante à que se segue:

```
{"nome":"João","idade":10}
```

No caso de se tratar de lista de valores a simplicidade também se mantém (Figura 8):

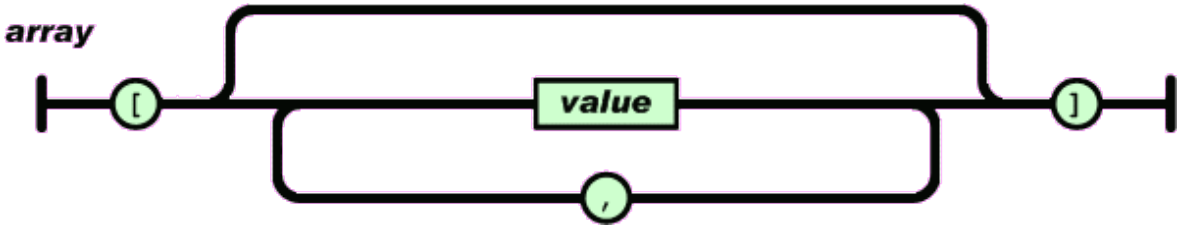

Figura 17: *JSON Array* - Retirada de [JSON, 2002]

- Um *JSON Array* começa com `[` e termina com `]`.
- Cada valor (Figura 5) deste conjunto pode ser um lista de caracteres (string), um número, um valor lógico (`true` ou `false`), um *JSON Object* ou até mesmo um *JSON Array*, podendo ainda ser `null`.

Um exemplo de uma implementação de um *JSON Array* poderia ser semelhante à que se segue:

```
[0,1,2,3,4,5]
```

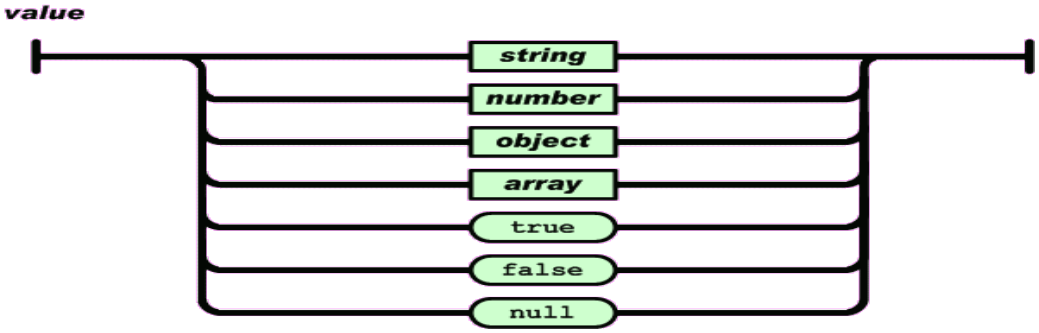

Figura 18: *Value* em *JSON Object* e *JSON Array* - Retirada de [JSON, 2002]

As *strings* são objecto de tratamento especial, pois não se poderão usar os caracteres de controlo " e \, sendo os responsáveis por identificar as propriedades de cada objecto JSON.

Assim, a representação de cada *string* é caracterizada pela figura 19:

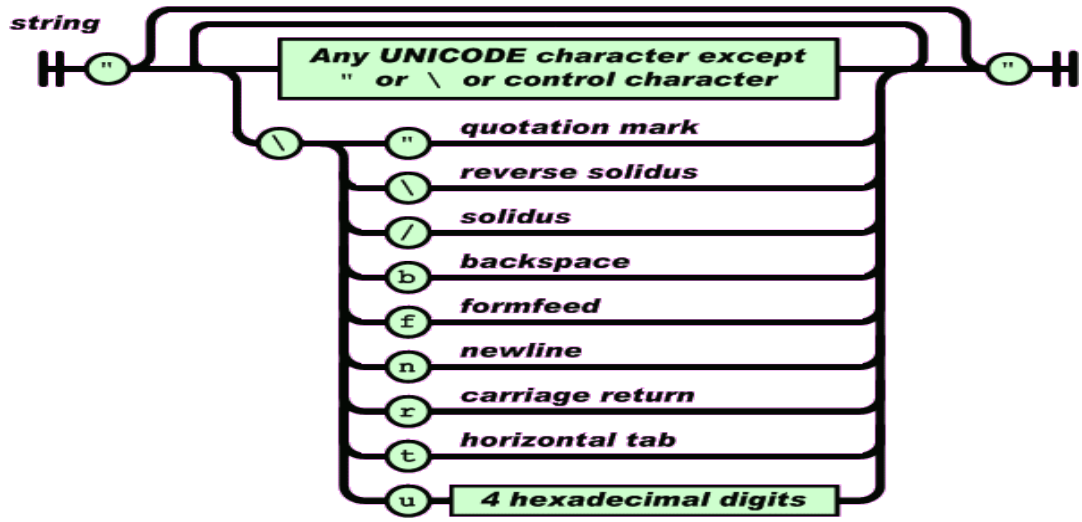

Figura 19: *Strings* como `value` em *JSON* - Retirada de [JSON, 2002]

Um *number* é muito semelhante ao encontrado noutras linguagens, sendo apenas usados números decimais.

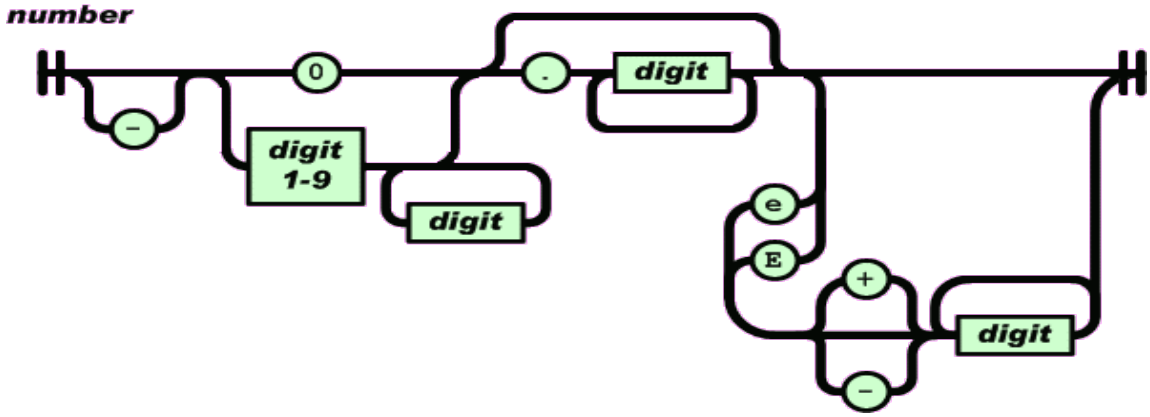

Figura 20: *Number* como `value` em *JSON* - Retirada de [JSON, 2002]

O JSON popularizou-se bastante no desenvolvimento de aplicações *web*, sendo usado como uma forma de comunicação entre um cliente e um servidor, existindo em qualquer *browser* moderno a implementação deste sub-conjunto do *Javascript*. [Crockford, 2006]

Deste modo e fruto do seu enorme sucesso surgiram, ao longo da sua curta vida, várias implementações para as mais variadas linguagens de programação como ASP [ASP_JSON, 2008],

Haskell [RJson, 2009], PHP [PHP, 2006], JAVA [JSONJava, 2002], ActionScript3 [JSON_AS3, 2009], C [JSON_Parser, 2009], e até mesmo PL/SQL [PL/JSON, 2009].

Os recentes avanços na tecnologia do lado do cliente (*browsers*) e a necessidade de obter mais informação em menos tempo, fazem com que os pedidos e as respectivas respostas a um servidor tenham de ser substancialmente rápidas, de modo a que a interactividade existente numa aplicação *web* seja cada vez maior. Fruto dessas necessidades e da contínua crescente popularidade do JSON surgem propostas para que os *browsers* comecem a implementar soluções mais seguras para pedidos enviados a um servidor no formato JSON e sendo as respostas também no mesmo formato. [Crockford, 2009] apresenta uma proposta para a próxima geração de aplicações *web*, pois as aplicações tendem a tornar-se, cada vez mais, propensas a um maior volume de dados e o `XMLHttpRequest` já não se enquadra nesta evolução, assim propõe que os *browsers* comecem a implementar soluções, principalmente relacionadas com segurança, tendo em vista as respostas recebidas através de JSON.

2.5.2 AJAX

Asynchronous JavaScript and XML (AJAX) é uma tecnologia que combina um conjunto de tecnologias já existentes. A ideia chave do AJAX é que os métodos *XmlHttpRequest* do *Javascript* podem ser usados para fazer pedidos ao servidor aplicacional em nome do utilizador que usa a aplicação. Esses pedidos são recebidos através de mensagens XML que são analisadas pelo analisador *Javascript* do *browser*. Este formato encoraja a que as mensagens XML enviadas pelo servidor sejam simples e pequenas, deste modo é possível não sobrecarregar o *browser* no processamento e interpretação da mensagem proveniente do servidor e como o tempo de resposta é relativamente curto existe uma maior interactividade. [Pierce & al, 2006]

O processamento no *browser* pode tornar-se um factor muito importante quando se constroem aplicações *web*, considerando que os recursos tendem a ser escassos e a grande parte das empresas se encontra, por vezes, "presa" a um determinado *browser*, existindo as limitações de memória que lhe são inerentes. Deste modo, o XML pode não ser a melhor alternativa para o envio

de mensagens simples e pequenas para um *browser,* existindo alternativas no mercado para permitir essa simplificação [Fonseca & Simões, 2007].

Capítulo 3

3 Mapeamento objectos/relacional

Ideias chave:
- Mapeamento objectos/relacional
- Ferramentas de persistência de dados:
  - JDBC
  - JPA;
  - EJB;
  - Hibernate

Sempre que uma aplicação *web* necessita de mapear os dados do modelo relacional em objectos *Java* recorre-se, normalmente, a *frameworks* de persistência de dados. Essas ferramenta têm como objectivo fazer o mapeamento do que existe na BD para algo que passará a ficar disponibilizado como um objecto na *web*. Neste capítulo são analisadas algumas das ferramentas que podem ser usadas em qualquer contexto de desenvolvimento de uma aplicação *web.*

3.1 Java Persistence API (JPA)

O Java Persistence Architecture API (JPA) é uma especificação Java para o acesso, persistência e gestão de dados entre objectos Java e uma base de dados relacional. O JPA foi definido como fazendo parte de especificação de EJB 3.0 em detrimento da especificação *Container-Managed-Persistence* (CMP) *Entity Beans* existente no EJB 2. É considerado por muitos como a principal abordagem para o *Object to Relational Mapping* (ORM) na comunidade Java. Mas o JPA é apenas uma especificação [JSR220, 2009] e, como tal, não é um produto e não é capaz de fazer persistência por si só.

Podemos ver o JPA como um conjunto de interfaces que necessitam de implementação. Deste modo e de uma forma perfeitamente natural, surgiram várias implementações (comerciais e livres) para escolha (Figura 21).

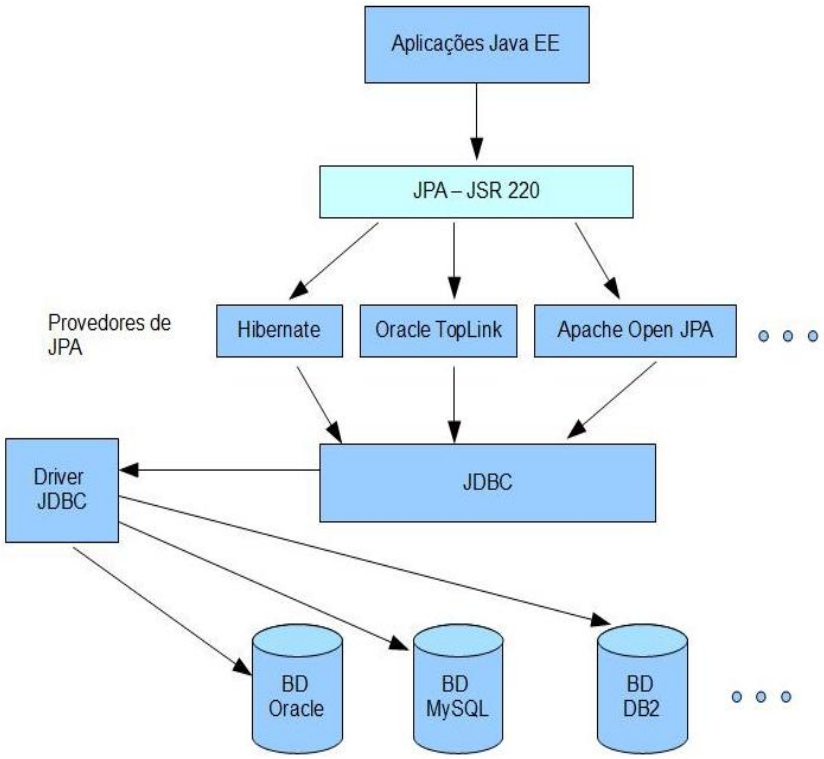

Figura 21: Implementações de JPA

Assim é perceptível que qualquer equipa de desenvolvimento possa recorrer a estas abstracções, criando uma camada intermédia entre o servidor *web* e o SGBD – a camada de persistência.

É colocada, portanto, uma camada responsável pelo mapeamento dos objectos Java nas correspondentes entidades da BD, criando, desta forma, a abstracção necessária para a comunicação entre a componente *web* e a camada de armazenamento de dados.

No contexto deste trabalho veremos as capacidades de usar EJB ou Hibernate como uma camada intermédia de persistência de dados, sendo, portanto, importante apresentar e analisar estas ferramentas de forma a considerá-las como uma ferramenta de integração entre a *web* e a BD,

validando a possibilidade que têm de invocar SPs de uma BD Oracle, de acordo com os pressupostos estabelecidos inicialmente para este trabalho.

Revela-se importante perceber que estas ferramentas aumentam o grau de complexidade na construção da aplicação *web*, mas, ao mesmo tempo potenciam a reutilização de código e reduzem os custos de manutenção do mesmo [Bauer & King, 2005] .

A escalabilidade é um componente crítico quando se trata de aplicações distribuídas e em aplicações *web* em particular. Habitualmente, uma pequena aplicação pode passar para produção para resolver um pequeno problema ou até mesmo como uma medida tampão nalgum caso particular. Com o tempo, essa pequena aplicação pode tornar-se o núcleo da estrutura de uma aplicação maior. Torna-se extremamente difícil fazer com que aplicações cliente/servidor cresçam se elas sofrerem de uma fase de desenho pobre, isto verifica-se ainda mais nas aplicações *web* porque tratam-se de aplicações distribuídas em multi-camada e, como tal, possuem ainda mais componentes no meio. Como vimos, o MVC e o seu *Model 2*, em particular, assim como os desenhos padrão e todos os padrões associados, vieram possibilitar isso, fazendo com que se criem aplicações cada vez mais modulares, onde as suas partes interagem umas com as outras de uma forma limpa, possibilitando uma maior facilidade em fazer alterações em qualquer módulo desenvolvido. [Ford, 2004]

3.2 Enterprise Java Beans (EJB)

Os nomes Java Beans e Enterprise Java Beans são muito semelhantes, mas existem enormes diferenças entre os dois. Já vimos exemplos de *Java Beans* e, de facto, são tão simples que um programador pode estar a codificar um sem que se aperceba, mas os EJB são substancialmente diferentes e bastante mais complexos. EJBs são componentes orientados para a lógica de negócio e comportamento. Como os *servlets*, são executados dentro de um *container*, num servidor aplicacional. Assim é perceptível que a criação, destruição e gestão de EJBs seja controlada por um servidor aplicacional. Existem três tipos de EJB, com sub-tipos em cada um dos principais tipos.

Tipo de EJB	Sub-tipo	Descrição
Session	*Stateless* (sem estado)	Componentes sem estado que executam invocações de um único método. Usados para lógica de negócio.
	Statefull (com estado)	Componentes com estado que actuam como um *proxy* para aplicações cliente. Estes componentes mantêm o estado entre as várias invocações de métodos
Message		Componentes sem estado que se encontram ligados ao *Java Message Service* (JMS), permitindo a invocação de métodos assíncronos.
Entity	*Container Managed Persistence (CMP)*	Componente com estado que encapsula uma entidade da BD. O servidor aplicacional cria o código necessário para a conexão com uma BD específica.
	Bean Managed Persistence (BMP)	Componente com estado que é capaz de encapsular uma entidade da BD mais complexa. O programador deverá escrever o código que permite o acesso à BD

Tabela 4: Tipos de EJB - Tabela baseada na listagem encontrada em [Ford, 2004]

Quando se usa um servidor aplicacional, a aplicação *web* é executada juntamente com os EJBs, habitualmente na mesma *Java Virtual Machine* (JVM). Uma das técnicas de escalabilidade usada em servidores aplicacionais é o recurso ao sistema de *pool* de objectos. Como a criação de uma nova instância de um objecto é um processo intensivo e uma tarefa que consome memória, o servidor aplicacional cria um elevado número de objectos e associa-os a um pedido de um cliente à medida que estes vão aparecendo. Em vez de permitir que o objecto saia do âmbito da aplicação e seja recolhido pelo *garbage collector*, o servidor aplicacional devolve o objecto ao sistema de *pool* de forma a que seja reutilizado em vez de destruído e novamente criado.

Assim, podemos perceber que um dos segredos para tornar uma aplicação escalável é fazer o máximo de *cache* possível. Muitos servidores aplicacionais possibilitam esta funcionalidade automaticamente, fazendo com que o programador se concentre na funcionalidade da aplicação.
A forma como os EJBs se relacionam com uma aplicação *web* é representada na figura que se segue:

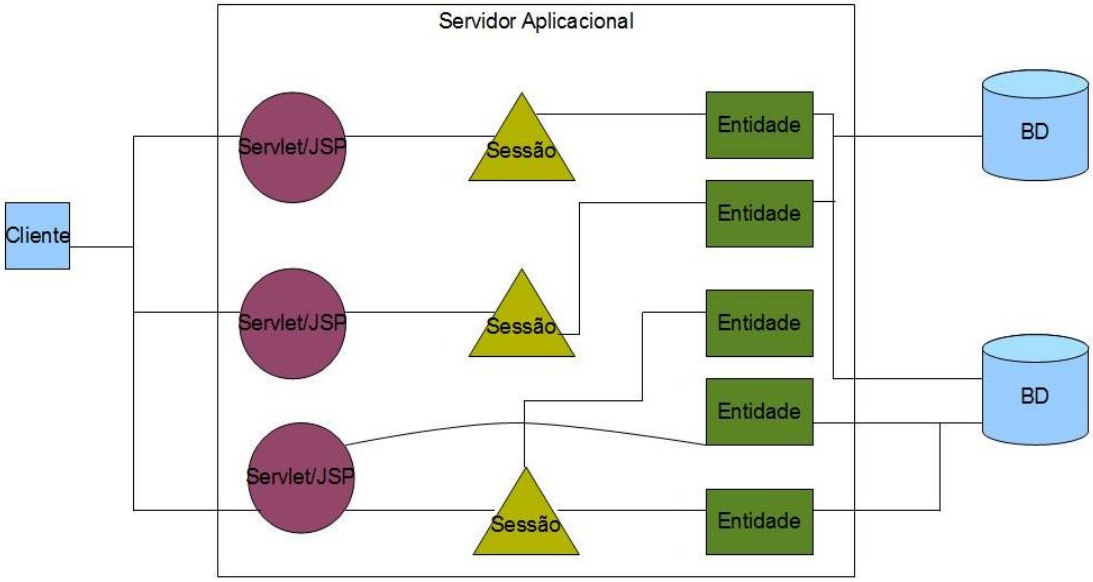

Figura 22 - Relação entre EJB e aplicação *web* – Retirado de [Ford, 2004]

É perceptível a existência das entidades EJB como sendo a ponte entre o servidor aplicacional e a base de dados. O recurso a EJB implica um conhecimento do funcionamento de anotações na linguagem de programação Java e, como veremos mais à frente, essa é forma de estabelecer relações, entre entidades, de acordo com a informação existente na BD. Torna-se, então, importante perceber as possíveis relações existentes entre as várias entidades (*Entity*) existentes no servidor aplicacional. Estas relações podem ser de diversos tipos [J2EE, 2007]:

- **um-para-muitos (1:N) ou muitos-para-um (N:1)**: no primeiro caso, para uma entidade existe uma relação com várias entidades, ou seja, existem várias instâncias com as quais se pode relacionar. Como exemplo, temos a entidade “Disciplina” e a entidade “Aluno”.

Podemos ver duas relações entre estas entidades, uma "Disciplina" tem vários "Aluno" (1:N) ou um "Aluno" tem várias "Disciplina" (também 1:N). Incluindo, neste exemplo, a entidade "Curso", podemos estabelecer a relação de que muitas entidades "Disciplina" encontram relação com a entidade "Curso" (N:1).

- **um-para-um (1:1)**: este tipo de relação estabelece que, para cada entidade, existe uma e só uma correspondente, com a qual se relaciona. Como exemplo, podemos ver esta relação da seguinte forma, existente no dia-a-dia: para cada pessoa existe um e só um número de identificação, o bilhete de identidade. Assim poderíamos ter a entidade "Pessoa" e a entidade "BIs" e para cada pessoa existe apenas uma identificação na entidade "BIs".
- **Unidireccionais:** as relações unidireccionais estabelecem que a relação entre as entidades só pode ser feito num sentido e não nos dois. Por exemplo, pode estabelecer-se a regra de que uma entidade "Pessoa" só pode ter um "Sexo" (1:1), mas não se pode estabelecer que uma entidade "Sexo" terá apenas uma relação com uma entidade "Pessoa". As relações unidireccionais podem funcionar como regras a estabelecer tanto para relações (1:1), como para (1:N) e (N:1).
- **relações auto-referenciais (*self-referencial*)**: - Uma relação auto-referencial é uma relação estabelecida entre os campos existentes na mesma entidade. Por exemplo, a entidade "Humano" pode ter estabelecidas relações internas entre as entidades "Pessoa", "BI" e "Sexo", isto é, se definirmos a entidade "Humano" como tendo essas entidades como suas propriedades para a definição de "Humano".

Apresentadas as relações possíveis em EJBs é altura de perceber como se desenvolve e se cria um Enterprise Java Bean. Para desenvolver um EJB é necessária a existência de vários ficheiros:

- ***Enterprise Bean*** classe: implementa os métodos definidos no "Business Interface" e ainda métodos de *callback* necessários ao ciclo de vida.
- ***Business* Interfaces**: define os métodos que terão de ser implementados pela classe "Enterprise Bean".

- **Classes Auxiliares**: outras classes que podem ser usadas pelo "Enterprise Bean", como tratamento de excepções.

Um exemplo de uma possível implementação seria a de uma aplicação que fizesse a conversão entre dólares e euros e vice-versa (baseado no exemplo apresentado em [J2EE, 2007]):

Assim, temos de definir, um interface onde constem as declarações dos métodos de conversão das moedas para posterior implementação por um "Enterprise Bean":

```
import java.math.BigDecimal;
import javax.ejb.Remote;

@Remote
public interface Conversor{
public BigDecimal dollarToEuros(BigDecimal dollars);
public BigDecimal eurosToDollar(BigDecimal euros);
}
```

É possível observar a existência da anotação `@Remote`, que será responsável por informar o *container* de que o *ConversorBean* será acedido por clientes remotos.

```
import java.math.BigDecimal;
import javax.ejb.*;

@Stateless
public class ConversorBean implements Conversor{
      private BigDecimal dolar = new BigDecimal("0.4901");
      private BigDecimal euro = new BigDecimal("1.4709");

public BigDecimal dollarToEuros(BigDecimal dollars){
      BigDecimal res = dollars.multiply(euro);
      return res.setScale(2,BigDecimal.ROUND_UP);
}
public BigDecimal eurosToDollar(BigDecimal euros);
      BigDecimal res = euros.multiply(dolar);
      return res.setScale(2,BigDecimal.ROUND_UP);
}
}
```

É de realçar a anotação `@Stateless` antes da declaração da classe. Deste modo, o *container* saberá que o "ConversorBean" é um *bean* de sessão sem estado.

Serve isto como pequena introdução à forma de construir EJBs. Vejamos como podemos usar os EJBs em aplicações *web* que obtêm os dados de uma BD através de *stored procedures*.

Da tabela 4 podemos perceber facilmente que os *EJB* mais relevantes para a persistência de dados são:

- *Container Managed Persistence* (CMP);
- *Bean Managed Persistence* (BMP)

CMP – este termo encontra-se intrinsecamente relacionado com o tratamento feito pelo EJB *container* para todos os acessos à BD necessários pelo *entity bean*. Deste modo, o *bean* não contém nenhumas chamadas SQL de acesso à BD. Assim, o código do *bean* não se encontra unicamente ligado a um único SGBD. Com este mecanismo, percebe-se que existe uma maior flexibilidade em usar o mesmo *entity bean* em diferentes servidores aplicacionais e diferentes aplicações *web* que usam diferentes bases de dados, pois não existe a necessidade de recodificar o *bean*, nem de recompilar o seu código. Através de CMP os *entity beans* ficam mais portáveis. Com este método existe uma separação entre os *beans* e a sua representação persistente, permitindo uma melhor separação de conceitos, deste modo os programadores têm de preocupar-se apenas com a codificação dos *entity beans* e delegando o tratamento da persistência dos dados para o *container*, permitindo que esta abstracção seja feita através de ficheiros de configuração, sendo que as invocações/chamadas necessárias para obter os dados de uma BD são geradas pelo EJB *container*, retirando esse controlo ao programador. [CMP, 2002]

BMP - recorrendo a este tipo de persistência, o *entity bean* contém a codificação necessária para aceder à BD, sendo que as chamadas necessárias encontram-se codificadas dentro do mesmo. Desta forma, torna-se necessário recorrer à API do JDBC para poder codificar o estabelecimento da conexão, assim como os métodos necessários para obtenção/armazenamento de dados, existindo, naturalmente a codificação das *queries* necessárias a serem executadas pela BD, tornando o *bean* directamente ligado com o modelo de dados implementado, reduzindo a sua portabilidade e uso noutras aplicações.

Um componente pode aceder aos dados e funções de um sistema de informação empresarial de várias formas, seja acedendo directamente através da API disponibilizada ou indirectamente através

da abstracção da complexidade e detalhes de baixo-nível de acesso ao sistema, recorrendo a essa abstracção através de objectos de acesso, como vimos anteriormente, os *DAO*.

Observando as duas possibilidades, no contexto deste trabalho e de acordo com os pressupostos, os BMP perfilam-se como os mais propícios a usar quando se usam *stored procedures* para a manipulação/obtenção dos dados, sendo usados DAO para a obtenção/manipulação dos dados existentes na BD. [Singh et al, 2002]

Desta forma, os programadores possuem maior liberdade para codificar o tratamento das chamadas e também a forma de alocar os dados provenientes da execução das chamadas na BD.

Muitas aplicações *web* são desenvolvidas considerando que o sistema de armazenamento de dados é imutável, ficando muito dependentes do SGBD usado. Neste tipo de desenvolvimento é muito comum o recurso a agrupamento de serviços na BD. Em aplicações empresariais, esta metodologia não é aconselhável, pois as *queries* e os *stored procedures* necessitam que os BMP codificados sejam muito complexos. [Singh et al, 2002]

Percebe-se que o recurso a EJB em aplicações baseadas no pressuposto deste trabalho, pode dificultar o desenvolvimento da aplicação, pois a complexidade que lhe é associada é elevada, não permitindo que a devida abstracção seja feita de um modo simples e eficiente.

3.3 Hibernate

O Hibernate é uma biblioteca *Object-Relational Mapping* (ORM) para a linguagem Java que possibilita o mapeamento de modelos de dados orientados a objecto com uma base de dados relacional. A sua principal característica é fazer o mapeamento entre classes Java e tabelas existentes numa BD, além disso também são disponibilizadas funcionalidades capazes de efectuar pedidos e obtenção de dados de uma BD. É responsável pela criação das *queries* SQL necessárias à obtenção dos dados e possibilita, ao programador, a não necessidade de codificar o conjunto de resultados provenientes da BD, assim como fazer o tratamento dos objectos e respectiva conversão

do que provém da BD. Desta forma, as aplicações desenvolvidas com recurso a Hibernate tornam-se mais portáveis e independentes do SGBD a usar. A melhor forma de conhecer a biblioteca é observando como se encontra estruturada, conhecendo os seus interfaces torna-se possível recorrer a Hibernate como uma camada de persistência a usar numa aplicação *web*.

Na Figura 22 podemos identificar os vários interfaces existentes no Hibernate, sendo que é possível enquadrá-los da seguinte forma [Bauer & King, 2005] :

- Interfaces invocados por aplicações para proceder às funções básicas sobre uma BD: criar, ler, actualizar, apagar (em inglês foi criado o acrónimo CRUD que identifica estas quatro operações: *Create, Read, Update, Delete*) e também para proceder a operações básicas de pedido de dados (*query*). Assim, esses interfaces são `Session`, `Transaction` e `Query`.
- Interfaces executados pela aplicação *web* (o seu código) que possuem as configurações necessários para o Hibernate: `Configuration`.
- Interfaces de *callback* que possibilitam, à aplicação, a reacção a determinados eventos ocorridos dentro da própria biblioteca: `Interceptor`, `Lyfecicle` e `Validatable`.
- Interfaces que possibilitam a criação de extensões à própria biblioteca, que devem ser implementados na própria aplicação: `UserType`, `CompositeUserType` e `IdentifierGenerator`.

Esta camada de persistência recorre a várias APIs existentes na linguagem Java, nomeadamente JDBC, *Java Transaction API* (JTA) e *Java Naming and Directory Interface* (JNDI). Deste modo, a biblioteca possibilita várias abstracções ao dispor dos programadores, fornecendo os meios necessários para uma redução no código produzido, assim como o isolamento dos conceitos, conectando a BD ao servidor aplicacional (onde se encontra a aplicação) através de uma camada de persistência que é portável e independente da aplicação a desenvolver.

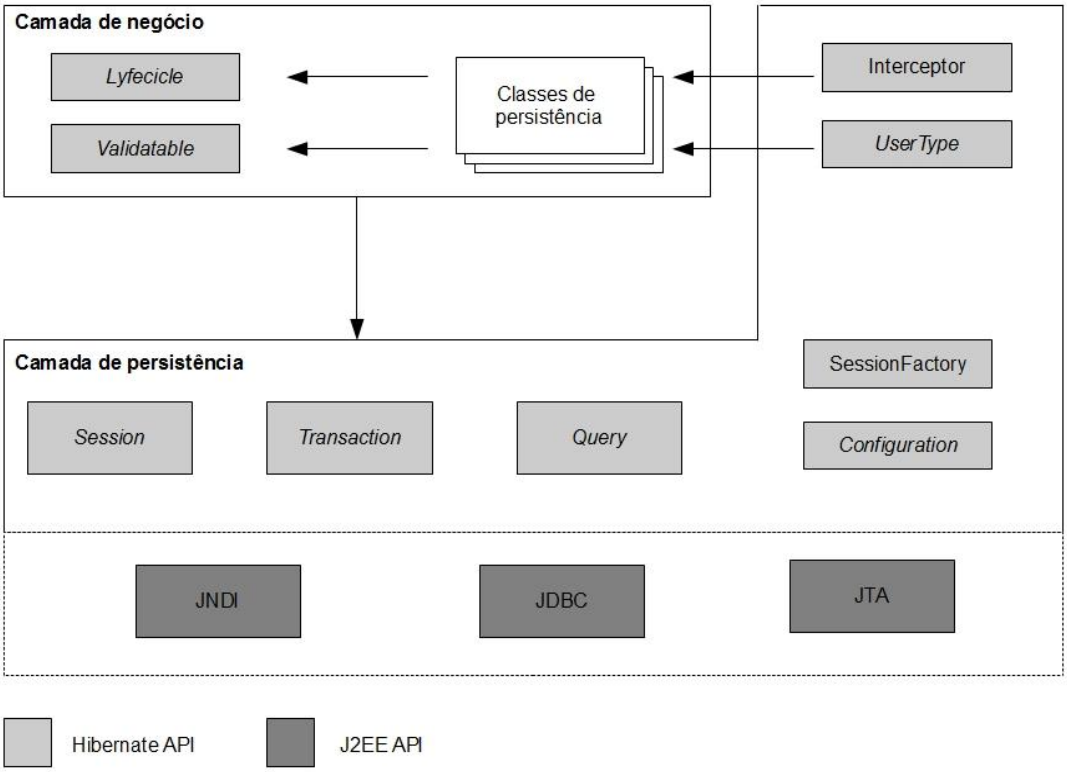

Figura 23 - Arquitectura de Hibernate - Imagem baseada em [Bauer & King, 2005]

Os principais Interfaces no Hibernate [Bauer & King, 2005]

- **Session:**

Este interface é o principal interface a ser usado em aplicações que usem Hibernate. A noção de *sessão*, em Hibernate, é algo que se encontra entre a *conexão* e a *transação*. A *sessão* pode ser vista como uma *cache* ou conjunto de objectos para um determinado trabalho, sendo que é possível, pela biblioteca, detectar alterações efectuadas nesses objectos. Assim, para um determinado trabalho/pedido, o *thread* obtém uma instância de `Session`, onde só é usada uma `Connection` (JDBC) quando estritamente necessário. Após iniciada uma `Session`, esta pode ser usada carregar e guardar objectos, tornando, quando se pretende armazenar, os dados persistentes através do método `save()`. Vejamos um exemplo, recorrendo a um objecto

`Pessoa` que mapeia directamente uma tabela `Pessoa` existente numa BD relacional podemos inserir uma nova pessoa da seguinte forma:

```
Pessoa p = new Pessoa();
p.setNome("João");
p.setIdade(21);

Session session = sessions.openSession();
Transaction t = session.beginTransaction();

session.save(p);

t.commit();
session.close;
```

- **SessionFactory**:

A forma de obter instâncias de `Session` é feita através deste interface. Este objecto é responsável pelo tratamento da conexão com uma determinada BD, sendo comum existir apenas um interface deste tipo em cada aplicação, sendo que é possível desenvolver novos interfaces caso exista a necessidade de comunicar com outras BD. Este objecto é partilhado por toda a aplicação, sendo criado durante a inicialização da aplicação, e é responsável pela criação das sessões necessárias. Este *Factory* recorre a um sistema de *cache* que gera declarações SQL e mapeamento de meta-dados em tempo de execução, potenciando a reutilização de objectos existentes na *cache* para poderem ser novamente usados.

- **Configuration**:

Este interface tem a responsabilidade de configurar o Hibernate a usar na aplicação, assim, esta última, recorre a uma instância deste interface para especificar a localização dos ficheiros que contêm o mapeamento dos objectos e também de propriedades relacionadas com a biblioteca e então cria um `SessionFactory`.

- **Transaction**:

Um interface opcional na biblioteca. As aplicações que usem esta biblioteca podem sentir a necessidade de codificar as suas próprias transacções, ignorando o uso deste interface. Um

Transaction é usado para abstrair a implementação da camada inferior de transacções, podendo tratar-se de JDBC, por exemplo. Desta forma, vemos o porquê da noção de portabilidade quando se recorre a Hibernate. Como vimos no exemplo acima referido, a abstracção do início de transacção com a BD pode ser feita através do método `session.beginTransaction();` que, no caso do JDBC, inicia uma transacção na conexão. A invocação do método `t.commit();` sincroniza o estado de `Session` com a base de dados.

Neste interface percebe-se perfeitamente a existência de uma elevada abstracção da forma como a aplicação *web* comunica com a BD.

- **Query** e **Criteria**:

O interface `Query` permite que o programador execute *queries* numa BD e controle como a *query* é executada. O Hibernate possibilita a criação de *queries* na linguagem SQL nativa da BD ou através de uma linguagem própria da biblioteca – *Hibernate Query Language* (HQL). Uma instância deste interface é usada para vincular parâmetros à *query*, limitar o número de resultados devolvidos pela *query* e executar a mesma. O interface `Criteria` é muito semelhante, permitindo ao programador criar e executar *queries* com critérios orientados ao objecto. Uma instância de `Query` não poderá ser executada fora do âmbito de um `Session` que o criou. Como exemplo, imagine-se que se quer obter os objectos `Pessoa` com idade igual a 27 anos da tabela `Pessoa` existente na BD (recorrendo a Query):

```
Query q = session.createQuery("from Pessoa p where p.idade == :idade");
q.setInteger("idade",27);
List result = q.list();
```

Através do interface `Criteria` poderíamos obter o mesmo resultado:

```
Criteria c = session.createCriteria(Pessoa.class);
c.addExpression(Expression.eq("idade",27));
List result = c.list();
```

Para ser possível executar uma *query* na linguagem nativa da BD, usar-se-ia, na mesma, a instância de `Query`, mas recorrer-se-ia a outro método:

```
Query q = session.createSQLQuery("select {p} from Pessoa {p} WHERE
p.idade == '27'");
List result = q.list();
```

Para uma aplicação poder usar Hibernate é necessário proceder a várias configurações. Como vimos, por cada BD podemos ter um `SessionFactory`. Para aceder à BD Oracle onde temos a tabela Pessoa, devemos proceder à seguinte configuração (num ficheiro xml – hibernate.cfg.xml):

```
<hibernate-configuration>
  <session-factory name="sessao">
    <property name="hibernate.dialect">
org.hibernate.dialect.OracleDialect</property>
    <property name="hibernate.connection.driver_class">
oracle.jdbc.driver.OracleDriver</property>
    <property name="hibernate.connection.url">
jdbc:oracle:thin:@127.0.0.1:1521:exemplo</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">admin</property>
    <mapping resource="exemplo/hibernate/mapping/pessoa.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

Neste ficheiro, como podemos comprovar, encontram-se as definições para a conexão com a BD, o *driver* a usar, os dados de acesso e existe também definido um recurso (através da *tag* mapping) onde se encontra o mapeamento entre a tabela `Pessoa` e o objecto `Pessoa`.

Vejamos o ficheiro de configuração para o mapeamento (pessoa.hbm.xml):

```
<hibernate-mapping>
  <class name="exemplo.Pessoa" table="pessoa">
    <id column="ID_PESSOA" name="idp" type="java.lang.Long">
      <generator class="increment"/>
    </id>
    <property column="NOME" length="100" name="nome"
type="java.lang.String"/>
    <property column="IDADE" name="idade" type="java.lang.Integer"/>
  </class>
</hibernate-mapping>
```

Alguns dos atributos encontram-se omitidos para uma melhor percepção deste ficheiro de configuração. Para uma melhor percepção de como configurar o mapeamento entre objectos e as tabelas existentes numa BD aconselha-se a leitura de [Bauer & King, 2005] .

Observando o ficheiro XML de configuração, identificamos a existência de uma *tag* identificadora da classe onde é feito o mapeamento (`exemplo.Pessoa`) assim como qual a tabela em que se encontra mapeado. As *tags* `property` identificam as colunas existentes na tabela e mapeiam directamente nos atributos do objecto.

Para os pressupostos deste trabalho, vejamos como é que o Hibernate possibilita a execução de *stored procedures* na BD. Como vimos, através do recurso a *queries* nativas podemos invocar *stored procedures*, bastando definir a *string* identificadora da *query* a executar. Mas esta solução obriga, naturalmente, à codificação de cada *stored procedure* não generalizando a forma de invocação. Até às versões anteriores à 3.x, do Hibernate, não existia forma, a não ser a descrita acima, com Hibernate3 já existe o suporte necessário para a execução de funções e *stored procedures* numa base de dados. No entanto, existem algumas limitações no seu uso [Hibernate, 2004]:

- o primeiro parâmetro a ser devolvido por uma função/*stored procedure* deverá ser um conjunto de resultados (resultset);
- para usar a *query* responsável pela execução do *stored procedure*/função, esta deverá estar mapeada através da *tag* `query` num ficheiro de configuração;
- só são devolvidos, nos *stored procedures*, escalares e entidades;
- para o uso de *stored procedures* com Hibernate é obrigatório que estes sejam executados através de `session.connection();`.
- Não pode ser feita paginação sobre os resultados através dos métodos `setFirstResult()`/`setMaxResults()`;
- a norma para invocação é a SQL92: `{ ? = call functionName(<parameters>) }` ou `{ ? = call procedureName(<parameters>}`, não é permitido o recurso à linguagem nativa.

- No caso do SGBD Oracle:
    - Uma função deve devolver um conjunto de resultados (*result set*). O primeiro parâmetro do procedimento deverá ser `OUT` que devolve o *result set*. Isto é feito através do tipo `SYS_REFCURSOR` em Oracle 9 ou 10. Na BD deve ser definido um tipo `REF CURSOR`. Podemos ver o exemplo da configuração de um stored procedure que devolve a lista de pessoas existentes na tabela. Assim, temos a definição do *stored procedure* no SGBD (Oracle, neste caso):

```
CREATE OR REPLACE PROCEDURE selectAllPessoas
    RETURN SYS_REFCURSOR
AS
    st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
 SELECT NOME, MORADA,
 IDADE, SEXO
 FROM PESSOA;
      RETURN  st_cursor;
 END;
```

Para ser usada esta *query* no Hibernate é necessário proceder à sua configuração:

```
<sql-query name="selectAllPessoas_SP" callable="true">
    <return alias="people" class="Pessoa">
        <return-property name="nome" column="NOME"/>
        <return-property name="morada" column="MORADA"/>
        <return-property name="idade" column="IDADE"/>
        <return-property name="sexo" column="SEXO"/>
    </return>
    { ? = call selectAllPessoas() }
</sql-query>
```

Deste modo, a partir da versão 3.x do Hibernate (Hibernate3) é possível proceder a uma abstracção das invocações a *stored procedures*. Naturalmente, apenas o recurso a esta funcionalidade, sem tirar partido das restantes, pode não justificar o uso da biblioteca numa aplicação *web*. Considerando ainda o facto de que muitas aplicações que não foram desenvolvidas com o recurso a esta tecnologia, pode existir uma maior entropia ao aplicá-la em aplicações já desenvolvidas.

O Hibernate, tal como os *Enterprise Java Beans*, obriga, também, que os programadores tenham de aprender as várias formas de usar a tecnologia, existindo, como é natural, uma maior curva de aprendizagem, que poderá dificultar a colocação em prática dos pressupostos da biblioteca.

Capítulo 4

4 Interfaces gráficos com o utilizador

Ideias chave:
- Evolução no desenvolvimento de GUI
- *JavaBeans*
- Apresentação de *frameworks* Java para *Web*:
  - JSF
  - Strutus2;
  - Tapestry
- Linguagens de expressão:
  - *Unified Expression Language*;
  - *Object Graph Navigation Language*;

Desde que a prestação de serviços começou a ser usada através da *web*, foi, automaticamente, identificada a necessidade dos conteúdos a disponibilizar serem dinâmicos, ao invés da estrutura estática que remonta aos primórdios da *web*. Numa fase inicial [Conallen, 1999] foram usados *Applets* para tentar criar este dinamismo necessário, sendo que a plataforma cliente era o aspecto central, onde se processavam os conteúdos de forma dinâmica, possibilitando ao utilizador a "noção" de usarem uma aplicação dinâmica. Da mesma forma e ao mesmo tempo, também foi investigada uma forma de, através da plataforma servidor, se fazer o mesmo exigindo que esse dinamismo fosse efectuado do lado do servidor. Os *scripts Common Gateway Interface (CGI)* [Fernandez, 2001] foram a principal tecnologia para gerar conteúdo dinâmico, tendo sido utilizados em larga escala um pouco por todo o lado. Mas trazia consigo algumas limitações, incluindo a dependência da plataforma onde eram usados e a falta de escalabilidade. Para superar estas

limitações foi criada uma tecnologia que providenciasse portabilidade, escalabilidade e permitisse que os conteúdos fossem disponibilizados de forma dinâmica e orientados ao utilizador. Essa tecnologia denominada de *Servlet*, é apresentada no sub-capítulo respectivo. Assim surgiu uma plataforma (ainda no domínio do POO) desenvolvida pela *Sun* que possibilitou o desenvolvimento e implementação de SI de grande porte, sendo difundidos e focalizados para a *web*. Essa tecnologia é, hoje em dia, reconhecida por J2EE ou Java EE – *Java Enterprise Edition* ([J2EE, 2007]).

4.1 Java Web Frameworks

Uma *Web Application Framework* (WAF) [Shan, 2006] , no contexto da linguagem de programação Java e da programação para *Web*, será um *software* devidamente estruturado que permite a criação de automatismos para tarefas habitualmente comuns, sendo também uma solução arquitectural "pronta a usar" onde as aplicações, aí implementadas, herdam as suas características . Ou seja, existe uma ligação intrínseca entre ambos, possibilitando que determinados elementos na parte web sejam gerados automática e dinamicamente. É uma ferramenta que permite a quem implementa uma abstracção do protocolo HTTP e mesmo da forma de tratar pedidos e direccionar respostas aos mesmos pedidos. Desta forma, existem inúmeras ferramentas, com estas características, disponíveis livremente no mercado. A escolha de uma *framework* para o desenvolvimento de um projecto de grande escala depende de vários factores [Shan, 2006] . No âmbito deste trabalho foram consideradas, numa fase inicial, aquelas que apresentam uma grande aceitação, que possuem uma comunidade consolidada e possuem margem de progressão.

Uma pesquisa, num motor de busca, por "Java web framework" apresenta um elevado número de resultados, havendo um vasto número de ferramentas ao dispor. Na tabela 5 encontram-se listadas aquelas que foram encontradas, ressalve-se que esta listagem não compreende todo o universo de frameworks existentes, trata-se apenas de uma amostra da enorme quantidade existente.

Tabela 5: Lista de *Frameworks* - Tabela baseada na listagem encontrada em [JavaWebFrameworks, 2009]

Frameworks	
Apache Coccon	Apache Struts
Struts2	AppFuse
Aranea MVC	Click Framework
Google Web Toolkit	Hamlets
ItsNat	IT Mill Toolkit
JavaServer Faces	JBoss Seam
jZeno	Lainsoft Forge
LecoWeb	OpenLaszlo
OpenXava	Play!
Reasonable Server Faces (RSF)	RIFE
Shale Framework	SmartClient
Spring Framework	Stripes
Tapestry	ThinWire
Webwork	Wicket Framework
ZK Framework	ztemplates

De entre o elevado número existente, escolher aquelas que servirão como base a uma análise pode ser, à partida, uma escolha difícil. Foi decidido reduzir a análise a apenas três frameworks, mas para atingir este número decidiu-se escolher aquelas que são mais referenciadas no mundo do desenvolvimento *web*.

As *frameworks* recorrem, principalmente, ao conceito de Java *bean* [Englander, 1997]. Cada *bean* fará o controle dos pedidos do utilizador, seja para posterior actualização do Modelo ou até mesmo para validação dos dados a inserir. Um exemplo de um bean seria o da entidade Pessoa, este *bean* tem os atributos `primeiroNome`, relativo ao primeiro nome de uma pessoa, `ultimoNome`, referente ao último nome de uma pessoa e `idade`, relacionado com a idade de uma pessoa.

Cada *bean* possui os métodos que permitem obter ou alterar estes atributos, esses métodos seguem a nomenclatura `nameXxx` onde $name \in \{get, set, has\}$ e `Xxx` é referente ao atributo em questão. Observe-se o facto de `Xxx` ter o primeiro carácter em maiúscula, este factor é importante, pois a *framework* irá aceder ao *bean* baseando-se nesta premissa, de que o acesso ao *bean* para obter o valor dos atributos ou para os alterar é feito através destes métodos que têm de estar, obrigatoriamente, implementados. O motivo para tal justifica-se através da noção de contexto. Assim, cada *framework* tem o seu contexto próprio, onde se encontram os objectos que lhe estão associados.

```
public class Pessoa{
      private String primeiroNome, ultimoNome; private int idade;
      public String getPrimeiroNome(){return this.primeiroNome;}
      public void setPrimeiroNome(String primeiroNome){
      this.primeiroNome = primeiroNome;
      }
      public int getIdade(){return this.idade;}
      public void setIdade(int idade){this.idade = idade;}
}
```

As *frameworks* recorrem a este conceito para, automaticamente, preencherem um determinado bean com dados recolhidos do utilizador ou para, a partir de um acesso à BD, apresentar os dados ao utilizador. Deste modo, sempre que existe um pedido produzido o *bean* é populado e sempre que se revela necessário apresentar uma resposta recorre-se ao *bean* para obter os valores. De um modo geral, as *frameworks*, quando pretendem aceder a valores dos beans, usam um mecanismo que permite identificar o atributo a ir buscar ou actualizar, através de `getXxx` ou `setXxx` onde `Xxx` será o atributo a identificar.

Partindo deste pressuposto, é possível, a cada *framework*, a existência de mecanismos que disponibilizem soluções, prontas a usar, para as tarefas mais comuns no desenvolvimento de aplicações para *web*, tornando a criação das páginas menos propensas ao ciclo repetitivo de codificar-*debug*-testar, fazendo apenas a reutilização desses mecanismos. Embora todas as frameworks almejem esse intuito, várias abordagens para a camada visual são possíveis.

Como referido anteriormente, as frameworks têm por base o *servlet* [J2EE, 2007], mas estendem este conceito tentando atingir mais eficiência, mais conveniência, uma maior capacidade de permitir a quem desenvolve abstrair-se dos conceitos mais próximos das camadas inferiores (como o protocolo HTTP).

O desenvolvimento através de *JavaServer Pages* (JSP) (que usa os *servlets*) por parte de muitos programadores, pode revelar uma imensa dor de cabeça se toda a camada de negócio for incluída nas páginas, fazendo com que a sua manutenção e extensão se torne um pesadelo. Deste modo, as *frameworks* vêm facilitar essa tarefa, providenciando ao programador um conjunto de ferramentas que lhe permitem desenvolver mais rápido, de forma mais eficiente e possibilitando que o código criado possa ser alterado e evoluído sem grandes problemas. Existe uma enorme disseminação de *frameworks Java* para *web*, no âmbito das estudadas revela-se necessário agrupá-las de acordo com as suas características comuns. Foram pesquisadas e analisadas aquelas que afectam directamente a camada de apresentação com o utilizador, mais propriamente aquelas que possibilitam a criação de UI. Mas, dentro deste conjunto, também existem diferenças, sendo que podemos ter dois tipos distintos de frameworks [Zoio, 2005]:

1. orientadas à operação/acção.
2. orientadas ao componente e tratamento de eventos.

Através do recurso a outras tecnologias ou até de integração mútua é possível permitir que uma framework de 1. usufrua das características de 2.

4.1.1 JavaServer Faces

A tecnologia JavaServer Faces (JSF) [J2EE, 2007] é uma *framework* orientada para componentes de interfaces com o utilizador, assim como orientada ao evento, ou seja, tudo o que possa alterar o estado de um determinado componente. É a proposta que mais se aproxima de uma norma no meio das inúmeras *frameworks* existentes, fruto do suporte proporcionado pela *Sun Microsystems* e pelo facto de já se encontrar integrada nas especificações do JavaEE.

As principais características do JSF são a existência de uma API (para representar os componentes de interface com o utilizador e controlar o seu estado) o controle e tratamento de eventos, a validação feita do lado do servidor e a conversão de dados, a definição de navegação entre páginas, tendo, também suporte a internacionalização e acessibilidade; assim como a existência de duas bibliotecas de *tags* para JSP que permitem invocar um componente de interface com o utilizador dentro de uma página JSP e conectar os componente a objectos que se encontrem do lado do servidor[J2EE, 2007]; sendo possível estender qualquer uma destas características. Deste modo surgiram, paralelamente, diversas implementações do próprio JSF, algumas com inúmeros componentes já desenvolvidos e prontos a usar. São exemplo disso o *MyFaces* [MyFaces, 2008], *ICEFaces* [ICEFaces, 2009], *RichFaces* [RichFaces, 2008], entre outros. Estas implementações do JSF visam colmatar algumas potenciais lacunas e também proporcionar acrescentos positivos à tecnologia, uma vez que a sua capacidade de permitir inúmeras extensões também permite que sejam feitas diversas implementações das especificações tecnológicas do JSF. O JSF faz uso de implementação do MVC – Model2, como se pode observar na figura 24.

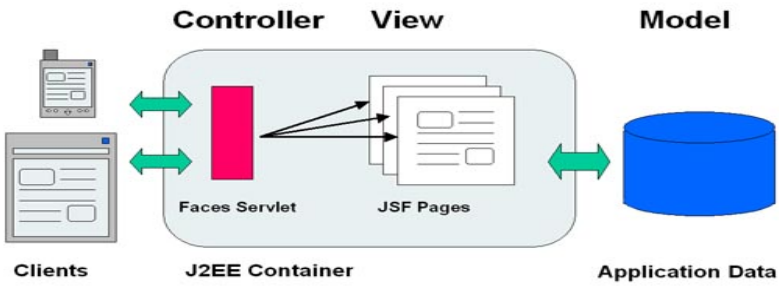

Figura 24 - Implementação de *MVC* em *JSF* - Imagem retirada de [J2EE, 2007]

Uma aplicação desenvolvida através de JSF necessita, obrigatoriamente, que todas as interacções do utilizador sejam tratadas por um *servlet* apropriado, denominado de "**FacesServlet**". Este *servlet* será responsável por receber pedidos do lado do cliente, enviá-los para o servidor e depois disponibilizar a resposta. A definição do *servlet* é feita no ficheiro *web.xml* (ficheiro responsável pelas configurações da aplicação web):

```
<!-- Faces Servlet -->
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup> 1 </load-on-startup>
    </servlet>
```

Cada pedido efectuado ao servlet da aplicação JSF tem um ciclo de vida associado, sendo este ciclo uma das principais características do JSF.

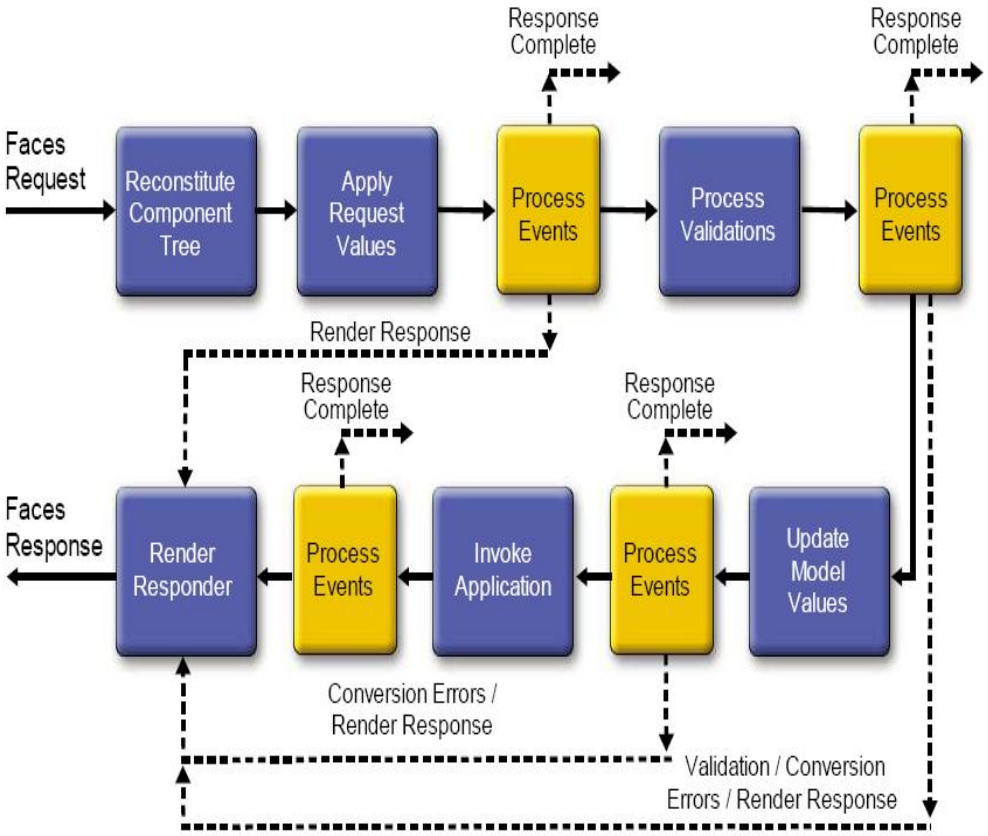

Figura 25 - Ciclo de vida – JSF - Imagem retirada de [J2EE, 2007]

O ciclo de vida de um pedido efectuado a uma aplicação JSF é composto por seis fases[J2EE, 2007]:

- **Reconstituição da árvore de componentes**

(Restore View Phase)

Nesta fase é criado um objecto, *FacesContext*, após ser efectuado um pedido do utilizador. De seguida é criada uma árvore de componentes.

- **Aplicação dos valores provenientes do pedido**

(*Apply Request Values Phase*)

Cada componente, da árvore de componentes, obtém o seu estado actual, sendo que, no final desta fase, todos os componentes são actualizados com os novos valores e as mensagens e os eventos encontram-se em fila de espera.

- **Validação *(Process Validation Phase)***

Nesta fase é realizada a validação dos valores que se encontram nos componentes.

- **Actualização do Modelo *(Update Model Values Phase)***

Após a validação ser concluída, com sucesso, os valores dos componentes são alocados aos objectos correspondentes que se encontram do lado do servidor, sendo, desta forma, feita a actualização do Modelo.

- **Invocação da Aplicação (*Invoke Application Phase*)**

Todo a lógica de negócio é aplicada nesta fase.

- **Apresentação da Resposta *(Render Response Phase)***

Por último, é apresentado o resultado ao utilizador através de renderização da página construída para o efeito.

Para se poder ter acesso aos componente de interface com o utilizador do JSF dentro de uma página JSP é necessário que seja permitido, às páginas, o acesso a bibliotecas que contêm as referências desses componentes, são as referências à biblioteca de componentes *HyperText Markup Language* (HTML) (prefixo "h") e a referência de base da tecnologia (prefixo "f"). Deste modo, é necessário fazer a seguinte declaração, na página JSP, para ter acesso aos componentes de interface com o utilizador e ao núcleo da tecnologia:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http:.//java.sun.com/jsf/core" prefix="f" %>
```

Todas as páginas desenvolvidas através de JSF são representadas de um árvore de componentes, este conceito é dos mais importantes na tecnologia JSF. A árvore de componentes é denominada por *view*, sendo que a sua *tag* é identificadora da raiz da árvore de componentes. Deste modo, todos os componentes têm de ser definidos dentro da tag que identifica a raiz:

```
<f:view>
outros componentes encontram-se definidos no corpo da tag
</f:view>
```

O acesso aos componentes, como visto anteriormente, é feito através do prefixo "h":

```
<f:view>
<h:form id="idForm">
</h:form>
</f:view>
```

Cada componente, que se encontra na árvore de componentes, tem propriedades próprias que podem ser associadas a propriedades que se encontram num *backing bean* [J2EE, 2007]. Estes objectos poderão ser responsáveis pela validação dessas mesmas propriedades. Assim e mais comumente, os métodos que um *backing bean* é capaz de usar num determinado componente são, entre outros, os seguintes:

1 Validação da informação contida no componente;
2 Tratamento de um evento accionado pelo componente;
3 Realizar o processamento necessário para determinar qual a próxima página para onde a aplicação irá direccionar.

De um modo geral, estas são as propriedades mais usadas a partir de um *backing bean*. Relembrando o exemplo da classe Pessoa, a configuração do *bean* numa aplicação JSF passa pela sua definição dentro de um ficheiro apropriado: *faces-config.xml*. Este ficheiro é responsável pela definição dos *beans* usados pela aplicação, assim como toda a navegação entre páginas dentro da aplicação. Deste modo, proceder-se-ia à definição do *bean* Pessoa da seguinte forma, no ficheiro *faces-config.xml*:

```
<managed-bean>
<managed-bean-name>PessoaBean</managed-bean-name>
<managed-bean-class>
package.Pessoa
</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Observe-se a inclusão de *scope* como sendo uma das propriedades do *managed-bean*, esta propriedade permite definir em que contexto se encontra o *bean*: "**session**", "**aplication**", "**request**".Sempre que se pretender associar propriedades a cada *managed bean* bastará incluir na definição a corresponde atribuição, suponha-se que se pretende associar a propriedade "Manuel" a primeiroNome:

```
<managed-property>
<property-name>primeiroNome</property-name>
<property-class>string</property-class>
<value>Manuel</value>
</managed-property>
```

Desta forma é possível, através da *tag* respectiva, aceder às propriedades dos *beans* que se encontrem definidos no ficheiro de configuração do JSF:

```
<h:outputText value="#{PessoaBean.primeiroNome}"/>
```

A *tag*, acima apresentada, permite que o texto referente à propriedade primeiroNome seja disponibilizado ao utilizador, deste modo a associação é feita dentro do próprio componente, através do atributo value. Observando em mais detalhe é possível identificar a existência de outros caracteres, nomeadamente é possível observar que o acesso ao *bean* e à sua propriedade é feito, apenas, através de PessoaBean.primeiroNome e encontrando-se entre **#{}**. Estes delimitadores são responsáveis pela avaliação sintáctica do seu conteúdo. Esta é uma das principais características da *Unified Expression Language (UEL)* e é algo que se pode encontrar noutras *frameworks*, mas definido de forma diferente. A UEL é responsável por:

(1) avaliação de expressões;

(2)possuir a habilidade de usar uma expressão de valor para ler/escrever dados;

(3)possibilitar o uso de métodos através das expressões.

Deste modo, dentro do contexto da *framework*, é possibilitado o acesso às propriedades dos objectos com grande facilidade. Assim se percebe que, para armazenar dados em objectos, a maior parte das *tags* do JSF usem, como atributo, o `value` nesse intuito. Algumas *tags* (principalmente as que permitem a inserção de dados por parte do utilizador) têm ainda um atributo que aceita como expressão um método que referencia um tratamento de eventos dos componente ou permita a validação ou conversão de dados.

```
<h:inputText value="#{PessoaBean.primeiroNome}"
validator="#{PessoaBean.validate}"/>
```

Como se pode observar, no exemplo anterior, para proceder à validação de dados introduzidos pelo utilizador pode-se recorrer ao atributo `validator` que irá invocar o método `validate()`, que tem de estar definido na classe, quando a expressão é avaliada, sendo identificado, no ciclo de vida do JSF, como referente à fase de validação. O JSF permite, a quem desenvolve, a criação de expressões, referentes a métodos, personalizadas, assim como conversores. Percebe-se, desde logo, a necessidade de, numa das fases, efectuar a validação dos dados introduzidos recorrendo aos métodos que se encontram definidos nos objectos que estão no servidor. Algumas *tags* permitem a inclusão de eventos *Javascript*, mas a validação feita pela *framework* só pode ser realizada recorrendo ao servidor.

A navegação entre as páginas (ou entre partes) da aplicação é definida no ficheiro de configuração *faces-config.xml*. O exemplo seguinte mostra a configuração necessária para associar uma navegação no ficheiro de configuração. Pode identificar-se, facilmente, que cada navegação existente na aplicação é identificada através de uma regra de navegação que pode ter como propriedades casos de navegação para direccionar o utilizador para uma página de sucesso, erro ou até para a mesma página de inserção de dados.

```
<navigation-rule>
        <from-view-id>/welcome.jsp</from-view-id>
        <navigation-case>
            <from-outcome>success</from-outcome>
            <to-view-id>/Pessoa.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
```

Esta contextualização com a framework JSF não pretende apresentar toda a informação sobre a mesma, é apenas feita uma introdução rápida e concisa ao JSF, mostrando como se pode configurar uma aplicação *web* com esta ferramenta. Para mais informações sobre os componentes pré-definidos, os métodos de validação presentes, conversores e outra informação relevante aconselha-se a consulta das respectivas referências bibliográficas.

4.1.2 Struts 2

A *framework* Struts 2 é uma evolução e uma combinação de duas *frameworks* que há uns anos tinham grande impacto junto da comunidade. Paralelamente existiam a *framework Struts* (pertença da *Apache Software Foundation - ASF*) e *Webwork (WW)*. *WW* era um projecto desenvolvido pela comunidade *OpenSymphony* que teve bastante aceitação junto da comunidade *Java*, particularmente junto de quem desenvolvia para *web*. Em 2007 foi decidida, após negociações entre a *OpenSymphony* e a *ASF*, a fusão entre os dois projectos, devendo-se à complementaridade que existia entre os dois projectos, deste modo proceder-se-ia à unificação do melhor dos dois mundos.

Nas próximas linhas é apresentada a *framework Struts 2*, sendo dada a devida importância aos aspectos que são provenientes de *WW* e aos que provêm de *Struts*, sendo referenciados os recursos que permitam obter mais informação sobre as mesmas. A arquitectura de *Struts2* é baseada, principalmente, no conceito de *action (*acção*)*. Ou seja, cada pedido do utilizador é considerado como sendo uma acção, sendo filtrada pela aplicação e depois executada.

O conceito de acção é a base desta *framework* orientada à acção, permitindo, desta forma, encapsular o trabalho feito para um determinado pedido [Brown, Davis & Stanlick, 2008].

Esta é, resumidamente, a forma como a *framework*, implementa o MVC:

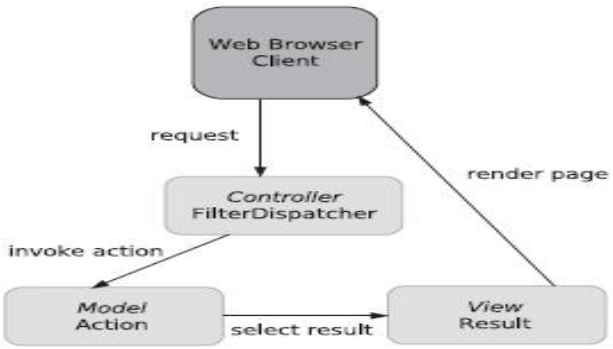

Figura 26 - MVC de Struts2 - Retirada de [Brown, Davis & Stanlick, 2008]

A forma como as acções são interceptadas é feita através de um mecanismo derivado do Webwork – *Interceptors* [Lightbody & Carreira, 2006]. Este mecanismo permite que uma acção seja interceptada para, caso exista a necessidade, ser feito o tratamento antes da execução da mesma ou até após a sua execução (como a alocação de novos recursos). Como todas as *frameworks,* possui uma abstracção do protocolo HTTP, sendo esse trabalho deixado a cargo de *servlets* apropriados.

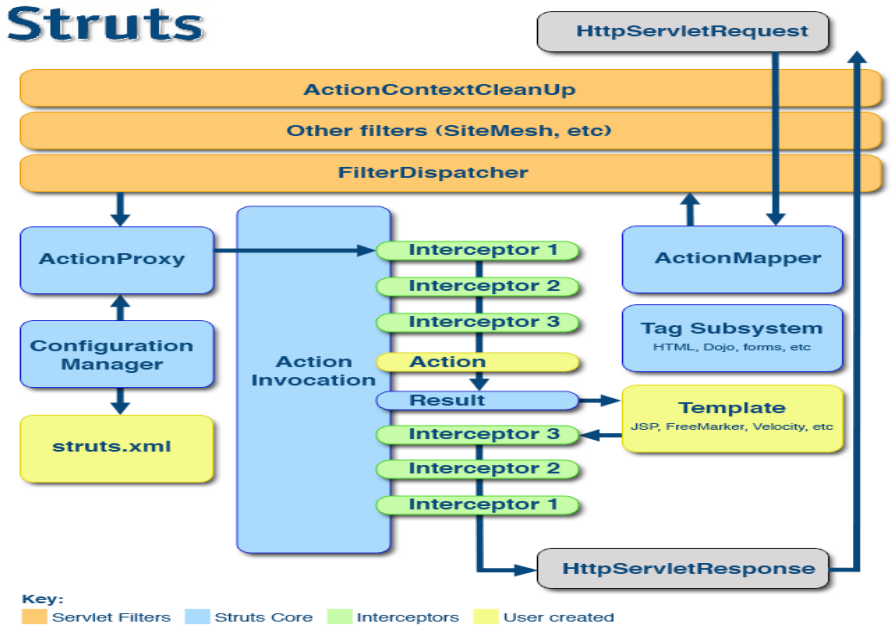

Figura 27 - Arquitectura Struts 2 - Retirado de [Brown, Davis & Stanlick, 2008]

Na figura 27 é apresentada a arquitectura do Struts 2, é perceptível, pela legenda, que a *framework* é caracterizada por filtros, ao contrário de *Struts* onde era caracterizada por *servlets*, que servem para identificar o que é relativo à *framework,* essa definição é feita no ficheiro *web.xml*:

```
<filter>
        <filter-name>struts2</filter-name>
        <filter-
class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
</filter>
<filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

Desta forma, sempre que for efectuado um pedido, o filtro actua direccionando para um mapeamento de acções, sendo filtrada a acção identificada, invocada, criado o seu resultado e enviada a resposta ao utilizador.

O programador apenas tem de definir as acções num ficheiro apropriado: *struts.xml*, este ficheiro é responsável por conter as definições de cada acção. Para se criar uma acção é necessária a implementação de uma classe que seja uma extensão da classe "ActionSupport", sendo possível identificar as classes como capazes de implementar propriedades de um *JavaBeans*. A *framework* usa, como linguagem de expressões (*Expression Language* - EL), o ***O****bject-****G****raph* ***N****avigation* ***L****anguage* – OGNL [OGNL, 2006], sendo responsável por associar os pedidos efectuados a propriedades, de*beans,* que tenham nomes iguais, este processo é feito automaticamente, permitindo ao programador abstrair-se do tratamento do pedido efectuado. As acções, por norma, possuem um método execute() que é a acção propriamente dita, ou seja, o que estiver contido nesse método é que será executado pela acção. Este é o princípio mais simples do conceito de acção. Após a implementação de uma acção é necessário que ela seja identificada como se tratando de uma acção pronta a ser invocada, para este efeito existem duas formas de o fazer:

- através da configuração de um ficheiro XML – struts.xml;
- através de anotações na classe Java responsável pela acção.

Os dois métodos podem estar presentes, ao mesmo tempo, numa aplicação desenvolvida através de Struts 2 ou até usar apenas um dos mecanismos. Segundo Brown et al [Brown, Davis & Stanlick, 2008] não existe um motivo especial para escolher qualquer dos mecanismos, sendo possível, facilmente, passar de um método para outro com a menor das dificuldades.

Observem-se as duas formas:

```
<struts>
<package name="default" extends="struts-default">
<action name="OlaMundo" class="exemplos.OlaMundo">
<result name="SUCCESS">/OlaMundo.jsp</result>
</action>
</package>
</struts>
```

```
@Result( value="/resultados/OlaMundo.jsp" )
public class OlaMundoAnotada extends ActionSupport {
/* construtores e métodos */
}
```

As duas formas de definir as acções não são exclusivas, mas, para quem desenvolve soluções, pode ser mais apelativa a configuração através de anotações, não existindo a necessidade de escrever mais ficheiros de configuração além dos estritamente necessários. O *Struts 2* implementa este conceito de "Zero Configuration" [ZeroConfiguration, 2008] possibilitando uma redução nas configurações, aumentando, desta forma, a produtividade em torno de uma aplicação. Observando, em mais detalhe, as duas formas é possível identificar um campo "Result". Este campo é responsável pela apresentação do resultado ao utilizador, ou seja, identifica, para determinados atributos, o resultado dessa acção, sendo, normalmente, um página para apresentação da informação, mas também pode ser uma outra acção ou um outro mecanismo.

É importante perceber o conceito de acção, pois toda a *framework* é baseada neste conceito. Mas convém não esquecer o conceito de *interceptor*. Um *interceptor* é a forma de "apanhar" uma acção e aplicar recursos antes ou depois da execução da mesma. A *framework* disponibiliza vários *interceptors* já criados que, segundo os autores, serão suficientes para o programador, mas é sempre possibilitada a criação de novos *interceptors*.

Exemplo de *interceptors* são a forma de capturar os parâmetros enviados pelo contexto da aplicação. O mecanismo para o uso dos *interceptors* é bastante simples, bastando definir em cada acção (no ficheiro *struts.xml*) quais os *interceptors* a serem usados.

4.1.3 Tapestry

A *framework* Tapestry caracteriza-se pela enorme diferença existente em relação a outras *frameworks* Java para *web*. De um modo geral é bastante complexo e mesmo tendo experiência com outras *frameworks* torna-se necessário um conhecimento profundo dos seus componentes e arquitectura.

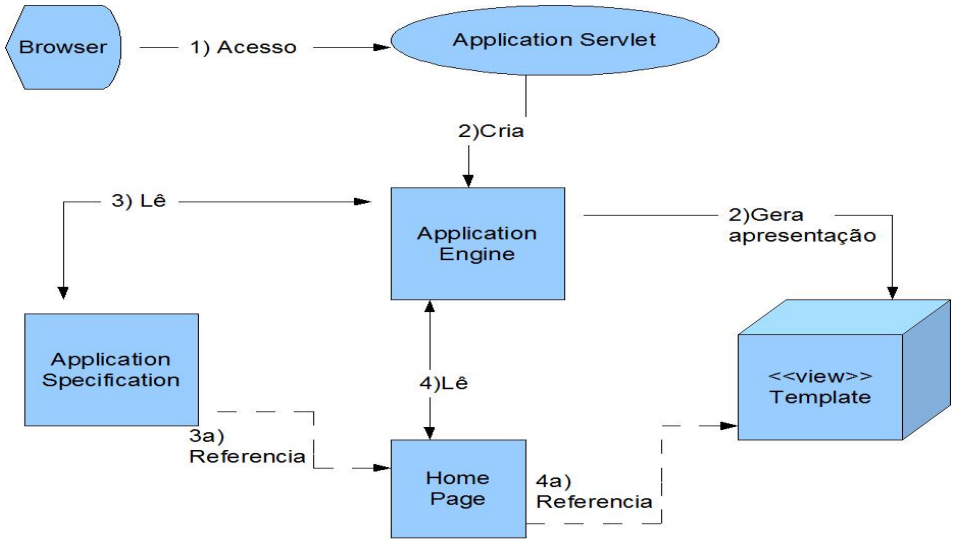

Figura 28 - Arquitectura Tapestry - Imagem baseada em [Ford, 2004]

Como se pode observar pela arquitectura do Tapestry, o princípio de comunicação com a aplicação não é estranho quando comparado com outras *frameworks*, ou seja, o acesso à aplicação é feito a partir de um s*ervlet* que irá criar um "application engine" responsável por identificar os componentes, ler as configurações para a página em questão e gerar a parte de apresentação, a visualização para o utilizador. Para cada aplicação Tapestry existe então um *servlet* responsável por

fazer a abstracção do tratamento de pedidos e respostas com o utilizador. Visto desta forma parece redutor e simplista todo o processo executado pela aplicação Tapestry, mas, de facto, não o é.

Esta ferramenta permite uma maior separação das diversas camadas. Caso exista uma equipa de design responsável pela edição do estilo da página, da alocação de interfaces simples na página, após esse design a página poderia ser enviada para uma equipa de desenvolvimento da aplicação para *web*. Neste ponto, a equipa de desenvolvimento só teria de definir os componentes a usar em cada *tag* HTML previamente "desenhada". Esses componentes terão de estar definidos num ficheiro com a extensão *.page*, pois é onde se encontram as suas definições, os acessos aos objectos *Java*, para posterior renderização da página final. Veja-se o seguinte exemplo do formulário de uma página, resultado produzida por uma equipa de design fictícia.

```
<form action="">
            Enter the secret word:
            <input type="text"/>
            <input type="submit" value="Submit">
</form>
```

É possível identificar dois componentes HTML, um de inserção de texto e outro de envio do *form* (naturalmente além do próprio *form*). A equipa de desenvolvimento recebe esta página HTML e tem de dinamizar a geração de conteúdo. Para tal recorre à definição de atributos dentro de cada *tag* HTML.

```
<form action="" jwcid="secretWordForm">
            Enter the secret word:
            <input type="text" jwcid="secretWord"/>
            <input type="submit" value="Submit">
</form>
```

Atente-se na diferença entre os dois formulários (são ambos armazenados num ficheiro com a extensão *.html*). A diferença consiste, apenas, na inclusão do atributo ***jwcid*** nos componentes HTML que serão renderizados através do Tapestry. Este atributo *Java Web Component ID* – ***jwcid***, é responsável por informar a aplicação de qual o componente, do Tapestry, a ser usado. Esses componentes têm de ser definidos num ficheiro com as especificações da página, um ficheiro que deverá ter o mesmo nome do ficheiro HTML, mas com extensão *.page*.

```
<page-specification class="classes.teste.TapestryTeste">
    <component id="secretWord" type="TextField">
        <binding name="value" value="theWord"/>
        <binding name="hidden" value="true"/>
    </component>
    <component id="secretWordForm" type="Form">
        <binding name="listener" value="listener:onWordSubmit"/>
    </component>
</page-specification>
```

Como se pode observar, pelo código apresentado anteriormente, esta especificação é baseada na construção de uma classe *Java*, neste caso é associado o valor do texto introduzido à propriedade definida na classe e também associado um *listener* de eventos a um método definido na classe.

É natural que se assuma uma enorme separação entre a camada respeitante ao controlo e a camada de visualização, nem sendo sequer abordado o modelo que lhe está associado. Embora pareça simplista, a *framework* não é de fácil aprendizagem, podendo aumentar a sua complexidade à medida que mais componentes vão sendo definidos ao longo de uma aplicação de grande porte. Através do uso de anotações, em Java, todo o trabalho de validação também passa a ser, deixado a cargo e, desenvolvido em cada classe, sendo depois associado ao respectivo componente.

Capítulo 5

5Metodologias para persistência de dados

Ideias chave:
- JSON como *Transfer Object*
- Arquitectura DAO com JSON;
- Abstracção de invocação de *stored procedures* em Oracle;
- Sistema de *cache*;
- Mapeamento objectos/dados resultantes de execução de *stored procedures*.
- *Design patterns* na asbtracção de invocações

Uma vez que já foram introduzidos e analisadas as três parte principais deste trabalho:

- *Frameworks Java* para *Web*;
- Persistência de dados e integração;
- JSON;

podemos então avançar neste capítulo para a definição de metodologias e práticas para persistência de dados em *frameworks Java* para *web* orientadas à acção através de JSON. Apresentado um conjunto de regras ou princípios que devem ser seguidos para cumprir os objectivos de:

1. usar uma arquitectura e uma implementação de JSON para Java e usar objectos JSON como "Transfer Objects";

2. obter uma forma de abstrair a invocação de *stored procedures* no SGBD Oracle através do recurso à geração de código e parametrização e tendo como base os padrões analisados;
3. obter um mapeamento entre os parâmetros usados pelo *JDBC* e os parâmetros a gerar;
4. obter um mecanismo de *cache* que permita o rápido *deploy* de implementações relacionadas com a camada de persistência de forma que não seja necessário reiniciar aplicações críticas;
5. possibilitar o recurso aos objectos JSON tanto no servidor como no cliente.

Esta abordagem considera as aplicações *web* que disponibilizam os dados de uma BD através da invocação de *stored procedures* e recorrem a um servidor aplicacional para a gestão dos pedidos e respostas ao cliente. Foi usado em ambiente de implementação o SGBG Oracle 9i, o servidor aplicacional JBoss e o Webwork 2.1.7 como *framework Java* para *web*, mas deve ser considerado que a proposta apresentada, apesar de directamente relacionada com *frameworks* orientadas à acção, poderá ser implementada noutro tipo de *framework*, sendo completamente independente da forma escolhida para implementar a camada de controle, o local onde são instanciados os *Business Objects*.

5.1 A arquitectura DAO com JSON

A base desta abordagem centra-se no padrão DAO e a forma de implementar esse mesmo padrão com recurso a JSON. Os objectos JSON podem ser usados, em desenvolvimento *web*, para enviar informação directamente para o cliente. No universo das *frameworks* Java para *web* iremos considerar o "Business Object" como se tratando das "acções"/*actions*, pois são elas as responsáveis por obter os dados de uma BD. Considerando que as *actions* obtêm os parâmetros provenientes do cliente, pode tornar-se mais prático e também boa prática aplicar o padrão "Namespace" no desenvolvimento da aplicação.

Observando que os parâmetros obtidos de um cliente, num pedido, são apenas texto (*string)* e observando que muitas vezes o envio de informação para o cliente é composto por texto (considerando que imagens e outro tipo de ficheiros encontram-se estruturados no servidor aplicacional) o recurso a JSON como "Transfer Object" apresenta-se como bastante útil.

Desta forma, a estrutura será a apresentada na figura seguinte:

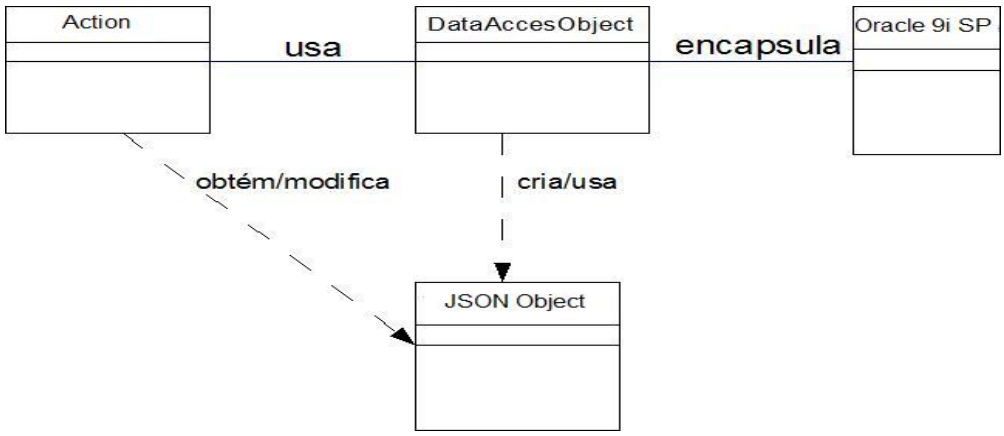

Figura 29 - Estrutura DAO proposta

Para ser possível usar um objecto JSON como "Transfer Object" é necessário que exista uma implementação em Java para o efeito. Recorrendo à biblioteca proposta em [JSONJava, 2002] podemos então ter uma representação de JSON em Java. Neste caso, já se encontram implementadas os pressupostos enunciados em 2.5.1 para a correcta estrutura, sendo fornecida uma API onde é possível fazer todas as operações relacionadas com os objectos JSON. São fornecidas as classes principais para instanciar objectos JSON. Assim, temos uma classe JSONObject (Figura 30) que possui os atributos:

- um *Map* – sendo privado;
- um *Object* NULL – sendo de acesso público;

Estes dois atributos são o aspecto central para instanciar um *JSONObject*. Como vimos antes, um objecto JSON será uma colecção, não ordenada, de pares nome/valor, assim parece trivial a escolha, para a organização *json.org*, da estrutura de dados *Map*. O atributo **null** está relacionado

com a representação de **null** em JSON. Existem vários construtores para instanciar um objecto *JSONObject*, desde o recurso a um *bean* (sendo os atributos obtidos através de *getters* e *setters* recorrendo a *Java Reflection* [Forman & Forman, 2005]) ou até mesmo através de uma *String*, desde que esteja de acordo com a especificação JSON. Os vários métodos públicos do objecto permitem fazer algumas operações sobre o objecto, como obter todas as chaves existentes no *Map* (através do método `keys()`) ou apenas os valores existentes no mesmo. Além disso, a forma de adicionar novos elementos à colecção é feita através do método `put(String key,X valor)` onde `X ∈ {boolean, java.util.Collection, double, int, long, java.util.Map, java.lang.Object}`.

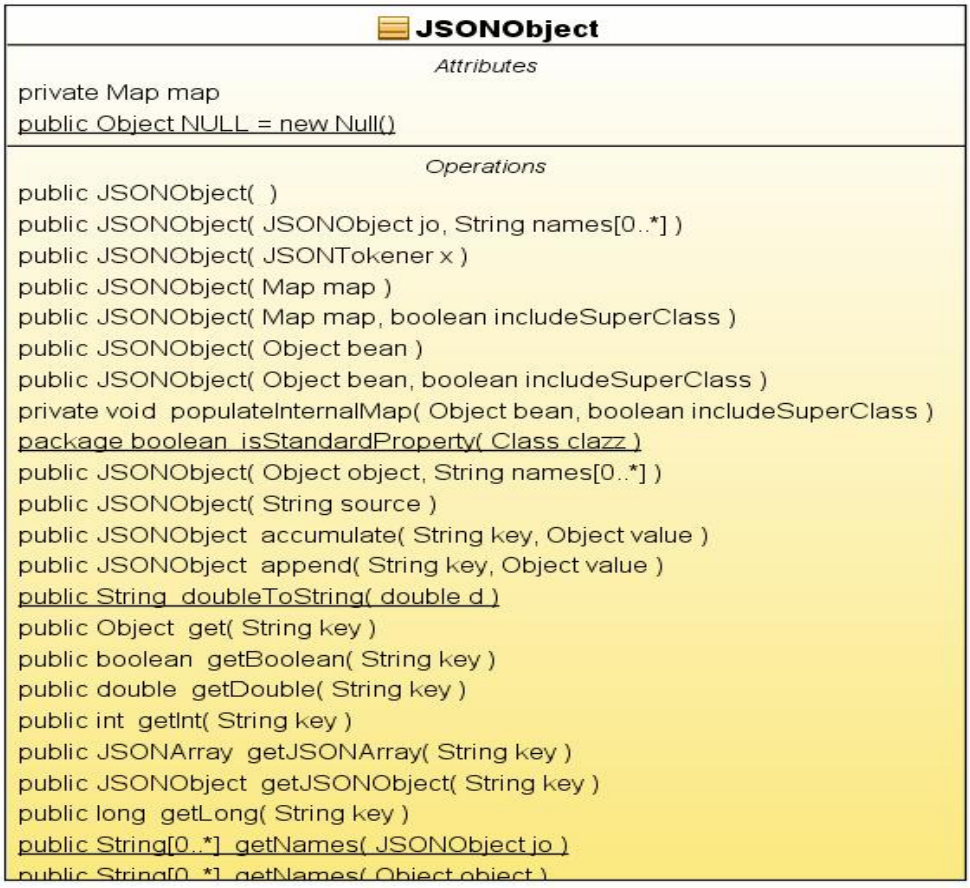

Figura 30 - Classe JSONObject

Naturalmente, também existe um classe JSONArray (Figura 29) onde se encontram implementadas as funcionalidades para este tipo de objecto. Neste caso, existe um único atributo: um ***ArrayList***. Deste modo temos uma sequência ordenada de valores, que podem ser do tipo enunciado em 2.5.1.

Figura 31 - Classe JSONArray

5.2 Stored Procedures e configuração

Para a obtenção da estrutura de um *stored procedure* optou-se por aplicar os padrões implícitos no DAO: *Factory Method* para definição de como interpretar os SP existentes na BD e *Abstract Factory* para possibilitar ao programador uma forma abstracta de invocar os SP e obter os dados. Foi tido em consideração que grande parte das aplicações recorre a POJO como "Transfer Object" e assim, também, se procedeu à devida abstracção para objectos do tipo POJO.

Para a obtenção dos parâmetros necessários, assim como da declaração da invocação de um SP, deve ser criado, também, um utilitário para a geração de um ficheiro que contivesse a estrutura do SP a invocar. Deste modo torna-se necessário inferir uma forma de obter a a assinatura de um SP, assim como o tipo dos parâmetros a enviar (caso existam) e o tipo de resultados devolvidos pela invocação do SP (caso existam). Isto pode ser obtido fazendo uma simples *query* à base de dados que nos permitirá então fazer o devido mapeamento para um ficheiro de configuração:

```
select OBJECT_NAME, OWNER from USER_ARGUMENTS where upper(OBJECT_TYPE) =
upper('PROCEDURE') order by OWNER, OBJECT_NAME
```

Esta *query*, embora ainda insuficiente, já permite obter, de uma BD Oracle, a informação de um determinado SP.

Para a definição do utilitário é necessário estabelecer alguns requisitos para a sua funcionalidade:

1. estabelecer conexão com a BD;
2. executar a *query* responsável por obter os parâmetros de entrada/saída (caso existam);
3. ler o resultado da *query* e armazenar essas definições para posterior uso;

Como se depreende dos requisitos acima descritos, torna-se evidente que existe a necessidade de armazenar o resultado obtido nalgum formato. Como veremos mais à frente, esse resultado será útil para a invocação dos SP a partir da *web*, sendo criada, assim, uma camada de persistência que é responsável pela comunicação com a BD, permitindo que os programadores se abstraiam deste facto.

Como vimos anteriormente, esta estratégia é uma das abordadas quando se implementa um DAO – o recurso a um utilitário. Para armazenar o resultado da *query* optou-se por gravar as definições num ficheiro XML, mas a forma de armazenar a informação pode ser outra desde que depois seja possível interpretar. Ao recorrer a XML temos a possibilidade de usar diversas ferramentas que possibilitam o tratamento de documentos XML.

Normalmente, recorre-se ao conceito de *package* para colocar SPs e *Functions* num SGBD. Deste modo, podemos assumir que os SP terão a seguinte estrutura:

```
package.nome_stored_procedure
```

Assim sendo, os ficheiros armazenados, com a informação relativa aos *stored procedures*, podem também ser organizados desta forma, o que possibilita um melhor controlo sobre as definições dos mesmos, permitindo centralizar a estrutura, definição da configuração e fazendo com que os programadores consigam ter um melhor acesso a estas configurações, procedendo, sempre que necessário a alterações nestes ficheiros. Com esta estrutura as possíveis alterações serão mais rápidas de realizar devido à centralização das configurações. Supondo que existe a necessidade de adicionar um parâmetro de saída num *stored procedure,* deixa de haver a necessidade de ter de se fazer essa alteração em várias localizações, desde o interface gráfico até aos vários POJOs que possam usar esse parâmetro como atributo.

Numa primeira fase, é necessário saber quais os tipos possíveis de argumentos existentes para invocar um *stored procedure*. Os tipos de dados revelam-se muito importantes, pois existirá a necessidade de mapear, nos ficheiros de configuração, esses tipos com os parâmetros a colocar como argumentos de um *stored procedure* ou nos parâmetros a interpretar como resultado de uma invocação de um *stored procedure* que devolve um conjunto heterogéneo de tipos de dados. Esses tipos deverão estar de acordo com os tipos existentes em SQL.

Recorrendo à classe `oracle.jdbc.driver.OracleTypes` podemos obter esses tipos, listados na tabela seguinte:

Tabela 6: Lista de tipos Oracle mapeados em Java

OracleTypes		Tipo de Identificador
ARRAY	BFILE	Inteiro (int)
BIGINT	BINARY	
BINARY_DOUBLE	BINARY_FLOAT	
BIT	BLOB	
BOOLEAN	CHAR	
CLOB	CURSOR	
DATALINK	DATE	
DECIMAL	DOUBLE	
FIXED_CHAR	FLOAT	
INTEGER	INTERVALDS	
INTERVALYM	JAVA_OBJECT	
JAVA_STRUCT	LONGVARBINARY	
LONGVARCHAR	NULL	
NUMBER	NUMERIC	
OPAQUE	OTHER	
PLSQL_INDEX_TABLE	RAW	
REAL	REF	
ROWID	SMALLINT	
STRUCT	TIME	
TIMESTAMP	TIMESTAMPLTZ	
TIMESTAMPTZ	TINYINT	
VARBINARY	VARCHAR	
PRIVATE_TRACE	TRACE	Booleano (boolean)
BUILD_DATE		String

Estes tipos têm um valor associado, ou seja, a forma do Java identificar de que tipo se trata é feita através do valor que o tipo tem associado, seja um inteiro, um valor lógico ou um conjunto de caracteres (*string*). A sua utilidade está relacionada com o mapeamento que será necessário fazer após a obtenção dos parâmetros através da execução da *query* que obtém os dados do SP, assim

como para estabelecer quais os argumentos necessários para a invocação. Naturalmente, poderá ser feito o mapeamento para todos os tipos existentes, sendo que procuramos apenas evidenciar dados devolvidos da BD em forma de informação (sem existir a necessidade de proceder a tratamento de dados em Java) torna-se evidente que alguns dos tipos serão mais importantes para a construção de uma configuração de um *stored procedure*.

Deste modo é necessário melhorar a *query* anteriormente apresentada para ser possível receber também o tipo de dados. Em [Web 32] temos listadas todas as colunas existentes na tabela USER_ARGUMENTS do SGBD Oracle (de acordo com o definido anteriormente, a invocação de um SP será feita através da navegação num package até se chegar ao stored procedure, como visto na estrutura definida anteriormente). Desta forma, sabendo que o objectivo é obter as colunas respeitantes a um "PROCEDURE" a seguinte query perfila-se como a ideal:

```
select PACKAGE_NAME, OBJECT_NAME, ARGUMENT_NAME, DATA_TYPE, IN_OUT from
USER_ARGUMENTS where upper(PACKAGE_NAME) = upper('PROCEDURE') order by
OBJECT_NAME
```

Recuperando o exemplo enunciado em D. PessoaDAO podemos expor o código SQL responsável pela execução do mesmo (o sp encontra-se, como se pode ver no exemplo, definido no *package* `pck_global`):

```
CREATE PROCEDURE pessoaSP(nome IN VARCHAR,morada IN VARCHAR,tlf IN INT,
tlm IN INT, nasc IN VARCHAR) AS
BEGIN
INSERT INTO PESSOA VALUES(nome, morada,tlf,tlm,nasc);
END pessoaSP;
```

Executando a *query*, definida para obtenção de SPs, para o *procedure*

```
pck_global.pessoaSP
```

obtemos toda a informação necessária para podermos proceder à configuração e respectivo armazenamento da mesma no ficheiro XML. Esta opção também se encontra relacionada com o facto de, em ambientes de produção, todos os pedidos efectuados à BD serem críticos, assim não se justificaria a criação de uma configuração "total" orientada ao objecto, seguindo, para tal, a filosofia adoptada no Hibernate, onde as configurações dos mapeamentos das tabelas se encontra

num ficheiro XML, deste modo os programadores podem fazer as alterações necessárias apenas no ficheiro XML. O resultado da execução da *query* exemplo pode ser listado na tabela 7:

Tabela 7: Exemplo de resultado de *query*

PACKAGE_NAME	OBJECT_NAME	ARGUMENT_NAME	DATA_TYPE	IN_OUT
pck_global	pessoaSP	morada	varchar	in
pck_global	pessoaSP	nasc	varchar	in
pck_global	pessoaSP	nome	varchar	in
pck_global	pessoaSP	tlf	int	in
pck_global	pessoaSP	tlm	int	in

Através do exemplo podemos observar que o resultado da execução da *query* nos possibilita saber quais os nomes dos argumentos do *stored procedure*, o tipo de cada um e a direcção do argumento (se é de entrada, saída ou ambos).

Desta forma poderemos passar para uma configuração do *stored procedure* num ficheiro XML. Sabendo as necessidades da aplicação torna-se necessário definir qual deverá ser a estrutura desse ficheiro. Recorrendo a DTD podemos definir as regras e os valores válidos para a implementação.

Como o ficheiro será usado posteriormente pela aplicação, é necessário que:

- os parâmetros (argumentos do *stored procedure*) possuam um atributo de "valor" para posterior associação, um atributo correspondente à "coluna", um ao "tipo" e outro ao "nome" (obrigatório);
- contemplar a possibilidade de não existirem parâmetros;
- deverão existir três listas, uma para cada tipo de direcção do argumento (IN, OUT, IN/OUT);
- cada lista é um conjunto de parâmetros desse tipo de direcção;
- seja identificado o *package* como um elemento possuindo um identificador único*;*
- seja identificado o *stored procedure* como um elemento possuindo um identificador único;
- um *package* poderá conter vários *stored procedures*;

Deste modo e estando estabelecidas as regras, chegamos à seguinte definição do DTD que será usado no XML para a validação do mesmo:

```
<!ELEMENT package (store_procedure*) >
<!ATTLIST package pck ID #REQUIRED>
<!ELEMENT store_procedure (paramsIN*,paramsOUT*, paramsINOUT*)>
<!ELEMENT paramsIN (param*) >
<!ELEMENT paramsOUT (param*,cursor*) >
<!ELEMENT paramsINOUT (param*,cursor*) >
<!ELEMENT param EMPTY >
<!ATTLIST param nomeArg CDATA #REQUIRED
                value CDATA #IMPLIED
                coluna CDATA #IMPLIED
                type CDATA #REQUIRED>
<!ATTLIST paramsIN>
<!ATTLIST paramsOUT>
<!ATTLIST paramsINOUT>
<!ELEMENT cursor (param*)>
<!ATTLIST cursor cursorName CDATA #REQUIRED>
<!ATTLIST store_procedure nome ID #REQUIRED>
```

Observe-se que os atributos "`value`" e "`coluna`" são facultativos, permitindo que o seu tratamento possa depois ser feito. Assim é possível configurar um conjunto de *stored procedures* num ficheiro XML que depois será lido na camada de persistência aquando da invocação do mesmo, mas, para evitar sucessivas leituras da mesma configuração, recorreu-se a um sistema de *cache,* como veremos mais à frente.

Para o exemplo acima descrito teríamos a seguinte configuração:

```
<package pck="pck_global>
<stored_procedure nome=”pessoaSP”>
<paramsIN>
<param nomeArg=”morada” type=”varchar” />
<param nomeArg=”nasc” type=”varchar”/>
<param nomeArg=”nome” type=”varchar”/>
<param nomeArg=”tlf” type=”varchar”/>
<param nomeArg=”tlm” type=”varchar”/>
</paramsIN>
</stored_procedure>
</package>
```

Existindo este mapeamento do SP em XML revela-se importante perceber como será interpretado e usado através de uma camada de persistência. A geração do(s) ficheiro(s) XML poderá ser feita de uma forma dinâmica, construindo para o efeito um utilitário que faça a leitura do resultado da

execução da *query* e construa o ficheiro de acordo com estas regras. Aconselha-se a consulta de [para obter informações sobre como construir a conexão com a BD, o recurso a SAX [SAX, 2004] para fazer o devido tratamento do ficheiro XML, assim como a sua construção.

Após a configuração de como serão representados os *stored procedures* para posterior acesso e mesmo modificações, é necessário perceber como este conceito irá ser usado para a invocação dos mesmos a partir do Java. Para tal, existe a necessidade de estabelecer uma correspondência para objectos, para permitir a abstracção desejada, fazendo com que os programadores não necessitem de codificar nada mais do que a invocação do mesmo. Como vimos nos pressupostos deste trabalho, o problema reside na manutenção do código que é feita quando se recorre a uma estratégia de implementação de um DAO por cada *stored procedure*, aqui apenas temos a configuração de cada um, algo que é feito apenas uma vez, pois sempre que forem adicionados ou removidos argumentos será possível adicionar mais um parâmetro à configuração, sem existir a necessidade de refazer o código contido na classe.

Mapeando a configuração acima descrita poderemos, de forma reduzida e para permitir a invocação do SP, ter os seguintes objectos:

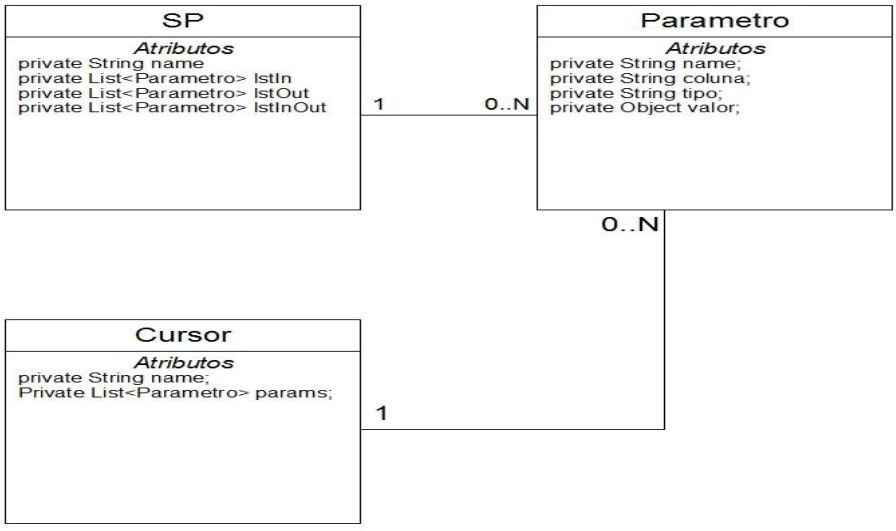

Figura 32 - Classes *SP*, *Parametro* e *Cursor*

Como é perceptível, pela figura, para cada SP teremos um conjunto, podendo ser vazio, de parâmetros, assim como o objecto `Cursor` se encontra na mesma situação, este objecto será útil quando se obtém a informação da execução de um *stored procedure*, como se verá mais à frente. Qualquer colecção de objectos, disponível na linguagem, permite colocar esses atributos na classe SP. Partindo desta estrutura, podemos passar a um sistema que permitirá generalizar a forma de invocar *stored procedures*. Numa primeira fase, revela-se necessário depreender que o sistema será usado por um *business object* que irá obter a informação desejada e manipulá-la de acordo com a sua finalidade.

Assim, temos a presença de um objecto neste sistema, o *Business Object,* que procede, quando necessário, à invocação de *stored procedures* numa BD. Como qualquer *Business Object*, poderá invocar zero ou mais SPs e estes poderão devolver dois tipos de *Transfer Object* [Gamma et al, 1994] (JSON ou POJO) proceder-se-á a uma abstracção desses dois tipos de objectos distintos através do recurso a um objecto que irá direccionar o pedido para o objecto adequado, deste modo e de acordo com esta especificidade, perfila-se como adequado recorrer ao padrão *Abstract Factory* [Gamma et al, 1994].

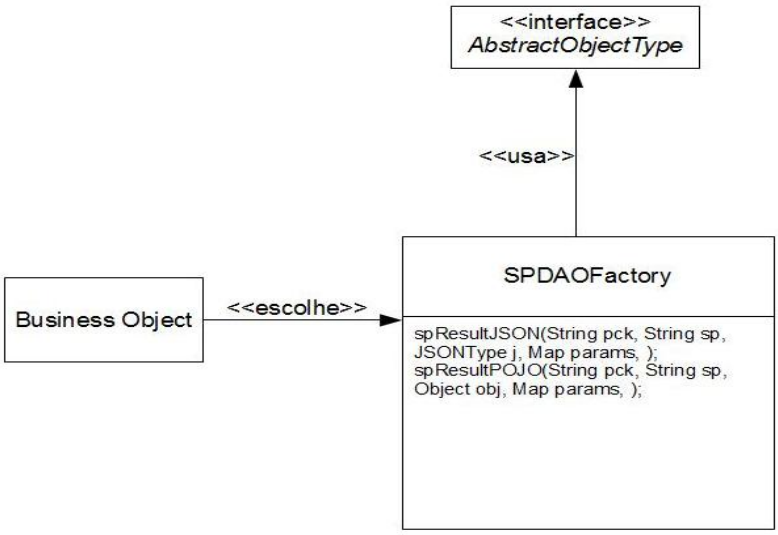

Figura 33 - *SPDAOFactory* e *AbstractObjectType*

Esta "fábrica" é responsável por diferenciar qual o tipo de objecto que é devolvido como resultado aquando da invocação do SP, possibilitando a escolha por um dos dois. Seguindo os pressupostos do tema deste trabalho, possibilita-se, também a escolha do tipo de objecto JSON a ser devolvido após a invocação do SP, deste modo possuímos outra *Abstract Factory* para separar os possíveis resultados JSON. Recorrendo a esta abstracção de alto nível é possível, posteriormente, adicionar outras estruturas que possam ser associadas ao resultado de um *stored procedure* sem afectar a arquitectura geral. Como se percebe, pela descrição, este objecto será o responsável por fazer a gestão da invocação de *stored procedures*, deste modo, por cada *web container* só será necessária a existência de uma instância deste objecto e para atingir este pressuposto podemos recorrer ao padrão *Singleton* [Gamma et al, 1994], desta forma é garantida a unicidade da classe *SPDAOFactory*, não permitindo que outras classes herdem as suas propriedades, para tal ser conseguido basta declarar a classe como `final`: `final class SPDAOFactory`

Esta classe é responsável por disponibilizar ao *Business Object* o resultado da invocação de um *stored procedure* baseado no tipo de pedido que o primeiro faz. Mas ainda não chegámos ao núcleo desta arquitectura. A *SPDAOFactory* terá a responsabilidade de usar um objecto que fará o controle de toda a estrutura interna da arquitectura:

- leitura da configuração de um SP existente num ficheiro XML e criação dos objectos necessários;
- alocação nos objectos dos valores provenientes do *Business Object;*
- instanciar um objecto responsável pela chamada do SP;
- tratamento do sistema de *cache* associado à arquitectura;

Neste ponto, temos portanto uma "caixa negra" onde se processam todos os mecanismos necessários para a execução do *stored procedure*, sendo completamente transparente ao programador a sua funcionalidade interna. Assim, podemos ver esta parte como sendo o núcleo de toda a arquitectura, pois é aqui que residem as principais funcionalidades da camada de persistência.

Deste modo podemos identificar, de acordo com a enumeração acima enunciada (Figura 34):

- um objecto que fará a leitura do ficheiro XML – **`XMLReader`**;
- um objecto que conterá o sistema de cache - **`SPCache`**;
- um objecto que criará, de acordo com o *parsing* do ficheiro XML, os objectos `SP` e `Parametro` e alocará os valores provenientes do *Business* Object nos atributos dos mesmos - **`SPMapper`**;
- um objecto responsável pela invocação do *stored procedure* – **`SPCaller`**.

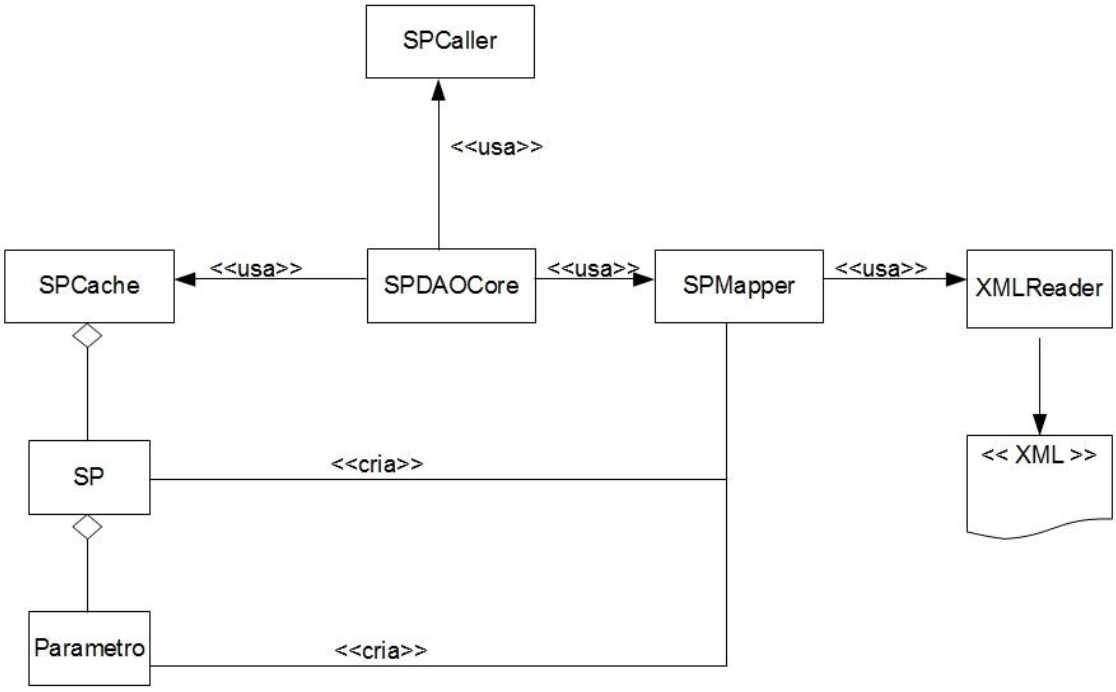

Figura 34 - *SPDAOCore*

5.3 *SPDAOCore*

O SPDAOCore funciona como um intermediário que, de acordo com o que o for disponibilizado pelo SPDAOFactory, irá instanciar todos os objectos necessários. Vejamos as partes que o compõem.

5.3.1 XMLReader

Este objecto fará o *parsing* do ficheiro que contém o *stored procedure*. Para tal, é usada a leitura do ficheiro através do *classloader* da aplicação *web*, ou seja, será procurado dentro do directório *AplicaçãoWEB\WEB-INF\classes* sendo, por norma, a localização inicial em cada aplicação. Para diferenciar aconselha-se o uso de uma directoria que contenha os ficheiros XML com a configuração dos *packages* de *stored procedures*. Deste modo garante-se a separação de outras funcionalidades existentes, sendo que qualquer estrutura pode ser seguida desde que essa mesma esteja identificada quando o objecto irá ler o ficheiros XML. Assim recorrendo ao directório *AplicaçãoWEB\WEB-INF\sps* podemos alocar todos os ficheiros gerados pelo utilitário. O formato dos ficheiros será *nome-do -package.xml*. Assim, este objecto terá como objectivos:

1. verificar a existência do ficheiro:
    1. caso não exista retorna erro;
    2. caso exista:
        - verifica se última alteração do ficheiro existe na *cache* :
            - caso leitura igual informa de que não é necessário proceder a nova leitura;
            - caso leitura actual seja diferente (superior):
                - valida o ficheiro XML de acordo com o DTD definido;
                    - caso inválido retorna erro;
                    - caso válido:
                        - lê ficheiro e faz o *parsing* do mesmo para um objecto `Document`;
                        - retorna `Document` e substitui leitura na cache.

Recorrendo a SAX será possível ler o ficheiro XML, assim como interpretá-lo e obter a informação necessária, não sendo do âmbito deste trabalho explorar a funcionalidade dos *parsers* de XML. Através do processo descrito acima, obtemos um objecto DOM contendo a existente no ficheiro XML

para posterior manipulação. Observe-se que este processo é interactivo entre os objectos `XMLReader` e `SPMapper`.

5.3.2 SPMapper

SPMapper tem como responsabilidade obter de XMLReader o documento contendo a informação de um *package* de *stored procedures* para

1. procurar um determinado s*tored procedure* no documento:
   a) caso não exista retorna erro;
   b) caso exista:
      i. cria uma instância do objecto SP;
      ii. cria listas vazias para cada tipo de argumento do SP (IN, OUT, IN/OUT);
      iii. procura as *tags* `paramsIN`, `paramsOUT` e `paramsINOUT` no documento:
         - para cada *tag* encontrada procura os nós `param` que não estejam contidos num nó `cursor` e:
            - caso existam:
               - instância, por cada `param` encontrado, um objecto `Parametro` com os atributos lidos do nó;
               - mapeia com os valores provenientes do *Business Object*;
               - adiciona `Parametro` à lista respectiva;
      iv. coloca lista no objecto `SP`.
      v. Procura as ocorrências da tag `cursor` e:
            - caso exista:
               - instância, por cada `cursor` encontrado, um objecto Cursor com os atributos lidos do nó;
               - disponibiliza o objecto para posterior uso.

2. Retorna objecto SP para SPDAOCore.

Alternativamente o objecto SP já poderá existir em SPCache e então terá a função de:

1. Obter objecto SP;
2. Mapear com os valores provenientes do *Business Object*;
3. Retorna objecto SP para SPDAOCore.

5.3.3 SPCache

O objecto SPCache é responsável pela criação de uma estrutura onde sejam alocados os objectos SP, sem os valores provenientes do Business Object, para mais fácil acesso quando invocados múltiplas vezes. Deste modo, os objectos SP ficam centralizados numa estrutura e disponíveis sempre que forem invocados pelo Business Object, proporcionando uma alternativa ao *parsing* constante dos ficheiros de configuração, o que poderia significar uma degradação no desempenho da aplicação.

Assim existem dois tipos de objectos a partilhar a mesma *cache*, os objectos SP e objectos que referenciam SP e têm a informação da altura em que foi lido o respectivo ficheiro de configuração. Para a implementação do sistema de *cache*, existem vários algoritmos de *caching* que permitem aceder aos objectos da melhor forma. Neste trabalho recorreu ao algoritmo *Least Recently Used* (LRU) que consiste em descartar, em primeiro lugar, os itens menos utilizados na escala temporal. Deste modo, os objectos que são constantemente acedidos ficam sempre disponíveis, potenciando o seu uso. Recorrendo a uma colecção de objectos do tipo chave/valor, que se encontre devidamente sincronizada para garantir "*thread safe*", podemos ir alocando os objectos à *cache* para que fiquem disponíveis caso seja necessário. Naturalmente a colecção não tem dimensão infinita e quando atingido o seu tamanho máximo deverá ser descartado o elemento menos usado antes de poder ser inserido outro. Ao colocar o objecto em *cache* será necessário identificar a chave pela qual será procurado, assim, recorrendo ao nome do *stored procedure* podemos usar a *string* que o identifica como chave do objecto SP criado. No caso do ficheiro a ler podemos seguir a

mesma metodologia, mas identificando a chave com um prefixo, sendo que o valor será o tempo, em milissegundos , da última alteração.

Um dos objectivos deste trabalho visa permitir que as alterações efectuadas na BD, a nível de *stored procedures,* sejam facilmente implementadas na *web* e também o sejam de forma dinâmica, potenciando o desempenho da equipa de desenvolvimento, evitando a necessidade de recompilar o projecto, assim como a possibilidade de visualizar as alterações de imediato. Naturalmente isso não acontecerá com POJOs usados para colocar os argumentos num *stored procedure* ou para obter a informação do mesmo, pois existe sempre a necessidade de compilar as classes e voltar a fazer *deploy* da aplicação. Recorrendo a um sistema de cache, é possível adicionar novos argumentos de entrada, saída e entrada/saída à configuração do *stored procedure* no ficheiro XML e, se os parâmetros tratados pelo Business Object estiverem devidamente tratados, verificar instantaneamente as alterações.

5.3.4 SPCaller

Este objecto tem como principal função comunicar com a BD, através de JDBC, invocar o *stored procedure* a executar e disponibilizar o resultado dessa execução. A partir do objecto SP e dos seus atributos irá gerar a invocação de uma forma dinâmica, assim como a colocação dos argumentos. O tratamento de erros permitirá saber se os argumentos se encontram correctos e se são do tipo esperado pela base de dados. A partir do Business Object temos definido qual o tipo de objecto a ser devolvido após a invocação e aqui encontra-se o pressuposto deste trabalho. Existe sempre a possibilidade de alterar o resultado da invocação de um *stored procedure* (a partir da BD, naturalmente) e recorrendo a POJOs há sempre a necessidade de, quando ocorrem essas alterações, proceder a nova codificação.

Este é o objecto central deste trabalho, sendo que é neste particular que serão apresentadas as metodologias que possibilitam a invocação de *stored procedures* de uma forma mais dinâmica.

Numa primeira fase veremos como se prepara a invocação do *stored procedure* através desta arquitectura dinâmica e posteriormente analisamos a forma de obter os resultados dessa execução.

Invocação de *Stored Procedure*

Para se proceder à invocação de um SP é necessário estabelecer uma conexão com uma BD, como enunciado anteriormente, neste trabalho são analisados os *stored procedures* do SGBD Oracle, podendo, naturalmente, ser possível generalizar a arquitectura para qualquer tipo de base de dados, para tal bastaria construir um *Factory* que tivesse a responsabilidade de escolher qual o SGBD a usar e, de acordo com tal, proceder às devidas invocações.

Recorrendo a um sistema de configuração semelhante ao de outras ferramentas de integração (Hibernate por exemplo) a configuração de acesso à BD pode ser feita num ficheiro de configuração, onde constam os dados de acesso e os *drivers* necessários para que se estabeleça a comunicação e troca de informação entre a BD e a *web*. Deste modo, existe uma camada intermédia, através de JDBC, que fará todo esse tratamento (Figura 35). Observando o anexo E. Classe DBConect podemos ver uma implementação de uma comunicação com uma BD Oracle, deste modo procede-se ao acesso a esse objecto para estabelecer a conexão e recorrendo ao objecto SP obtido pelo SPDAOCore pode ser construída invocação do SP na BD. Como o objecto SP contém as listas, de objectos, dos diversos tipos de argumento possíveis de registar na invocação (IN, OUT, IN/OUT) o processo de invocação do *stored procedures* revela-se dinâmico.

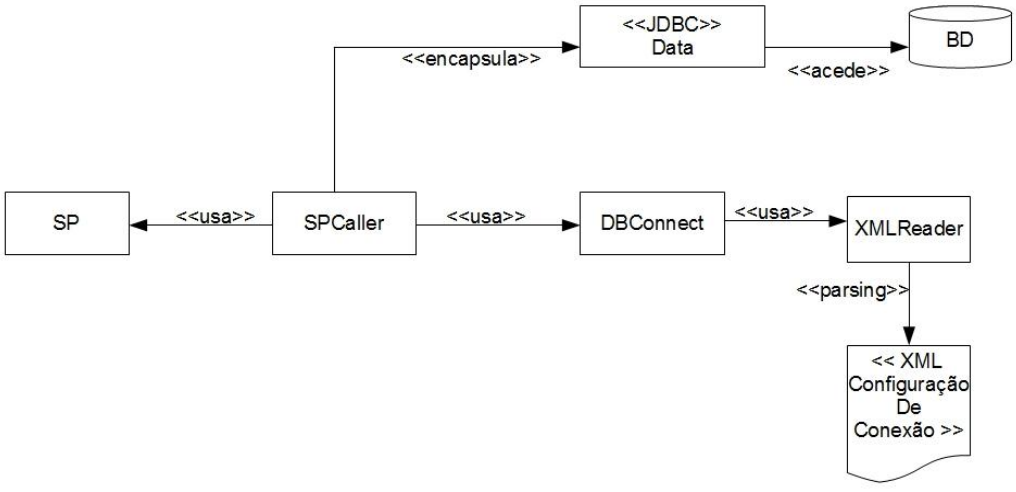

Figura 35 - *SPCaller*

Após estabelecida a ligação à BD é necessário preparar a invocação do *stored* procedure, registar os seus parâmetros e posteriormente executar. Como vimos anteriormente, a invocação do *stored procedure* faz-se através da seguinte declaração:

```
{call procedure(…)}
```

onde `(...)` pode conter zero ou mais argumentos. Assim e sabendo os parâmetros que existem em cada lista que consta do objecto SP podemos construir dinamicamente a *query* a executar. Sabendo o somatório do tamanho das três listas e que cada argumento do *stored procedure* tem de ser declarado com "?" é então possível construir a *query* responsável pela execução. Através da API do JDBC podemos preparar uma declaração a executar:

Sabendo que foi instanciada a conexão com a BD e contendo a informação dos argumentos a colocar na sua invocação, podemos criar o dinamismo que permitirá executar cada *stored procedure* sem ser necessário codificar cada invocação:

```
1. CallableStatement pstmt = null;
2. DBConnect dbconn = new DBConnect();
   Connection conn = dbconn.getConnection();
3.  String buildargs =        setNumberofArgs(sp.getParamsIn().size(),
sp.getParamsOUT().size(), sp.getParamsINOUT().size());
String prepareCallStr = "{call " + sp.getPackageName() + "." +
sp.getNomeSP() + buildargs + "}";
4. pstmt = conn.prepareCall(prepareCallStr);
5. setStatementsIn(pstmt, sp.getParamsIn());
6. setStatementsOut(pstmt, sp.getParamsOUT());
7. setStatementsInOut(pstmt, sp.getParamsINOUT());
8. pstmt.execute();
```

Observando a implementação acima podemos perceber que:

1. Recorre-se a `CallableStatement` para executar as *queries* na BD;
2. Obtém-se a conexão com a BD;
3. Recorre-se a um método que construa a string com a *query* para executar, este método construirá os diversos "?" necessários para estarem de acordo com a assinatura do *stored procedure* na BD;
4. O `CallableStatement` prepara a invocação do *stored procedure*;

5. É feita a atribuição dos parâmetros de entrada;

6. É feita a atribuição dos parâmetros de saída;

7. É feita a atribuição dos parâmetros de entrada/saída;

8. É executado o procedimento.

Naturalmente deverão existir as excepções necessárias para tornar o processo mais robusto, assinalando, sempre que necessário, os erros que acontecem durante o mesmo.

Após a instrução (4.) torna-se necessário colocar os valores, de cada argumento, na invocação a executar. Para tal pode recorrer-se a dois métodos auxiliares que serão responsáveis por atribuírem os valores dos parâmetros de entrada e registarem os parâmetros de saída, no caso de se tratar de um parâmetro IN/OUT é boa prática registar o parâmetro de saída primeiro e posteriormente atribuir o valor no parâmetro de entrada [IBM, 2005], assim será necessário um terceiro método que implementará esta prática. Estes três métodos usam o `CallableStatement` para registarem os parâmetros, assim são compostos por um ciclo que itera sobre a lista de `Parametro` e de acordo com o tipo definido no objecto fazem o registo do parâmetro.

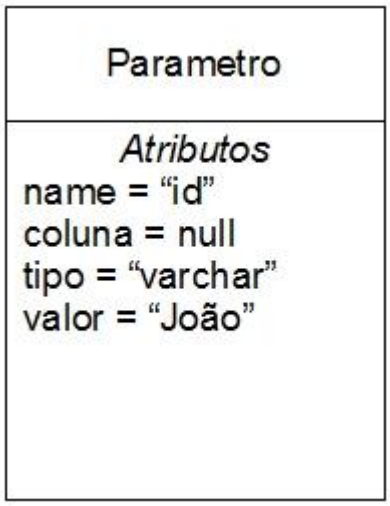

Figura 36 - Exemplo de **Parametro**

Como exemplo, podemos ter um objecto `Parametro` como o apresentado na Figura 36, assim, supondo que o parâmetro exemplo da figura faz parte da lista de parâmetros de entrada (IN) podemos exemplificar a sua atribuição:

1. através do tipo definido como atributo verifica-se a sua existência na Lista de tipos Oracle mapeados em Java;
2. colocando essa estrutura num `Enumerator` é possível recorrer à declaração `switch case` existente na linguagem podemos definir qual o tipo de parâmetro e registá-lo dessa forma:
    1. se o seu valor for nulo o objecto deve ser registado com o tipo Oracle `NULL`;
    - caso contrário deverá ser feita a sua atribuição de acordo com o tipo enumerado.

Assim, a título de exemplo, na iteração `i` da lista de parâmetros temos a seguinte implementação (faltando as devidas excepções, naturalmente):

```
Parametro p = listaParametros.get(i)
switch (TiposOracle.valueOf(p.getTipe().toUpperCase())) {
      case VARCHAR:
            if(p.getValor() == null){
            pstmt.setNull(p.getName(),OracleTypes.NULL);
            }
            else{
      pstmt.setString(p.getName(),p.getValor().toString());
}
break;
}
```

Generalizando, podemos definir a estrutura de atribuição dos valores desta forma, de igual modo podemos generalizar o registo dos parâmetros de saída, sendo que não existe a necessidade de validar o tipo null, assim, supondo que queremos registar o parâmetro `idade` temos:

```
pstmt.registerOutParameter("idade", OracleTypes.INTEGER);
```

No caso da lista de parâmetros IN/OUT podemos usar os dois métodos seguindo a boa prática indicada.

Após estas instruções é executado o *stored procedure* através do `CallableStatement` (8.).

Resultados da execução

O resultado da execução de um *stored procedure* pode não devolver nenhum resultado, para tal bastará verificar o tamanho das listas OUT e IN/OUT, caso se encontrem vazias não existe a necessidade de continuar, devendo ser fechado o `CallableStatement` e a conexão com a BD. Assim, sabendo que a lista de parâmetros de saída (tal como a lista de entrada/saída) contém parâmetros é necessário obter o resultado da execução do *stored procedure*. De acordo com o que foi definido como tipo de resultado podemos ter então dois tipos: POJO e JSON. O resultado da execução de uma query à BD pode ser obtido através do recurso ao interface `ResultSet` [ResultSet, 2003] caso se trate de uma lista de resultados a devolver, podendo ser visto como uma tabela de resultados. Deste modo, podemos iterar sobre as linhas do resultado e alocar os dados em formato apropriado, normalmente, recorre-se ao tipo (listado na Tabela 6) `Cursor` para obter este tipo de resultado, sendo também este um dos pontos centrais deste trabalho, a forma como alocar os dados existentes num objecto deste tipo. Deste modo, é perceptível, agora, a existência do atributo `coluna` no objecto `Parametro`, sendo o atributo a usar para obter uma determinada coluna de uma linha de um `ResultSet`. Para obter os outros tipos de objectos usar-se-á o mesmo método apresentado anteriormente, recorrendo ao atributo `name` do objecto `Parametro`. Assim, unindo as listas de saída (OUT) e de entrada/saída (IN/OUT) podemos obter uma lista com os parâmetros a obter do resultado da execução do *stored procedure*. Deste modo, é necessário saber que tipo de parâmetro se está a ler do resultado da execução. Para tal, é prática comum recorrer a POJOs para fazer a atribuição dos valores resultantes, sejam do tipo `Cursor` ou de qualquer outro tipo, desde que tal seja possível. Assim, sabendo o tipo do parâmetro, já previamente definido, é possível saber exactamente que tipo de objecto se estará a obter do resultado. Revela-se então a possibilidade de alternar o tipo de resultado a ler, assim como o método a invocar. Vejamos através de um exemplo onde se recorre à definição do *stored procedure* apresentada neste trabalho:

```
<package pck="pck_aluno>
<stored_procedure nome=”discip_media”>
<paramsIN>
<param nomeArg=”nome” type=”varchar” />
</paramsIN>
<paramsOUT>
<param nomeArg=”idade” type=”integer”/>
<param nomeArg=”curso” type=”varchar”/>
<param nomeArg=”media_notas” type=”float”/>
<param nomeArg=”discip_efect” type=”cursor”/>
<param nomeArg=”discip_por_efect” type=”cursor”/>
</paramsOUT>
</stored_procedure>
</package>
```

Observando a configuração do *stored procedure* `pck_aluno.discip_media` percebemos que existe um parâmetro de entrada e cinco de saída, sendo alguns de diferentes tipos. O processo habitual implica o recurso a um POJO Aluno para armazenar a informação da execução (Figura 37). Para termos um objecto deste tipo bastaria definir como atributos os que se encontram listados na configuração do *stored procedure* com particular interesse para os `type="cursor"` que ficarão como conjuntos de disciplinas, sendo que os elementos destes últimos também poderão ser definidos como POJO. Este exemplo é muito comum no dia-a-dia de programadores *web* e não só. Devemos notar que são necessários vários objectos (neste caso dois) para proceder à alocação dos valores resultantes da execução do *stored procedure*. Assim, analisemos como poderá esta estrutura dinamizar também este processo, a colocação dos valores provenientes da BD nos atributos do POJO.

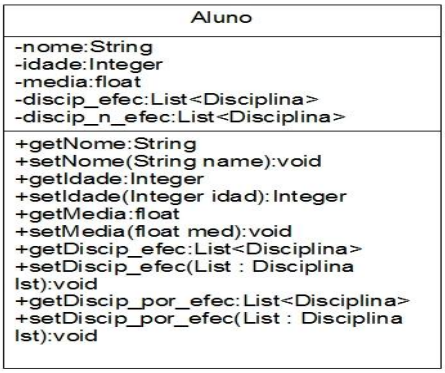

Figura 37 - POJO Aluno

Resultados com POJO

Deste modo, após a execução do *stored procedure* devem ser obtidos os objectos para posterior uso, no caso deste exemplo e como vimos pela configuração do SP poderemos (recorrendo à API do JDBC) obter esses valores através dos nomes dos argumentos/parâmetros listados na configuração. Então, para o exemplo em questão:

```
int idade = pstmt.getInt("idade");
String curso = pstmt.getString("curso");
float media_notas = pstmt.getInt("media_notas");
```

Permite-nos obter os objectos após a execução procedendo à colocação dos mesmos nos atributos do POJO `Aluno`:

```
Aluno aluno = new Aluno();
aluno.setIdade(idade);
aluno.setCurso(curso);
aluno.setMedia_notas(media_notas);
```

Antes de passarmos à análise do tipo `cursor`, devemos considerar a possibilidade de tornar este processo mais generalizado, tratando-se de POJO. O programador tem, habitualmente, a necessidade de proceder à codificação acima representada, neste caso o `SPCaller` recebe do `SPDAOCore` o POJO a instanciar e através de Java Reflection [Forman & Forman, 2005] irá colocar os valores nos atributos do objecto:

```
Object temp = new Object();
Class klass = object.getClass();
Method[] methods = klass.getMethods();
for(int i=0;i<paramsOUT.size();i++){
Parametro p = (Parametro) paramsOUT.get(i);
try {
temp = (Object) pstm.getObject(p.getNomeArg());
Method met = getMethod(methods, p.getNomeArg());
setValor(o, met, temp, p.getType());
}
}
```

Assim temos a forma de generalizar a atribuição dos valores provenientes da BD em objectos POJO. De realçar que existem, no extracto de código acima, dois métodos que não estão aqui enunciados.

O método `getMethod` é responsável por encontrar, na lista de métodos do objecto POJO, o correspondente ao nome do argumento da iteração actual, enquanto o método `setValor` é responsável, pelos princípios de Java Reflection, pela colocação, no atributo respectivo, do objecto proveniente da BD. Como os parâmetros do tipo cursor são uma listagem devolvida pela BD, também se poderá proceder do mesmo modo, sabendo, à partida, que um cursor será instanciado como um objecto `List`. De certo modo, observando a definição das regras de construção do ficheiro de configuração (DTD), observamos a existência de uma atribuição para o elemento `paramsOUT` como contendo zero ou mais `param` e também zero ou mais `cursor`:

```
<!ELEMENT paramsOUT (param*,cursor*) >
```

Sendo que também existe a definição da respectiva *tag* :

```
<!ELEMENT cursor (param*)>
<!ATTLIST cursor cursorName CDATA #REQUIRED>
```

Assim possuímos uma *tag* que poderá conter os parâmetros a ler do `ResultSet`, nesta definição assume-se que o programador irá proceder a essa codificação, pois só é possível saber as colunas existentes num cursor aquando da execução do procedimento que levou à sua existência. Percebe-se agora a existência do atributo `nomeColuna` no objecto `Parametro`, deste modo é possível, quando se percorre uma linha de um `ResultSet` saber qual a coluna a obter. Assim, retomando a configuração anteriormente definida, poderemos passar a ter a seguinte configuração para o *stored procedure* exemplo:

```
<package pck="pck_aluno>
<stored_procedure nome=”discip_media”>
<paramsIN>
<param nomeArg=”nome” type=”varchar” />
</paramsIN>
<paramsOUT>
<param nomeArg=”idade” type=”integer”/>
<param nomeArg=”curso” type=”varchar”/>
<param nomeArg=”media_notas” type=”float”/>
<cursor cursorName=”discip_efect”>
<param nomeArg=”nome” coluna=”discip” type=”varchar”/>
<param nomeArg=”professor” coluna=”prof” type=”varchar”/>
</cursor>
<cursor cursorName=”discip_por_efect”>
```

```
<param nomeArg="nome" coluna="discip" type="varchar"/>
<param nomeArg="professor" coluna="prof" type="varchar"/>
</cursor>
</paramsOUT>
</stored_procedure>
</package>
```

Deste modo, podemos proceder à mesma metodologia aplicada quando obtemos um parâmetro do *statement* executado, com a condicionante de existir a necessidade de verificar se se trata de um ResultSet e assim poder ser percorrido, para tal é necessário quando se itera sobre os objectos existentes, verificar se estamos na presença de um `ResultSet`.

```
Cursor k = (Cursor) listaCursors.get(i);
ResultSet rs = null;
Object obj = null;
obj = pstmt.getObject(k.getCursorName());
if(obj instanceof ResultSet){
rs = obj;
while(rs.next()){
/* aplicar a metodologia através de Java Reflection para os parâmetros
que contêm o atributo coluna diferente de null */
/* é aconselhável quando se faz a primeira iteração sobre a lista de
parâmetros de saída fazer esta verificação e alocar esses objectos numa
lista auxiliar, diminuindo o número de acessos à lista inicial */
}
}
```

Assim apresentou-se uma forma mais genérica de obter o resultado da invocação de um *stored procedure*, quando existente, e colocar a informação em POJO. Vejamos como podemos atingir o mesmo objectivo (disponibilizar a informação) recorrendo a objectos JSON.

Resultados com JSON

Mantendo a mesma forma de invocação de *stored procedures* resta saber como proceder à alocação da informação obtida em objectos JSON. Já vimos que esta estrutura é caracterizada por possuir um objecto JSONObject, sendo um conjunto de pares/valor e JSONArray para um conjunto não ordenado de objectos. Assim, retomando o exemplo anterior teremos para a iteração dos parâmetros de saída após a execução do *stored procedure*:

```
JSONObject json= new JSONObject();
for(int i=0;i<paramsOUT.size();i++){
Parametro p = (Parametro) paramsOUT.get(i);
try {
temp = (Object) pstm.getObject(p.getNomeArg());
json.put(p.getNomeArg(),temp.toString());
}
}
```

Esta estrutura, naturalmente, não será viável para todos os tipos de objectos devolvidos pela execução, mas, na maioria dos casos, é devolvida informação para disponibilizar ao utilizador (função do *Business Object*). Deste modo e recorrendo à estratégia usada na atribuição dos valores de entrada do *stored procedure* poderemos usar a declaração `switch` para validar só determinados tipos Oracle. Assim, sabemos que iremos usar este tipo de resultado em situações em que é devolvida apenas informação em formato de texto, contendo números reais ou inteiros, informação passível de surgir ao utilizador. Retomando o exemplo anterior teremos então a seguinte implementação:

```
JSONObject json= new JSONObject();
for(int i=0;i<paramsOUT.size();i++){
Parametro p = listaParametros.get(i)
switch (TiposOracle.valueOf(p.getTipe().toUpperCase())) {
      case VARCHAR:
      try {
      temp = (String) pstm.getString(p.getNomeArg());
      json.put(p.getNomeArg(),temp);
      }catch(Exception e){/*faz tratamento de excepção*/}
      break;
      /* outros case possíveis */
}
}
```

Para o caso de nos encontrarmos na presença de um cursor devemos, como vimos anteriormente, verificar se se trata de um `ResultSet`:

```
Cursor k = (Cursor) listaCursors.get(i);
ResultSet rs = null;
Object obj = null;
obj = pstmt.getObject(k.getCursorName());
if(obj instanceof ResultSet){
rs = obj;
while(rs.next()){
```

```
     /* enquanto iteramos sobre o resultset podemos aplicar a
estratégia referida na implementação anterior, percorrendo as colunas da
linha através dos nomes da coluna existente em cada objecto Parametro */
}
}
}
```

Como observamos, nos dois casos, a obtenção dos valores existentes no cursor é feita com o recurso ao objecto `Parametro` que contém o atributo coluna com um valor diferente de `null`. Deste modo existe sempre uma dependência do conhecimento das colunas existentes no cursor por parte da equipa de desenvolvimento *web*, obrigando a uma elevada interacção com a equipa de desenvolvimento de BD. Alternativamente, poderemos obter os meta-dados do cursor e, a partir daí, obter a informação das colunas de cada linha iterada no `ResultSet`, assim, através do uso de uma lista contendo os nomes das colunas do ResultSet podemos obter, facilmente, a informação:

```
/* outros métodos auxiliares */
private JSONArray getInfoFromResultSet(ResultSet r) throws SQLException{
        JSONArray jarr = new JSONArray();
         ResultSetMetaData rmeta = r.getMetaData();
         int numeroColunas = rmeta.getColumnCount();
         for (int i = 1; i <= numeroColunas; i += 1) {
             JSONObject jsonRS = new JSONObject();
             String tmp = rmeta.getColumnName(i);
             Object coluna = r.getObject(tmp);
                if(col != null){
                      jsonRS.put(tmp,coluna);
                  }
                  else{
                      jsonRS.put(JSONObject.NULL);
                  }
                   jarr.put(jsonRS)
          }
          return jarr;
}
/* outros métodos auxiliares */
```

Desta forma, é devolvido um objecto JSONArray contendo, em número igual de linhas do ResultSet, JSONObjects com a informação de cada linha, estando cada coluna referenciada como sendo a chave e o objecto obtido do ResultSet (para essa coluna) sendo o valor do par chave/valor. Naturalmente também poderíamos proceder à construção desta iteração, sobre os meta-dados,

através da construção de um JSONObject por cada linha a iterar e colocar, como pares chave/valor, um JSONObject contendo o nome_da_coluna/valor_da_coluna nesse objecto.

Business Object e os resultados

Após todo este processo, é devolvido ao *Business Object* um objecto contendo a informação, os dados que serão disponibilizados no interface gráfico com o utilizador. Observe-se que em nenhum lado se abordou a validação dos dados enviados/recebidos. Esse não é o objectivo desta arquitectura, nem o deve pretende ser, a arquitectura deverá ser o mais inócua possível dentro de uma aplicação *web*, não devendo conter "inteligência" associada ao tratamento dos dados. Essa responsabilidade está ao acordo do *Business Object* e dos *stored procedures* do SGBD. Os objectos recebidos, no *Business Object*, pela execução do *stored procedure* poderão ser colocados em HTML da mesma forma que outros objectos o são, não sendo necessário qualquer codificação extra além da obtenção do valor para uma determinada chave.

Como vimos, esta arquitectura possibilita, ao programador, a obtenção de dados de um *stored procedure* através de POJO ou JSON. Naturalmente, ficará a cargo do mesmo a decisão de escolher qual o que se adequa mais à situação em questão. Em *frameworks Java* para *web*, como vimos no capítulo 4, orientadas à acção, o *Business Object*, no caso de Struts2, é um objecto "Action", sendo ele o responsável por direccionar o pedido efectuado, tratar e enviar a resposta. Neste caso particular, pode existir a possibilidade, caso a *framework* o permita, de enviar o resultado directamente para o utilizador em formato JSON e proceder ao tratamento dos através do processamento do objecto. Como a abstracção atingida permite que a arquitectura possa ser colocada em qualquer aplicação *web* desenvolvida com Java, observa-se que pode ser usada por qualquer tipo de *framework* sem existir a necessidade de proceder a qualquer tipo de codificação além da configuração de acesso à BD Oracle que contém os *stored procedures*.

Normalmente, os *Business Objects* têm implementadas interfaces que permitem obter os dados enviados pelo utilizador (a abstracção do tratamento do pedido) através do *browser*, sendo prática

comum, como vimos também no capítulo 4, o recurso a *getters* e *setters* para alocar esses valores como atributos do objecto. Assim, os parâmetros enviados pelo utilizador estarão numa estrutura do tipo chave/valor, sendo colocados no objecto através da "descoberta" do método *set* que corresponde à chave em questão. Muitas das validações são feitas ainda no cliente, através de *Javascript*, existindo para o efeito inúmeras ferramentas que facilitam a codificação das mesmas.

Nestes casos e recorrendo a JSON como *Transfer Object*, o envio dos parâmetros para o Business Object não necessita da colocação dos *setters* identificativos de cada parâmetro, bastando, para tal, colocar a colecção de parâmetros na arquitectura, desde que os parâmetros estejam de acordo com o ficheiro de configuração. Deste modo, potencia-se a uniformização de nomes de parâmetros/argumentos, facilitando a interpretação do código, assim como a análise feita por um, potencial, novo elemento da equipa.

Deste modo temos uma arquitectura/metodologia suficientemente dinâmica para possibilitar aos programadores uma melhor gestão do tempo através da não repetição de código, nem da criação de inúmeros objectos/classes que podem propagar eventuais erros de codificação.

A arquitectura de persistência em *stored procedures* com POJO e JSON

Após a apresentação das metodologias, é altura de agrupar as várias decisões numa arquitectura geral, passível de ser usada em qualquer projecto *web* que recorra a *stored procedures* para a manipulação da informação. Este tipo de aplicações normalmente possui as regras de negócio e do tratamento da informação implementados no SGBD devido a decisões arquitecturais tomadas no início do projecto, não sendo de todo impossível, nestes casos, aplicar a arquitectura aqui apresentada.

É então importante perceber como fica a arquitectura proposta para procedermos à avaliação da mesma:

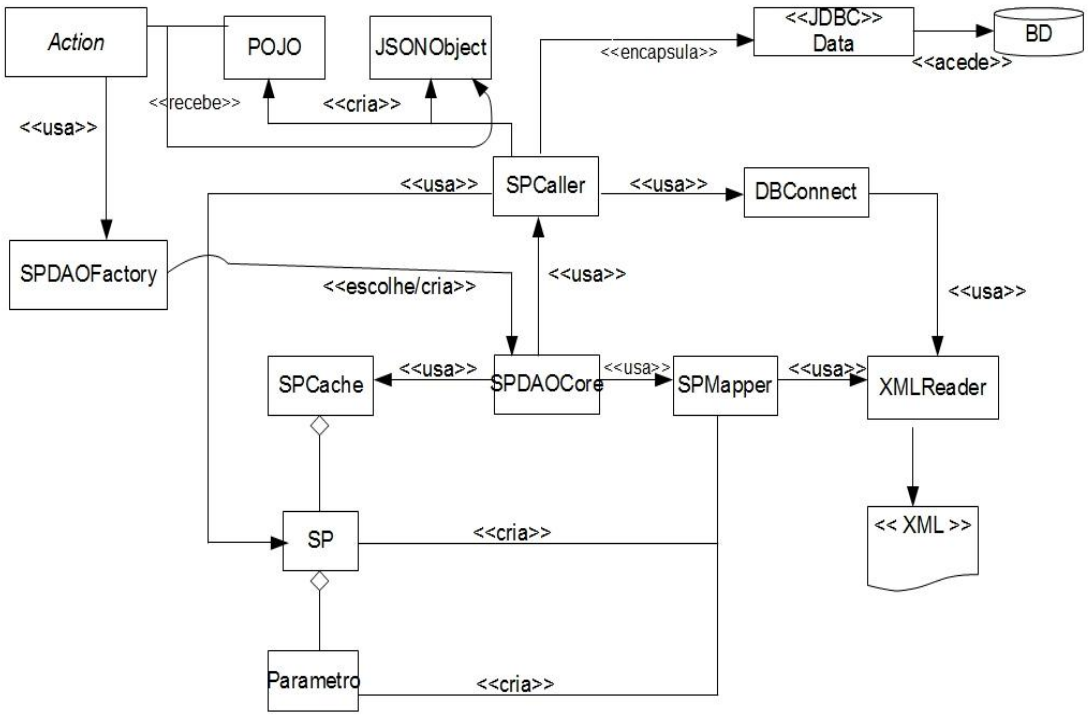

Figura 38 - Arquitectura de persistência

5.4 Avaliação da Metodologia

Tratando-se de uma metodologia com aplicação directa e existindo outras soluções possíveis, alguns aspectos relacionados com a avaliação da metodologia foram sendo apresentados ao longo do documento através de exemplos. Quando confrontada com o uso desta tecnologia, normalmente, a gestão de um projecto interroga-se sempre sobre o custo de uma implementação de uma arquitectura de persistência ou mesmo do recurso a uma ferramenta que faça essa integração. Observando a codificação manual de cada *stored procedure,* existente na BD, para um respectivo objecto que fará a invocação a partir de Java, percebe-se imediatamente que a arquitectura aqui apresentada será muito mais fácil de manter e também mais eficiente e menos propícia a erros, senão vejamos:

No problema apresentado neste trabalho (um DAO por cada *stored procedure*):

- o recurso a um DAO por *stored procedure* implica a codificação específica de cada DAO;
- o uso de POJO para alocação dos resultados existentes num cursor implica que, de acordo com o número de possíveis cursores exista um número equivalente de POJOs codificados em Java;
- a manutenção do código deste tipo de solução aumenta a possibilidade de existirem erros e da propagação dos mesmos, pois desde os interfaces gráficos até aos POJOs resultado da iteração sobre um cursor, sempre que existir a necessidade de proceder a alterações no código, esta terá de ser efectuada em diversos ficheiros, sendo uma tarefa fastidiosa e longa;
- novos programadores inseridos num projecto que recorra a este método sentirão muita dificuldade em integrar novas funcionalidades devido à inúmera propagação de objectos;
- sempre que é efectuada uma alteração num POJO ou objecto DAO responsável pela invocação de um *stored procedure* existe a necessidade de fazer *re-deploy* da aplicação;
- o envio de um objecto JSON contendo a informação obtida pela BD obriga à codificação do mesmo;
- a implementação desta solução é directa, não sendo necessário grandes conhecimentos de persistência de dados, o que potencia uma redução de custo a curto prazo;

Através do recurso às metodologias aqui apresentadas:

- existe um único local onde os *stored procedures* estão configurados e de fácil acesso e manutenção do código (os ficheiros de configuração XML);
- não há a necessidade de codificar cada *stored procedure*, sendo essa tarefa abstraída pela arquitectura;
- o recurso a JSON como resultado possibilita que os objectos sejam enviados directamente para o *browser* do cliente;

- o recurso ao sistema de *cache* permite que não exista a necessidade fazer *re-deploy* mesmo que os argumentos do *stored procedure* tenham sido alterados, bastando, para tal, editar o ficheiro de configuração;
- o código é mais fácil de manter, pois as alterações a efectuar são em menor número e independentes da arquitectura;
- recorrendo à colocação de nomes iguais aos da configuração nos parâmetros recebidos pelo *Business Object* não existe a necessidade de codificar *setters* e *getters* no objecto;
- a codificação efectuada pelos programadores e a posterior manutenção do código torna-se menos enfadonha e mais eficiente, com menos erros e menos propagação dos mesmos;
- a implementação de uma solução deste género não será trivial, podendo, numa fase inicial, aumentar o custo com a sua aplicação;

Capítulo 6

6Conclusões e Trabalho Futuro

Ideias chave:
- Dificuldades enfrentadas;
- *Cache* em *deploy;*
- Disponibilidade em *web*;
- Ferramentas e bibliotecas;
- Implementação e uso;

6.1 Avaliação das etapas do trabalho

Nesta secção são avaliadas as etapas, opções e suas percussões, que permitem retirar diversas conclusões detalhadas sobre o desenrolar deste trabalho. Esta avaliação é feita através da descrição do processo que originou a criação de cada capítulo deste documento.

Capítulo 1

Neste capítulo é abordada e introduzida a problemática existente no desenvolvimento de aplicações *web* que recorrem a *stored procedures* para armazenar os dados. É o ponto de partida para este documento, onde se apresenta, de forma sucinta, as características que compõem este documento, a sua estrutura e, principalmente, o seu objectivo.

Capítulo 2

Deste modo, foi feito um estudo sobre metodologias e arquitecturas que correspondessem à necessidade deste trabalho. Numa primeira fase foram identificados diversos desenhos padrão, capazes de resolver um conjunto de problemas e, quando agrupados, fornecessem uma arquitectura padrão para um determinado problema. Assim, avaliando a arquitectura em três camadas, foi necessário perceber o seu funcionamento, assim como perceber o seu enquadramento em soluções J2EE, deste modo fez-se um estudo da arquitectura mais referenciada para aplicações *web* – MVC, tendo sido analisados alguns dos padrões usados para a sua implementação. , como tal procedeu-se a um estudo de J2EE e alguns dos seus padrões e a forma de os implementar.

Paralelamente, foram estudados os formatos de interpretação disponíveis nos *browsers* actuais, o que levou a um estudo de uma parte da norma da linguagem de programação *Javascript*, o JSON. Analisada a alternativa mais comum, XML, verificou-se que os *browsers* possuem ferramentas integradas capazes de interpretar objectos deste tipo, o que levou à percepção de que a informação poderia ser enviada, directamente para o *browser*, pelo servidor aplicacional, fazendo a sua manipulação já no cliente, o que mostra uma maior interacção na aplicação, tornando-a mais dinâmica.

Capítulo 3

Não existindo uma camada intermédia de persistência, definida de forma clara no MVC, foi necessário perceber que soluções existem actualmente que sejam capazes de satisfazer a criação de uma camada abstracta. Para tal, procedeu-se a uma análise a ferramentas, já existentes no mercado, que possibilitassem a devida abstracção. Recorrendo a EJB, numa primeira fase, observou-se que estaria a ser criada entropia na aplicação, devido à complexidade associada, além de não resolver na plenitude o problema surgido. Com Hibernate verificou-se que também se estaria a criar uma intrusão não desejada no desenvolvimento de aplicações *web* com estas características,

pois é uma ferramenta mais propícia a aplicações que contenham a manipulação dos dados mais centradas no servidor aplicacional, através de *CRUD* (Create, Read, Update, Delete) esta ferramenta possibilita o manuseamento dos dados existentes num SGBD, sendo também possível executar *queries* SQL através da linguagem própria que a ferramenta possui. Observada a entropia que poderia criar, decidiu-se analisar em melhor detalhe as arquitecturas padrão para *Data Access Objects* (DAO) para proceder à estruturação de uma arquitectura orientada a *stored procedures*.

Capítulo 4

Para este trabalho foram analisadas várias *frameworks* Java para *web* e a forma como estas constroem interfaces gráficos com o utilizador, tendo sido efectuado um estudo sobre como essas ferramentas se encontram arquitectadas e como podem ser usadas em projectos de desenvolvimento de aplicações *web*.

Capítulo 5

Feito o estudo e pesquisa e enquadrado com necessidades industriais, surgiu a interrogação de como obter os dados, no servidor *web*, a partir de *stored procedures* executados numa base de dados Oracle.

Aqui é apresentada a arquitectura proposta e as metodologias que podem levar à sua implementação em qualquer projecto de desenvolvimento *web*. As análises efectuadas foram sempre de acordo com projectos empresariais e visando sempre a abstracção de metodologias usadas diariamente e que são, muitas vezes, propícias a erro.

Através de experiência directa, foram analisadas os métodos de codificação e as arquitecturas usadas no diário desenvolvimento de aplicações *web* que possuem muita informação e que contêm regras de negócio aplicadas directamente no SGBD. Assim, percebeu-se que se poderiam aplicar alguns dos padrões estudados para melhorar o desenvolvimento das tarefas diárias.

- **DAO e POJO**

Foi bastante perceptível que muitos projectos recorrem à codificação de um DAO para cada *stored procedure* e que os POJOs são um recurso muito habitual quando se pretende mapear a informação proveniente do cliente e também aquela que provém da BD. Assim, foi identificado que esta solução aumenta o número de classes existentes num projecto e quanto maior o projecto, maior o número de classes e mais difícil a manutenção do código, estas análises foram feitas *in loco* no dia-a-dia do mestrando.

- ***Caching***

Observou-se a existência de aplicações *web* suficientemente críticas que uma paragem do sistema traria consequências nefastas para a empresa que usasse essas aplicações. Como tal, percebeu-se a necessidade de recorrer a um sistema de cache e a uma arquitectura de persistência de dados que pudessem diminuir o número de *deploys* existentes em cada actualização.

- **Rápida disponibilidade em *web***

Percebeu-se a necessidade que o cliente tem em fazer novos requisitos e como esses podem ser mais rapidamente implementados e como, hoje em dia, o trabalho a realizar tem sempre em vista como *terminus* o dia anterior. Deste modo, compreendeu-se a necessidade de disponibilizar a informação o mais rápida e eficientemente possível, para tal ponderou-se o recurso a JSON com objecto de transferência entre a BD e o servidor aplicacional. A decisão tomada em determinada altura, regeu-se pelos princípios de que os *browsers* obtêm e tratam texto, isto visto de uma forma muito simplista, naturalmente.

- **Ferramentas**

Foi necessário procurar as ferramentas que possibilitassem as devidas abstracções e permitissem um melhor desempenho por parte de equipas de desenvolvimento *web*. Recorrendo a várias bibliotecas existentes e analisando alguns padrões arquitecturais e de desenho, começou a estruturar-se a metodologia aqui apresentada. Com os padrões de DAO identificados e bibliotecas capazes de interpretar XML e DTD, uma biblioteca JSON para Java, procedeu-se a uma pequena

implementação da forma de invocar os *stored procedures*. Numa primeira fase observou-se que a forma de comunicar com a BD poderia, perfeitamente, ser abstraída através de JDBC e que com o *driver* Oracle é possível obter os tipos possíveis de resultado de um *stored procedure*, assim como os seus argumentos necessários à invocação (quando existentes), depois percebeu-se a necessidade de existir um objecto que pudesse ser o centro da arquitectura para registar parâmetros, colocar valores, invocar *stored procedures* e outras funcionalidades.

- **Implementação e uso**

No fim, foi implementada uma solução desta arquitectura que foi evoluindo e, com a opinião e crítica de outros programadores, foi sendo aprimorada até chegada a altura de colocar em ambiente de produção.

6.2 Conclusões

A arquitectura/metodologia de persistência de dados com recurso a JSON/POJO apresentada neste trabalho é uma solução para empresas que recorram à aplicação de regras de negócio na BD e devolvam a(os) informação(dados) através da invocação, por parte de um servidor aplicacional *web*, de *stored procedures* e não vejam a necessidade de recorrer a arquitecturas mais complexas que possuem mais do que é necessário.

Esta arquitectura foi concebida tendo em vista resolver a dificuldade que existe, nalguns casos, em estruturar e manter o código desenvolvido em aplicações *web* de grande porte, através de um aumento linear do número de objectos POJO existentes na sua estrutura.

O recurso a JSON como objecto de transferência (*Transfer Object*) entre a BD e o servidor aplicacional revelou ser uma abordagem bastante interessante de idealizar e implementar, tendo reduzido drasticamente o número de linhas de código usadas no projecto.

A definição da arquitectura de persistência com os diversos mecanismos mostrou que os programadores cometem menos erros no mapeamento dos dados obtidos da BD, tendo diminuído o tempo necessário para implementar os requisitos, pois a preocupação em codificar a execução dos

procedimentos a partir do servidor aplicacional deixou de existir e encontrando-se devidamente abstraída possibilita que os seus recursos sejam direccionados para a resolução de outros problemas. Através de experiência directa ficou perceptível que na fase de implementação sempre que existia a necessidade de alterar um POJO (através, por exemplo, da adição de atributos) havia também a necessidade de reinstalar a aplicação *web*, pois as classes haviam sido compiladas e o servidor aplicacional necessitava de recarregar esses objecto no *classloader*, pois não efectua isso de forma dinâmica. Assim percebeu-se que recorrendo à devida parametrização no URL seria possível colocar directamente esses parâmetros (e os respectivos valores) na arquitectura apresentada, possibilitando também a abstracção do envio dos dados para o *Factory*. Naturalmente, este processo também poderia ser efectuado pelo Business Object, mas sempre que fosse necessário alterá-lo, teria de ser feito um novo *deploy* da aplicação. Para evitar o tempo desperdiçado e também para tornar mais eficiente a aplicação, recorreu-se ao sistema de *cache* que melhora a disponibilidade da aplicação.

De um modo geral, é apresentada uma arquitectura escalável, robusta e eficiente que melhora os processos de desenvolvimento *web*, diminuindo o custo de manutenção com a aplicação, reduzindo o tempo necessário aos programadores *web* para implementar determinados requisitos, potenciando os seus recursos para outras tarefas.

6.3 Trabalho Futuro

Presentemente esta arquitectura encontra-se já implementada em várias aplicações *web* que possuem um grande número de regras de negócio implementadas na BD através de *stored procedures*. Tendo, durante os trabalhos com vista à conclusão, o mestrando desenvolvido e implementado a arquitectura aqui descrita, é prática comum ir identificando potenciais novas funcionalidades e corrigindo eventuais erros na codificação da arquitectura. Deste modo, foram identificados novos aspectos a incluir/abstrair na arquitectura, nomeadamente:

- acrescentar um nível entre o *Business Object* e a arquitectura de forma a poder-se abstrair, também, o tipo de *Business Object*;
- a possibilidade de abstrair mais um nível na arquitectura, tornando-a completamente independente do SGBD usado;
- identificar estruturas passíveis de abstrair tipos de dados característicos do SGBD (apesar de JSON conseguir essa abstracção, na plenitude, para texto, valores numéricos e valores lógicos);
- desenvolver uma solução OGNL que permita navegar sobre os objectos JSON da mesma forma que é feita a navegação sobre qualquer POJO;
- após a solução OGNL, desenvolver uma API *(framework)* de interfaces gráficos que pudesse ser acoplada a esta arquitectura, mas de forma independente;
- potenciar uma solução integrada que, respeitando a arquitectura MVC, permitisse agilizar o desenvolvimento de aplicações *web* com recurso a esta arquitectura e recorrendo, eventualmente, a *frameworks Java* para *web* que permitam a criação dos interfaces gráficos e a abstracção dos pedidos e respostas entre um cliente e um servidor aplicacional.
- desenvolver soluções *CRUD* sobre um SGBD que permitam, através da mesma metodologia, agilizar o mapeamento objectos/relacional.

Referências

[Aarsten et al, 1996] Aarsten, A. and Brugali, D. and Menga, G. (1996) "*Patterns for three-tier client/server applications"*, Proc. of Plop, vol. 96, Citeseer.

[Alexander et al, 1977] Alexander, C. and Ishikawa, S. and Silverstein, M. (1977). **"***A pattern language: towns, buildings, construction."*, Oxford University Press, USA.

[Alur et al, 2001] Alur, D. and Malks, D. and Crupi, J. (2001). *"Core J2EE patterns: best practices and design strategies",* Prentice Hall PTR Upper Saddle River, NJ, USA.

[Annotations, 2004] *Annotations,* Sun Microsystems, 2004*. Site* contendo informação sobre como desenvolver e implementar soluções com recurso a anotações em java. Acedido a 04/02/2009.
[Online] http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

[ASP_JSON, 2008] *JSON for ASP*, Página com informação relacionada com a implementação de JSON para ASP. Acedido a 18/03/2009.
[Online] http://code.google.com/p/aspjson/

[Avgeriou & Zdun, 2005] Avgeriou, P. and Zdun, U. (2005). "*Architectural patterns revisited - a pattern language"*, 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), *pags. 1-39*.

[Bauer & King, 2005] Bauer, C. and King, G. (2005). *"Hibernate in action",* Manning.

[Berry et al, 2002] Berry, C.A. and Carnell, J. and Juric, M.B. and Kunnumourath, M.M. and Nashi, N. and Romanosky, S. (2002). *"J2EE design patterns applied",* Wrox Press.

[Boehm, 1988] Boehm, B. (1988). "*A Spiral Model of Software Development and Enhancement.", Computer*, Maio 1988, pp.61-72

[Brown, Davis & Stanlick, 2008] Brown, D., Davis, C.M., Stanlick, S., (2008). "*Struts 2 in Action*", Manning.

[Burbeck, 1997] Burbeck, S., 1997. "*Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)*", [Online] (Actualizado em 4 Março 1997)
Disponível em http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

[Burleson, 2009] *USER_ARGUMENTS view tips,* Página mantida por Don Burleson, autor Oracle, onde estão listadas algumas dicas para o SGBD Oracle. Acedido a 28/03/2009.
[Online] http://www.praetoriate.com/data_dictionary/dd_user_arguments.htm

[CMP, 2002] *Container-Managed Persistence*, Página contendo os conceitos relacionados com *Container Managed Persistence*, assim como a contextualização através de exemplos. Acedido a 21/06/2009.
[Online] http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts4.html#62950

[Conallen, 1999] Conallen, J.,1999. *"Modeling Web application architectures with UML". Communications of the ACM*, Outubro 1999, 42(10), pp.63-70.

[CoreJ2EE, 2002] *Blueprints, Welcome to Core J2EE Patterns!*, página contendo descrições sobre as arquitecturas padrão encontradas no Java Enterprise Edition. Acedido a 11/03/2009. [Online] http://java.sun.com/blueprints/corej2eepatterns/index.html

[Crockford, 2006] Crockford, D. *"RFC 4627: "The application/json media type for javascript object notation (json)"* 2006. The Internet Society.

[Crockford, 2009] Crockford, D. *"JSONRequest"* 2009.
Disponível em http://www.json.org/JSONRequest.html

[DAO, 2002] *Core J2EE Patterns - Data Access Object,* Página da *Sun* contendo a descrição do padrão DAO, assim como a estrutura para implementação do padrão. Acedido a 14/03/2009. [Online] http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

[DesigningJ2EE, 2005] *Designing Enterprise Applications with the J2EE Platform, Second Edition*, página referência para o livro com o mesmo título. Acedido a 17/05/2009. [Online] http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/

[DSI, 2007] http://sim.di.uminho.pt/ensino2.php3?seccao=apoio&id=67

"Slides aulas teóricas da disciplina Desenvolvimento de Sistemas de Informação", Vários documentos contendo informação teórica sobre o desenvolvimento de sistemas de software. Disponibilizado pelos Professores Campos, J. Creissac e Ribeiro, A. Nestor.

[DTD, 2008] *Guide to the W3C XML Specification ("XMLspec") DTD, Version 2.1*, Página referência para a especificação de DTD. [Online] http://www.w3.org/XML/1998/06/xmlspec-report.htm Acedido a 12/03/2009

[ECMA, 1999] Ecma International. Standard ecma-262. http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf, 1999. ECMAScript Language Specification.

[Englander, 1997] Englander, R., 1997. *"Developing Java Beans"*, 1st ed. O'Reilly, 1997.

[Fernandez, 2001] Fernandez, G. (2001). *"WebLearn: A Common Gateway Interface (CGI)-Based Environment for Interactive Learning"*. Journal of Interactive Learning Research. 12 (2), pp. 265-280. Norfolk.

[Fielding et al, 1999] Fielding, R. and Gettys, J. and Mogul, JC and Frystyk, H. and Masinter, L. and Leach, P. and Berners-Lee, T. (1999). "*Hypertext Transfer Protocol-HTTP/1.1."*, The Internet Society.

[Fonseca & Simões, 2007] Fonseca, R. and Simões, A.. (2007). *"Alternativas ao XML: YAML e JSON"*, XATA2007: XML: aplicacoes e tecnologias associadas: actas da Conferencia Nacional, *vol 5, pags 33-46*.

[Forman & Forman, 2005] Forman, I.R. and Forman, N. (2005). "*Java reflection in action"*, Manning.

[Ford, 2004] Ford, N., "*Art of Java Web Development"*, 1st ed. Manning, 2004.

[Fowler, 1999] Fowler, M. (1999). "*Refactoring, Improving the design of existing code.",* Addison-Wesley Professional.
Disponível em http://www.martinfowler.com/bliki/POJO.html

[Fraternali, 1999] Fraternali, P. R. (1999). "*Tools and approaches for developing data-intensive Web applications: a survey.",* ACM Computing Surveys, *vol. 31*, *number 3*.

[Gamma et al, 1994] Gamma, E. and Helm, R. and Johnson, R. and Vlissides, J. (Outubro, 1994). "*Design Patterns: elements of reusable object-oriented software."*, Addision-Wesley.

[GoF, 2006] *Gang of Four*, Página relacionada com os autores do livro *"Design Patterns: elements of reusable object-oriented software."* e a origem da alcunha *GangOfFour*. Acedido a 07/07/2009. [Online] http://c2.com/cgi/wiki?GangOfFour

[GoFPatterns, 2001] *Gang of Four Design Patterns*, página contendo os padrões propostos pelo GoF. Acedido a 18/05/2009. [Online] http://www.tml.tkk.fi/~pnr/GoF-models/html/

[Korthaus & Merz, 2003] Korthaus, A. and Merz, M. (2003). *"A Critical Analysis of JDO in the Context of J2EE",* Proceedings of the 2003 International Conference on Software Engineering Research and Practice (SERP'03), *vol 1, pags 34-40,* Citeseer.

[Hibernate, 2004] *HIBERNATE – Relational Persistence for Idiomatic Java*, Página com informação sobre como executar *stored procedures* em Hibernate. Acedido a 05/08/2009. [Online] http://docs.jboss.org/hibernate/stable/core/reference/en/html_single/#sp_query

[Hirschfeld, 1996] Hirschfeld, R. (1996). "*Three-Tier Distribution Architecture.",* Collected papers from the PloP, *vol. 96*, *pags. 97-07,* Citeseer.

[HTTP, 1999] *Hypertext Transfer Protocol – HTTP/1.1*, página contendo a especificação do protocolo HTTP, na versão actual – 1.1. Acedido a 10/03/2009.
[Online] http://tools.ietf.org/html/rfc2616

[IBM, 2005] *CallableStatements and INOUT Parameters,* página proprietária da IBM contendo referências sobre como registar os parâmetros num CallableStatement. Acedido a 28/03/2009.
[Online] http://publib.boulder.ibm.com/infocenter/cscv/v10r1/index.jsptopic=/com.ibm.cloudscape.doc/rre fjdbc75719.html

[ICEFaces, 2009] *ICEFaces.org*, Página principal do projecto ICEFaces. É possível aceder ao fórum dos utilizadores, assim como obter acesso a alguns tutoriais. Para obter determinados recursos é necessário registo. Acedido a 24/02/2009.
[Online] http://www.icefaces.org/

[J2EE, 2007] Sun Microsystems, *"The Java EE Tutorial",* Setembro 2007.

[Jamae & Johnson, 2008] Jamae, J. and Johnson, P., "*JBoss in Action"*, Manning Publications, Maio 2008.

[JavaWebFrameworks, 2009] *Comparison of web application frameworks*, página *wiki* com uma lista de *frameworks* java existentes para *web*. Acedido a 01/02/2009.
[Online] http://en.wikipedia.org/wiki/List_of_web_application_frameworks#Java

[JDBC, 1999] *Getting Started with the JDBC API,* Página contendo informação e um tutorial sobre como usar o JDBC. Acedido a 11/03/2009.
[Online] http://java.sun.com/j2se/1.3/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html

[JSON, 2002] *Introducing JSON*, página principal do projecto JSON, com informação relativa ao seu formato e formas para interpretar e analisar objectos JSON. Acedido a 02/05/2009.
[Online] http://www.json.org/

[JSON_AS1, 2008] *JSON library for ActionScript 1.0*, Código fonte para implementações de JSON em ActionScript1. Acedido a 03/05/2009.
[Online] http://joose-js.googlecode.com/svn/trunk/ext/actionscript1/json.html

[JSON_AS3, 2009] *ActionScript 3.0 library for several basic utilities.* Biblioteca contendo implementação e outras funcionalidades de JSON para ActionScript3. Acedido a 03/05/2009.
[Online] http://code.google.com/p/as3corelib/

[JSONJava, 2002] *JSON in Java*, página contendo a documentação da implementação da estrutura JSON em java, sendo possível obter o código-fonte da mesma estrutura. Acedido a 02/05/2009.
[Online] http://www.json.org/java/index.html

[JSON_Parser, 2009] *JSON Parser*, um parser de JSON para a linguagem de programação C. Acedido a 02/05/2009.
[Online] http://fara.cs.uni-potsdam.de/~jsg/json_parser/

[JSR220, 2009] *JSR-000220 Enterprise JavaBeans 3.0,* página contendo a especificação da arquitectura de persistência de dados JPA. Acedido a 14/03/2009.
[Online] http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html

[Lightbody & Carreira, 2006] Lightbody, Patrick and Carreira, Jason, *"Webwork In Action"*, 3rd ed. Manning, 2006.

[Martins, 2000] Martins, F. Mário, "*Programação Orientada aos Objectos em JAVA2"*, FCA, 2000.

[MyFaces, 2008] *Apache MyFaces*, página principal do projecto *MyFaces.* Possui informação actualizada sobre o projecto, assim como documentação que permite compreender a tecnologia. Acedido a 24/02/2009.
[Online] http://myfaces.apache.org/

[Ntier, 2009] *N-Tier Application Development with Microsoft.NET*, página com informação sobre o desenvolvimento de aplicações *web* em n-camadas através da *framework .NET*. Acedido a 11/07/2009.
[Online] http://www.microsoft.com/belux/msdn/nl/community/columns/hyatt/ntier1.mspx

[OGNL, 2006] *Object Graph Navigation Language*, página onde se encontra informação sobre o OGNL. Acedido a 12/02/2009.
[Online] http://www.ognl.org/2.6.9/Documentation/html/LanguageGuide/index.html

[Ozyurt, 2003] Ozyurt, I.B. (2003). "*J2EE based Web-Interface Development Framework for Clinical Imaging Databases."*, [Online]
Disponível em http://www.nbirn.net/tools/human_imaging_gui/hid_app_manual.pdf

[PatternWiki, 2009] *Design pattern (computer science*), página wiki com contexto histórico e evolução dos *design patterns* ao longo do tempo, contendo também uma análise crítica aos mesmos. Acedido a 06/07/2009.
[Online] http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29

[Petersen, 2007] Petersen, J. (2007). "*Benefits of using the n-tiered approach for web applications"*, ColdFusion Developers Journal.
Disponível em http://www.adobe.com/devnet/coldfusion/articles/ntier.html

[PHP, 2006] *PHP 5.2.0 Release Announcement*, página com informação relacionada com a implementação de JSON para PHP, contendo informações relacionadas com a versão 5.2.0. Acedido a 04/05/2009.
[Online] http://www.php.net/releases/5_2_0.php

[Pierce & al, 2006] Pierce, M.E. and Fox, G. and Yuan, H. and Deng, Y. (2006). *"Cyberinfrastructure and Web 2.0",* Proceedings of HPC2006, July, *vol 4, pags 2006,* Citeseer.

[PL/JSON, 2009] *PL/JSON*, PL/JSON é um objecto JSON genérico escrito em PL/SQL. Biblioteca contendo a implementação do objecto JSON em PL/SQL. Acedido a 02/05/2009.
[Online] http://sourceforge.net/projects/pljson/

[Push&Pull, 2003] *Clarification on MVC Pull and MVC Push,* página contendo informação e clarificando a definição de *Push* e *Pull* no âmbito do Model-View-Controller. Acedido a 21/02/2009.
[Online] http://www.theserverside.com/patterns/thread.tss?thread_id=22143

[ResultSet, 2003] *ResultSet*, página contendo a documentação do interface *ResultSet* assim como exemplos de utilização. Acedido a 14/03/2009.
[Online] http://java.sun.com/j2se/1.4.2/docs/api/java/sql/ResultSet.html

[Richardson, 2006] Richardson, C. (2006). "*Untangling enterprise java.",* AddisoACM New York, NY, USA.

[RichFaces, 2008] *Rich AJAX enabled components for your JSF applications*, página principal do projecto RichFaces, sendo uma implementação de JSF por parte da JBoss. Acedido a 22/02/2009.
[Online] http://www.jboss.org/jbossrichfaces/

[RJson, 2009] *RJson: A reflective JSON serializer/parser*, página com informação relacionada com a implementação de JSON para Haskell. Acedido a 18/03/2009.
[Online] http://hackage.haskell.org/cgi-bin/hackage-scripts/package/RJson

[SAX, 2004] *About SAX*, página oficial do projecto SAX (*Simple API for XML*). Acedido a 14/03/2009.
[Online] http://www.saxproject.org/

[Shan, 2006] Tony C Shan; Winnie W Hua. (2006) *"Taxonomy of Java Web Application Frameworks,"* *e-Business Engineering, 2006. ICEBE '06. IEEE International Conference on* , vol., no., pp.378-385, Oct. 2006. Disponível em
http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4031677&isnumber=4031614

[Singh et al, 2002] Singh, I. and Johnson, M. and Stearns, B. (2002), *"Designing enterprise applications with the J2EE platform"*, Addison-Wesley Professional.

[Webwork, 2005] *Webwork 2.1.7*, página com informações relacionadas com a *framework Webwork*. Acedido a 19/02/2009.
[Online] http://www.opensymphony.com/webwork/wikidocs/WebWork%202.1.7.html

[ZeroConfiguration, 2008] *Zero Configuration*, página com informação relativa ao conceito de "Zero Configuration" para permitir a escrita de meta-dados directamente nas classes implementadas. Acedido a 07/02/2009.
[Online] http://struts.apache.org/2.0.11.2/docs/zero-configuration.html

[Zoio, 2005] Zoio, P. (2005). *"JavaServer Faces vs Tapestry"*. A Head-to-Head Comparison. [Online] TheServerSide.com.

Anexos

A. Package org.json

Interface Summary	
JSONString	The `JSONString` interface allows a `toJSONString()` method so that a class can change the behavior of `JSONObject.toString()`, `JSONArray.toString()`, and `JSONWriter.value(`Object`)`.

Class Summary	
CDL	This provides static methods to convert comma delimited text into a JSONArray, and to covert a JSONArray into comma delimited text.
Cookie	Convert a web browser cookie specification to a JSONObject and back.
CookieList	Convert a web browser cookie list string to a JSONObject and back.
HTTP	Convert an HTTP header to a JSONObject and back.
HTTPTokener	The HTTPTokener extends the JSONTokener to provide additional methods for the parsing of HTTP headers.
JSONArray	A JSONArray is an ordered sequence of values.
JSONML	This provides static methods to convert an XML text into a JSONArray or JSONObject, and to covert a JSONArray or JSONObject into an XML text using the JsonML transform.
JSONObject	A JSONObject is an unordered collection of name/value pairs.
JSONStringer	JSONStringer provides a quick and convenient way of producing JSON text.
JSONTokener	A JSONTokener takes a source string and extracts characters and tokens from it.
JSONWriter	JSONWriter provides a quick and convenient way of producing JSON text.
Test	Test class.
XML	This provides static methods to convert an XML text into a JSONObject, and to covert a JSONObject into an XML text.
XMLTokener	The XMLTokener extends the JSONTokener to provide additional methods for the parsing of XML texts.

Exception Summary	
JSONException	The JSONException is thrown by the JSON.org classes when things are amiss.

B. *Action* inseredados()

```
package pck.exemplo;
 public class PessoaAction extends ActionSupport implements
ParameterAware, SessionAware {

      private Pessoa p;

      public PessoaAction(){}

      public void inseredados(){
            try{
                  PessoaDAO pDao = new PessoaDAO();
                  boolean executed = pDao.insereDados(p);
                  pDao = null;
                  if(executed){
                  System.out.prinln("Executou SP na BD");
                  }
                  else{
                  System.out.prinln("Não executou SP na BD");
                  }
            }
             catch (Exception e) {
            System.out.prinln("ERRO: "+e.getMessage());
      }

    public String getMorada() { return p.getMorada();}
    public void setMorada(String morada) { this.p.setMorada(morada);}
    public String getNacionalidade() { return p.getNacionalidade(); }
    public void setNacionalidade(String nacionalidade) {
        this.p.setNacionalidade(nacionalidade);
    }
    public String getNome() { return p.getNome(); }
    public void setNome(String nome) { this.p.setNome(nome); }
    public Pessoa getP() { return p; }
    public void setP(Pessoa p) { this.p = p; }
    public Integer getTelefone() { return p.getTelefone(); }
    public void setTelefone(Integer telefone){
        this.p.setTelefone(telefone);}
    public Integer getTelemovel() {return p.getTelemovel();}
    public void setTelemovel(Integer telemovel){
        this.p.setTelemovel(telemovel);
    }
}
```

C. *Bean* Pessoa

```
public class Pessoa{
      private String nome;
      private String dataNasc;
      private String nacionalidade;
      private String morada;
      private Integer telefone;
      private Integer telemovel;

      public Pessoa(){
      }

    public String getMorada() {return morada;}

    public void setMorada(String morada) {this.morada = morada;}

    public String getNacionalidade() {return nacionalidade;}

    public void setNacionalidade(String nacionalidade) {
        this.nacionalidade = nacionalidade;
    }

    public String getNome() {return nome;}

    public void setNome(String nome) {this.nome = nome;}

    public Integer getTelefone() {return telefone;}

    public void setTelefone(Integer telefone) {this.telefone=telefone;  }

    public Integer getTelemovel() {return telemovel;}

    public void setTelemovel(Integer telemovel) {
        this.telemovel = telemovel;
    }
}
```

D. PessoaDAO

```
package.DAO

import java.sql.SQLException;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.CallableStatement;
import oracle.jdbc.driver.OracleTypes;

public class PessoaDAO{

private Pessoa pessoa;
Connection conn = null;
Statement stmt = null;
CallableStatement pstmt = null;
DBConnector dbConn = new DBConnector();

      public PessoaDAO(){
      }
      public PessoaDAO(Pessoa p){
            this.pessoa = p;
      }

    public boolean insereDados(Pessoa p) {
      conn = dbConn.obtemConexao();
      pstmt = conn.prepareCall("{call pck_global.pessoaSP(?,?,?,?,?)}");
      pstmt.setString(0,p.getNome());
      pstmt.setString(1,p.getMorada());
      pstmt.setInt(2,p.getTelefone());
      pstmt.setInt(3,p.getTelemovel());
      pstmt.setString(4,p.getNacionalidade());
      return pstmt.execute();
    }
}
```

D. ListaPessoaDAO

```
package.DAO

import java.sql.SQLException;
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.CallableStatement;
import oracle.jdbc.driver.OracleTypes;

public class ListaPessoaDAO{

private Pessoa pessoa;
private ArrayList listaPessoas;
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
CallableStatement pstmt = null;
DBConnector dbConn = new DBConnector();

public ListaPessoaDAO(){}

public ArrayList<Pessoa> getPessoas() {
      conn = dbConn.obtemConexao();
      pstmt = conn.prepareCall("{call pck_global.lstPessoas_SP()}");
      if(pstmt.execute()){
            rs = (ResultSet) pstmt.getObject(1);
            while(rs.next()){
              pessoa = new Pessoa();
              pessoa.setNome(rs.get(0));
              pessoa.setIdade(rs.get(1));
              pessoa.setNacionalidade(rs.get(2));
              listaPessoas.add(pessoa);
            }
      }
      return listaPessoas;
    }
}
```

E. Classe DBConect

```
package exemplo.utilitario;

import java.sql.*;

public class DBConect {

    public DBConect() {
    }

    public Connection dbConnect(String db_connect_string, String
db_userid, String db_password) {
        try {
     DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
     Connection conn = DriverManager.getConnection(db_connect_string,
db_userid, db_password);
            return conn;

        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
    public void close(Connection connection) {
        if(connection != null) {
            try {
                connection.close();
                connection = null;

            } catch (SQLException sqlexcept) {
                sqlexcept.printStackTrace();

            }
        }
    }
}
```