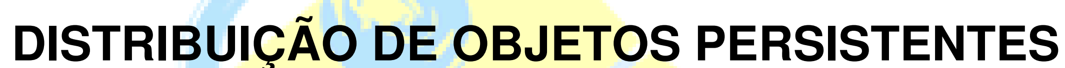
CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DO PIAUÍ
POS-GRADUAÇÃO EM BANCO DE DADOS



DISTRIBUIÇÃO DE OBJETOS PERSISTENTES

MONOGRAFIA DE PÓS-GRADUAÇÃO

Leandro Sales Lima

Teresina – PI
2005

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

Leandro Sales Lima

DISTRIBUIÇÃO DE OBJETOS PERSISTENTES: usando RMI para distribuir de forma transparente objetos persistentes *Hibernate*

Trabalho de conclusão do Curso de Especialização apresentado ao Centro Federal de Educação Tecnológica do Piauí como requisito parcial à obtenção do título de Especialista em Banco de Dados.

Orientador: Erick Passos

Teresina – PI

2005

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

DEDICATÓRIA

Dedico este momento glorioso da minha vida aos meus pais, minha esposa e a todos que direto ou indiretamente, contribuíram para o sucesso deste projeto. E, o mais importante, além da sensação de dever cumprido, é saber que não estive só.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

RESUMO

Este trabalho tem o objetivo de mostrar as principais funcionalidades do Mapeamento Objeto-Relacional (MOR) usando a tecnologia chamada *Hibernate*, bem como mostrar o seu uso com RMI (*Remote Method Invocation*), para uma distribuição de forma transparente dos objetos persistentes, mostrando as vantagens, desvantagens e pontos a serem observados na implementação.

Palavras-chave: RMI, Orientado a Objetos Distribuídos, *Hibernate*, Mapeamento Objeto-Relacional, Banco de Dados.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

LISTA DE SIGLAS/ABREVIATURAS

BDOO	Banco de Dados Orientado a Objeto
CORBA	Common Object Request Broker Architecture
DTD	Document Type Definitions
EJB	Enterprise JavaBeans
J2EE	Java 2 Enterprise Edition
JDBC	Java DataBase Connectivity
HQL	Hibernate Query Language
MOR	Mapeamento Objeto Relacional
OO	Orientado a Objeto
OQL	Object Query Language
RMI	Remote Method Invocation
SGBD	Sistema Gerenciador de Bando de Dados
SGBDOO	Sistema Gerenciador de Bando de Dados Orientado a Objetos
SGBDR	Sistema Gerenciador de Bando de Dados Relacional
SQL	Structured Query Language
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	Extensible Markup Language

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

LISTA DE FIGURAS

Figura 1 – Funcionalidade de uma aplicação com distribuição de objetos persistentes ..... p 09

Figura 2 – Esquema conceitual de “Formula 1” ..... p 20

Figura 3 – Modelo entidade-relacionamento do esquema conceitual de “Formula 1” ..... p 21

Figura 4 – Exemplo do código de parte da classe Pais ..... p 22

Figura 5 – Exemplo do arquivo de configuração do *Hibernate* (hibernate.cfg.xml) ..... p 23

Figura 6 – Possíveis valores para a propriedade dialect ..... p 24

Figura 7 – Exemplo de mapeamento da classe Pais (Pais.hbm.xml) ..... p 25

Figura 8 – Exemplo de mapeamento da classe Equipe (Equipe.hbm.xml) ..... p 27

Figura 9 – Exemplo do arquivo Equipe.java que possui a implementação da classe Equipe ..... p 28

Figura 10 – Exemplo do arquivo ImplBO.java que possui a implementação da classe ImplBO ..... p 29

Figura 11 – Exemplo do arquivo HibernateUtil.java que possui a classe que gerencia as sessões ..... p 30

Figura 12 – Exemplo da interface EquipeInterface que é implementada pela classe Equipe ..... p 32

Figura13 – Exemplo da interface ImplBOInterface que é implementada pela classe ImplBO ..... p 32

Figura 14 – Exemplo do arquivo Formula.java responsável pela instanciação dos objetos distribuídos ..... p 33

Figura 15 – Exemplo da classe Server (Server.java) que registra o objeto Formula ..... p 33

Figura 16 – Classe ObjetoRemoto responsável pelo gerenciamento da conexão do cliente ..... p 34

Figura 17 – Arquivo Client.java mostra um exemplo em modo texto de “cliente” p 35

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

SUMÁRIO

RESUMO.. p 04
LISTA DE SIGLAS / ABREVIATURAS .... p 05
LISTA DE FIGURAS.. p 06
SUMÁRIO .... p 07
1 INTRODUÇÃO .... p 08
2 OBJETIVOS.. p 10
2.1 Objetivo Geral .... p 10
2.2 Objetivo Específico .... p 10
3 CONCEITOS DE MAPEAMENTO OBJETO RELACIONAL .... p 11
3.1 Visão Geral Sobre Orientação a Objeto.. p 11
3.2 Persistência Objeto-Relacional... p 12
3.2.1 Camadas de Persistência.. p 13
3.2.2 Vantagens da utilização .... p 14
3.2.3 Requisitos de uma Camada de Persistência.. p 14
4 SISTEMAS DISTRIBUIDOS .... p 17
4.1 Visão Geral Sobre Sistemas Distribuídos .... p 17
4.2 Objetos Distribuídos .... p 17
4.2.1 Tecnologias Orientadas a Objetos Distribuídos .... p 18
5 DISTRIBUIÇÃO DE OBJETOS PERSISTENTES .... p 20
5.1 Esquema Conceitual .... p 20
5.2 Mapeando Objeto para Tabelas .... p 21
5.2.1 Configuração do *Hibernate*... p 22
5.2.2 Criação do Mapeamento .... p 24
5.2.3 Criação das Classes de Persistência .... p 28
5.3 Distribuindo os Objetos Persistentes.. p 31
5.3.1 Criação da camada de distribuição dos objetos persistentes..................... p 32
5.4 Implementação do Cliente .... p 35
6 CONCLUSÃO .... p 37
REFERÊNCIAS.. p 39

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

1 INTRODUÇÃO

O estudo de Mapeamento Objeto-Relacional (MOR) e sistemas distribuídos é um assunto bastante abordado devido à necessidade de trabalhar com sistemas totalmente orientados a objetos adotando o "*design patterns* OO" e aplicações distribuídas.

Com o amadurecimento e a grande aceitação do Java, a programação orientada a objeto assumiu o primeiro plano no cenário de desenvolvimento de aplicações devido aos seus ricos modelos de dados e ao suporte a conceitos que melhoram a produtividade, tais como encapsulamento, herança e polimorfismo. Tecnologias de objetos como Java, C++, C# e COM [MF2003] são preferidas pelos atuais desenvolvedores de aplicações.

Entretanto, uma grande parte dos dados existentes no mundo ainda reside em bancos de dados relacionais. Os desenvolvedores de aplicações de bancos de dados (ou seja, qualquer aplicação que acesse dados armazenados) freqüentemente se vêem brigando com problemas de diferenças de impedância: a inerente falta de casamento entre os modelos de dados relacionais e os orientados a objeto. Mas porque não usar Banco de Dados Orientado a Objeto (BDOO)? BDOO ainda não são tão maduros quanto os relacionais em respeito à recuperação, desempenho e distribuição [FL2004].

As bases de dados relacionais são estruturadas em uma configuração tabular e os exemplos orientados a objetos são estruturados tipicamente em uma maneira hierárquica. Essa combinação inapropriada conduziu ao desenvolvimento de diversas tecnologias diferentes para persistência de objetos que tentam construir uma ponte entre o mundo relacional e o mundo orientado a objeto.

Um outro problema que os desenvolvedores encontram é a dificuldade de manutenção e atualização de novas versões e o alto custo das maquinas. Quando fazemos com que a aplicação trabalhe de forma distribuída, a atualização de versão, por exemplo, é muito mais simples. Em muitos casos você só precisará alterar o banco de dados, ou apenas o código que está rodando no servidor. Outro detalhe importante no uso de sistemas distribuídos é a diminuição dos custos, tendo em vista que as máquinas denominadas clientes não necessitarão ser máquinas de

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

grande porte, pois toda a lógica de negócio estará rodando apenas na máquina denominada servidora.

Outra idéia de ter a aplicação trabalhando de forma distribuída é fazer a segurança do banco de dados, fazendo com que os clientes não acessem diretamente o banco de dados. Isso seria possível implementando um *firewall* com regras que fizesse com que a única máquina com permissão para acessar o banco de dados seria a maquina servidor.

Para resolver os problemas citados, surgiram diversas tecnologias, dentre as quais, o *framework Hibernate*, que é um recurso bastante avançado de mapeamento objeto-relacional (de classes Java para tabelas de banco de dados). Com ele é possível construir facilmente uma camada de persistência transparente aos objetos.

Já o RMI é uma tecnologia bem simples de trabalhar para quem quer distribuir de forma transparente os objetos e invocar seus métodos remotamente. É possível transportar objetos pela rede e também chamar métodos que estejam em outro computador (servidor), mantendo o objeto na máquina remota.

Na figura 1 pode-se ter uma idéia de como uma aplicação funcionará usando essas tecnologias:

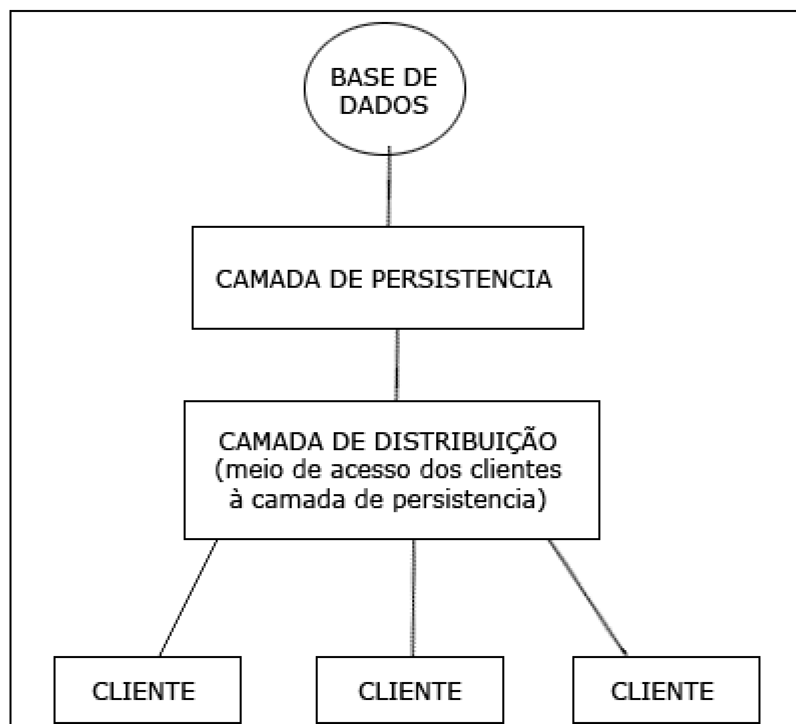


Figura 1 – Funcionalidade de uma aplicação com distribuição de objetos persistentes

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

2 OBJETIVOS

2.1 OBJETIVO GERAL

Os objetivos principais deste trabalho é o estudo das vantagens e desvantagens do mapeamento objeto-relacional com a distribuição dos objetos e mostrar as principais dificuldades na implementação de um sistema usando essas tecnologias.

2.2 OBJETIVOS ESPECÍFICOS

No presente trabalho procura-se contribuir com uma pesquisa abrangente sobre a tecnologia de mapeamento objeto-relacional e a distribuição de objetos persistentes.

Inicialmente, são apresentados os conceitos de OO (Orientação a Objeto), posteriormente estes conceitos são aplicados na criação do mapeamento objeto-relacional usando o *framework Hibernate*, e por último, a distribuição dos objetos persistentes criados anteriormente com RMI.

Este trabalho também tem o objetivo de mostrar a facilidade de trabalhar com essas tecnologias de forma simples e transparente, criando uma aplicação funcional sem ter que usar uma especificação *J2EE* (*Java 2 Enterprise Edition*) que é muito mais complexa.

Após o estudo teórico, é realizada uma avaliação detalhada e cuidadosa sobre a utilização desta tecnologia na pratica. É apresentada uma visão geral das tecnologias a serem analisadas, e posteriormente, avalia-se cada um de acordo com as dificuldades de implementação, melhoras na facilidade de manutenção e o desempenho da aplicação.

Com isso, este trabalho contém as informações necessárias para nortear quem procura implantar a tecnologia de mapeamento objeto-relacional com objetos distribuídos, iniciando um projeto ou migrando de um modelo relacional já existente. Este trabalho apresenta a base teórica fundamental para a compreensão destas tecnologias.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

3 CONCEITOS DE MAPEAMENTO OBJETO RELACIONAL

3.1 VISÃO GERAL SOBRE ORIENTAÇÃO A OBJETO

Segundo Larman [CL2000], a orientação a objeto é um paradigma para o desenvolvimento de aplicações. Ela aproxima a linguagem de computadores ao mundo real, visando realizar a modelagem e codificação de uma forma mais natural. Com ela, o software fica organizado como uma coleção de objetos que interagem entre si.

A técnica de OO propõe uma maior reusabilidade, através da eliminação de redundância de código, com conseqüente aumento na produtividade; melhorias na manutenabilidade, pois as modificações no código são bem localizadas e não provoca descontrole em outras partes do software; e também maior confiabilidade, devido ao encapsulamento que impede o acesso direto aos dados (atributos) dos objetos [CL2000].

Para Larman [CL2000], o objeto é o conceito central da abordagem OO. Ele pode ser entendido como algo do mundo real com limites bem definidos e identidade. Um objeto possui atributos (dados) e operações. Quando se define um objeto conceitual a partir de um objeto do mundo real, diz-se estar fazendo uma abstração.

De acordo com Larman [CL2000], classe é um conjunto de objetos que possuem características e comportamento em comum. Por exemplo, para os objetos "Barrichello", "Schumacher", e "Coulthard" podem-se definir a classe "Piloto".

Cada objeto deve ser criado antes de ser utilizado. O ato de criar um objeto é denominado instanciação. Todo objeto é instancia de uma classe. A instanciação de um objeto é feita através de um método especial denominado construtor. Todos os objetos instanciados terão a mesma estrutura de métodos e atributos de sua classe [CL2000].

Para Larman [CL2000], os atributos de um objeto são como variáveis que armazenam dados. Por exemplo, um objeto de uma classe "Piloto" pode conter os atributos "nome" e "idade", e os valores "Barrichello", "35". As operações são o que os objetos podem realizar. Para este mesmo objeto da classe "Piloto", pode-se ter, por exemplo, a operação "getPontos".

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

Quando uma operação é solicitada ao objeto, diz-se que este objeto recebeu uma mensagem. Esta mensagem pode trazer consigo parâmetros, que são valores passados ao objeto que a recebe. Por exemplo, traduzindo a mensagem “piloto.setIdade(36)”, significa-se estar solicitando ao objeto “piloto” para executar a operação “setIdade”, passando como parâmetro a idade “36”.

Conforme Larman [CL2000], o fato de uma classe reunir tanto as características quanto o comportamento dos objetos é chamado de encapsulamento. Assim, outros objetos só terão acesso às operações ou atributos que estiverem declarados como públicos.

Dá-se o nome de polimorfismo o fato de objetos de classes diferentes reagirem de forma particular a uma mesma operação [CL2000]. Por exemplo, a operação “getPontos” do objeto da classe “Piloto” tem uma codificação diferente do objeto de uma classe “Equipe”, embora os objetivos da operação sejam semelhantes.

Um outro conceito muito importante da OO é a herança. Com ela, uma classe pode herdar todas as operações e atributos de uma outra classe, acrescentando os seus próprios atributos e operações. A classe que herdou a estrutura é chamada de subclasse, enquanto a que cedeu a estrutura chama-se superclasse. Por exemplo, a classe “Pessoa” pode ser herdada pela classe “Piloto”, acrescentando atributos como “equipe”, “salário”, etc.

Uma classe pode ter um atributo que se refere a uma outra classe. A essa referência, dá-se o nome de associação. Por exemplo, o atributo “equipe” de uma classe “Piloto” pode referir a classe “Equipe”, que tenha mais atributos como “nome”, “pais”, “chefe”, etc.

3.2 PERSISTÊNCIA OBJETO-RELACIONAL

Os objetos em uma aplicação OO podem ser transientes ou persistentes. Transientes são os objetos que desaparecem ao término da execução do programa. Já os persistentes permanecem após o término da aplicação, podendo ser recuperados ao executar o aplicativo novamente.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

No processo de desenvolvimento de aplicações Orientadas a Objetos, nem sempre é dada a devida atenção à forma como a persistência de objetos será gerenciada e implementada. Na grande maioria dos sistemas, incluir código SQL (*Structured Query Language* – Linguagem de Consultas Estruturadas) para acesso ao Sistema Gerenciador de Banco de Dados (SGBD) em meio ao restante da lógica do sistema é a solução adotada, graças à rapidez de implementação. Esta é uma escolha perigosa: sua adoção implica, muitas vezes, no acoplamento do sistema ao SGBD utilizado, o que dificulta o processo de manutenção de código. Além disso, quaisquer mudanças na estrutura (ou mesmo na nomenclatura de colunas) das tabelas existentes no Banco de Dados trazem o caos à aplicação – todo o código SQL codificado na aplicação tem que ser reescrito, recompilado e testado novamente.

Para diminuir este acoplamento, surge uma segunda opção: separar o código SQL das classes da aplicação, de forma que as alterações no modelo de dados requerem modificações apenas nas classes de acesso a dados (*Data Access Classes*), restringindo o impacto das mudanças no sistema como um todo. Esta estratégia, trás um maior controle quanto ao escopo dos possíveis erros gerados por mudanças no esquema de dados do sistema. No entanto, apesar da limpeza de código e melhor divisão de responsabilidades trazidas pela adoção das *Data Classes*, a solução ainda não é a ideal, por ainda manter os dois mundos (objetos e dados relacionais) intimamente ligados.

A principal função de uma camada de abstração de acesso a dados é garantir aos desenvolvedores de software a total independência entre o modelo de objetos e o esquema de dados do banco, permitindo que a base ou detalhes do esquema de dados sejam substituídos sem impacto nenhum na aplicação. Mais que isso, uma Camada de Persistência pode permitir o armazenamento dos dados em outros tipos de bases (mecanismos de persistência) diferentes dos relacionais, incluindo os SGBDOO, Objeto-Relacional e arquivos XML, por exemplo.

3.2.1 Camadas de Persistência

Conceitualmente, uma Camada de Persistência de Objetos é uma biblioteca que permite a realização do processo de persistência (isto é, o

armazenamento e manutenção do estado de objetos em algum meio não-volátil, como um banco de dados) de forma transparente. Graças à independência entre a camada de persistência e o repositório (*backend*) utilizado, também é possível gerenciar a persistência de um modelo de objetos em diversos tipos de repositórios, teoricamente com pouco ou nenhum esforço extra. A utilização deste conceito permite ao desenvolvedor trabalhar como se estivesse em um sistema completamente orientado a objetos – utilizando métodos para incluir, alterar e remover objetos e uma linguagem de consulta para SGBDs Orientados a Objetos – comumente a linguagem OQL (*Object Query Language* – Linguagem de Consultas de Objetos) – para realizar consultas que retornam coleções de objetos instanciados.

3.2.2 Vantagens da utilização

As vantagens decorrentes do uso de uma Camada de Persistência no desenvolvimento de aplicações são evidentes: a sua utilização isola os acessos realizados diretamente ao banco de dados na aplicação, bem como centraliza os processos de construção de consultas (queries) e operações de manipulação de dados (*insert*, *update* e *delete*) em uma camada de objetos inacessível ao programador. Este encapsulamento de responsabilidades garante maior confiabilidade às aplicações e permite que, em alguns casos, o próprio SGBD ou a estrutura de suas tabelas possam ser modificados, sem trazer impacto à aplicação nem forçar a revisão e recompilação de códigos.

3.2.3 Requisitos de uma Camada de Persistência

Segundo Scott Ambler [SA2005], pesquisador e autor de diversos livros, uma Camada de Persistência real deve implementar as seguintes características:

- Dar suporte a diversos tipos de mecanismos de persistência: um mecanismo de persistência pode ser definido como a estrutura que armazenará os dados – seja ele um SGBD relacional, um arquivo XML ou um SGBDOO, por exemplo. Uma Camada de Persistência deve suportar a substituição deste mecanismo livremente e permitir a gravação de estado de objetos em qualquer um destes meios.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

- Encapsulamento completo da camada de dados: o usuário do sistema de persistência de dados deve utilizar-se, no máximo, de mensagens de alto nível como *save* ou *delete* para lidar com a persistência dos objetos, deixando o tratamento destas mensagens para a camada de persistência em si.
- Transações: ao utilizar-se da Camada de Persistência, o programador deve ser capaz de controlar o fluxo da transação – ou ter garantias sobre o mesmo, caso a própria Camada de Persistência preste este controle.
- Extensibilidade: A Camada de Persistência deve permitir a adição de novas classes ao esquema e a modificação fácil do mecanismo de persistência.
- Identificadores de Objetos: A implementação de algoritmos de geração de chaves de identificação garante que a aplicação trabalhará com objetos com identidade única e sincronizada entre o banco de dados e a aplicação.
- Cursores e Proxies: As implementações de serviços de persistência devem ter ciência de que, em muitos casos, os objetos armazenados são muito grandes – e recuperá-los por completo a cada consulta não é uma boa idéia. Técnicas como o "*lazy loading*" (carregamento tardio) utilizam-se dos proxies para garantir que atributos serão carregados à medida que forem importantes para o cliente e do conceito de cursores para manter registro da posição dos objetos no banco de dados (e em suas tabelas específicas).
- Registros: Apesar da idéia de trabalhar-se apenas com objetos, as camadas de persistência devem, no geral, dispor de um mecanismo de recuperação de registros - conjuntos de colunas não encapsuladas na forma de objetos, como resultado de suas consultas. Isto permite integrar as camadas de persistências a mecanismos de geração de relatórios que não trabalham com objetos, por exemplo, além de permitir a recuperação de atributos de diversos objetos relacionados com uma só consulta.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

- Arquiteturas Múltiplas: O suporte a ambientes de programas *stand-alone*, cenários onde o banco de dados encontra-se em um servidor central e mesmo arquiteturas mais complexas (em várias camadas) deve ser inerente à Camada de Persistência, já que a mesma deve visar a reusabilidade e fácil adaptação a arquiteturas distintas.
- Diversas versões de banco de dados e fabricantes: a Camada de Persistência deve tratar de reconhecer diferenças de recursos, sintaxe e outras minúcias existentes no acesso aos bancos de dados suportados, isolando isto do usuário do mecanismo e garantindo portabilidade entre plataformas.
- Múltiplas conexões: Um gerenciamento de conexões (usualmente utilizando-se de *pooling*) é uma técnica onde vários usuários utilizarão o sistema simultaneamente sem quedas de performance.
- Queries SQL: Apesar do poder trazido pela abstração em objetos, este mecanismo não é funcional em todos os casos. Para os casos extremos, a Camada de Persistência deve prover um mecanismo de queries que permita o acesso direto aos dados – ou então algum tipo de linguagem de consulta similar à SQL, de forma a permitir consultas com um grau de complexidade maior.
- Controle de Concorrência: Acesso concorrente a dados pode levar a inconsistência. Para prever e evitar problemas decorrentes do acesso simultâneo, a Camada de Persistência deve prover algum tipo de mecanismo de controle de acesso. Este controle geralmente é feito utilizando-se dois níveis – com o travamento pessimístico (*pessimistic locking*), as linhas no banco de dados relativas ao objeto acessado por um usuário são travadas e tornam-se inacessíveis a outros usuários até o mesmo liberar o objeto. No mecanismo otimístico (*optimistic locking*), toda a edição é feita em memória, permitindo que outros usuários possam alterar o objeto.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

4 SISTEMAS DISTRIBUIDOS

4.1 VISÃO GERAL SOBRE SISTEMAS DISTRIBUÍDOS

Sistemas Distribuídos consistem em uma coleção de computadores autônomos ligados por uma rede de comunicação [GC2005]. O uso de tais sistemas tem se expandido nos últimos anos principalmente devido ao contínuo barateamento e disponibilidade de hardware para computadores, bem como de meios físicos de comunicação. As vantagens de Sistemas Distribuídos incluem a possibilidade de seu crescimento incremental (ou seja, novos computadores e linhas de comunicação ser acrescidos ao sistema), a possibilidade de implementação de aplicações inerentemente distribuídas e a possibilidade de implementação de tolerância a falhas através da replicação de processos em unidades de computação distintas. Embora os hardwares desses sistemas estejam num estágio avançado de desenvolvimento, o mesmo não se pode afirmar em relação ao software devido à complexidade adicional inerente a sua distribuição.

4.2 OBJETOS DISTRIBUIDOS

Com o surgimento de processadores mais robustos e redes de computadores mais confiáveis, novas aplicações começaram a ser desenvolvidas tentando separar as camadas. As chamadas aplicações em duas camadas separavam em uma camada a apresentação e a lógica da aplicação e em outra camada os dados (banco de dados). Neste caso um microcomputador mantém em seus dispositivos de armazenamento locais a aplicação instalada (apresentação e lógica da aplicação). O banco de dados fica hospedado em uma máquina separada sendo acessado através da rede pela aplicação.

Atualmente fala-se em desenvolvimento em 03 (três) ou mais camadas, sendo que a única camada que seria mantida em um cliente é a de apresentação. As camadas da lógica da aplicação, regras de negócio e informações poderiam ser mantidas em uma ou mais máquinas através do sistema distribuído.

Isso é possível através do projeto de desenvolvimento orientado a objetos, no qual classes podem ser definidas para atuar nas diferentes camadas da aplicação. Quando em execução a aplicação pode instanciar objetos destas classes na mesma máquina ou em máquinas diferentes no sistema distribuído, tendo ainda o

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

sistema de banco de dados instalado em outra máquina do mesmo sistema distribuído.

No início tínhamos uma máquina centralizada realizando todo o processamento, seja aquele relacionado apenas à apresentação ou àquele relacionado ao acesso ao banco de dados ou qualquer outro. Hoje podemos ter aplicações desenvolvidas sob o enfoque de sistemas distribuídos, onde uma máquina realiza requisitos e recebe respostas; outra máquina processa a lógica da aplicação através de objetos específicos; outra ainda para armazenar as bases de dados através de um sistema gerenciador de banco de dados. As regras de negócio, se modeladas através de classes, poderiam ser também processadas por objetos específicos para tal e até mesmo mantidas em hardware diferente. Ou ainda, todos estes objetos poderiam ser mantidos em um mesmo hardware!

4.2.1 Tecnologias Orientadas a Objetos Distribuídos

As tecnologias mais importantes e mais usadas são:

- RMI – *Remote Invocation Method* – É uma API padrão para construir sistemas distribuídos feitos em Java. O sistema de invocação remota de Métodos (RMI) Java permite, a um objeto que esteja sendo executado em uma Maquina Virtual Java (VM), poder chamar os métodos de um outro objeto que esteja em uma outra VM diferente. Esta tecnologia é associada à linguagem de programação Java, que permite a comunicação entre os objetos criados nesta linguagem.
- CORBA – *Common Object Request Broker Architecture* – tecnologia introduzida pelo Grupo de Administração de Objetos (OMG), criada para estabelecer uma plataforma para a gerência dos objetos remotos independentes da linguagem de programação.
- J2EE – *Java 2 Enterprise Edition* – uma especificação para servidores de aplicação que definem padrões de suporte a componentes e serviços. Ela contém um pacote de *APIs* e ferramentas para desenvolver componentes que rodam nesses servidores.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

A especificação J2EE é basicamente implementada em cima de RMI. É uma plataforma mais complexa de se trabalhar e mais difícil de implementar, além de ser um servidor de aplicação que necessita de mais recursos da máquina. O *container EJB* (*Enterprise JavaBeans*), que faz parte da plataforma J2EE, é o núcleo de uma aplicação distribuída. É desenvolvido usando RMI que gera objetos CORBA (podem ser chamados por clientes CORBA, mesmos clientes que são escritos em outras linguagens).

Como o objetivo deste trabalho visa uma arquitetura leve de fácil implementação, será apresentado como fazer uma aplicação com essas características usando apenas RMI, deixando ela mais simples e eficiente.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

5 DISTRIBUIÇÃO DE OBJETOS PERSISTENTES

Para iniciar um projeto de distribuição de objetos persistentes deve-se passar por três fases: o projeto conceitual, o mapeamento de objeto para tabelas e a distribuição do sistema.

No exemplo apresentado nesse trabalho o projeto conceitual será feito usando uma ferramenta UML chamada CaseStudio. No mapeamento objeto-relacional será usado o *framework Hibernate*, uma ferramenta Java poderosa para mapeamento objeto-relacional. E para finalizar, na distribuição do sistema será usado o RMI, criando uma aplicação leve sem uso de uma especificação J2EE que deixa o servidor de aplicação muito mais complexo.

5.1 ESQUEMA CONCEITUAL

No projeto conceitual, é modelado um esquema que apresenta uma abstração do mundo real. No exemplo que será mostrado modela-se um esquema de Formula 1. Na figura 2 está o esquema que será aproveitado pelos capítulos posteriores:

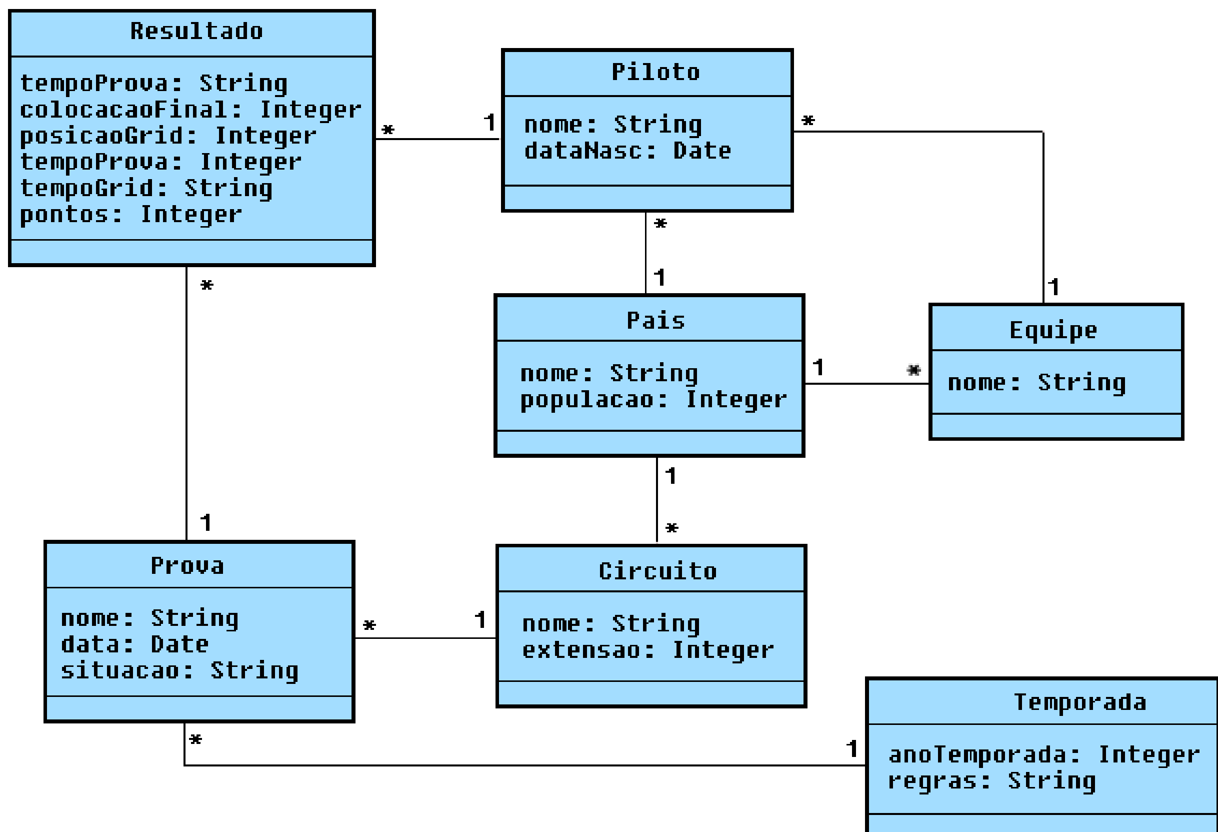


Figura 2 – Esquema conceitual de “Formula 1”

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

Com um modelo de objetos criado, vejamos como essas classes poderiam ser transformadas em tabelas. Na figura 3 vemos o diagrama entidade-relacionamento do esquema mostrado na figura 2.

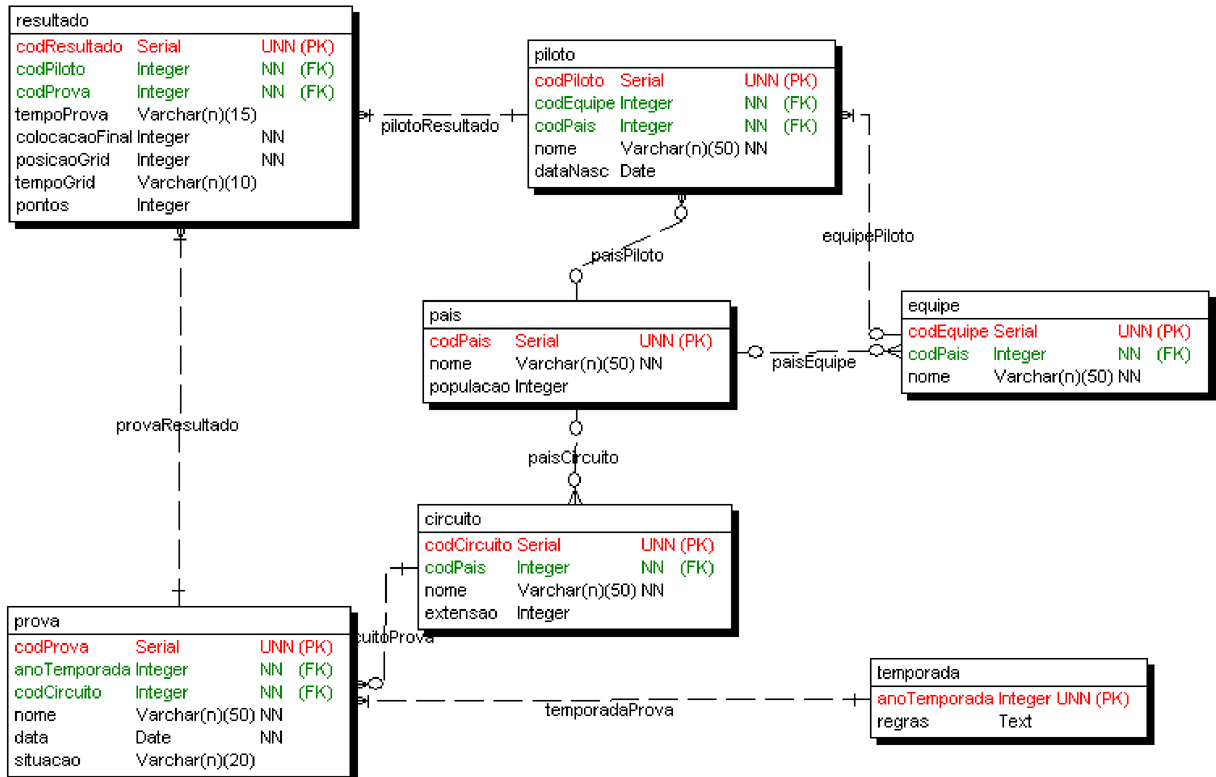


Figura 3 – Diagrama entidade-relacionamento do esquema conceitual de “Formula 1”

Quando estiver mapeando as classes do modelo para o banco de dados, por questão de organização, é aconselhável usar o mesmo padrão de nomeclatura Java para relações e atributos, com os mesmos nomes das classes e suas propriedades.

5.2 MAPEANDO OBJETO PARA TABELAS

Para permitir a correta persistência de objetos em um SGBDR, algum acordo deve ser feito no tocante à forma como os dados serão armazenados. Existem diversas técnicas que permitem o mapeamento de objetos, cada qual com suas vantagens e desvantagens sobre as demais. Em geral, uma Camada de Persistência implementa uma destas técnicas, de forma que o desenvolvedor de software, ao escolher o mecanismo de persistência com o qual trabalhará, sabe como deve organizar as tabelas em seu banco de dados para suportar o esquema de objetos desejado. Nesse caso será usado o *framework Hibernate*.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

Hibernate é uma ferramenta *open-source* que permite a persistência transparente de objetos em bases de dados utilizando JDBC. Ele facilita a comunicação entre aplicação Java e banco de dados, como manipulação de dados, gerenciamento de transações, *pooling* de conexões, nos permite herança, associação um-pra-muitos, um-pra-um ou muitos-pra-muitos.

Antes de começar a fazer os mapeamentos do *Hibernate*, faz-se necessário rever um conceito de banco de dados: a identidade. Para um banco de dados, o modo de diferenciar uma linha das outras, é usando chaves, de preferência chaves não naturais, como por exemplo, a coluna “codPais” da tabela “pais” mostrada na figura 3. No nosso modelo OO, a identidade não é encontrada dessa forma. Em Java, nós definimos a identidade dos objetos sobrescrevendo o método “Object.equals(Object object)”, do modo que nos convier. A implementação “default”, define a identidade através da posição de memória ocupada pelo objeto.

Não podemos usar o método “equals()” no banco de dados, porque o banco de dados não sabe que temos objetos, pois não estamos trabalhando com BDOO. A solução é adicionar aos nossos objetos um identificador não natural, como os que nós encontramos no banco de dados, porque assim o banco de dados e o próprio *Hibernate* vão ser capazes de diferenciar os objetos e montar os seus relacionamentos. Nós fazemos isso adicionando uma propriedade chamada, por exemplo, de “codPais” do tipo Long a todas as nossas classes, como no exemplo de código a seguir:

```
package formula;
import java.rmi.RemoteException;
import java.util.HashSet;
import java.util.Set;
public class Pais extends ImplBO implements PaisInterface {
        private Long codPais;
        private String nome;
        private Long populacao;
        private Set equipeH;
        private Set pilotoH;
        private Set circuitoH;
        //métodos getters e setters das propriedades
}
```

Figura 4 – Exemplo do código de parte da classe Pais

5.2.1 Configuração do *Hibernate*

Toda a configuração do *Hibernate* é feita através de arquivos XML [HIBERNATE2002], os quais contêm mapeamentos de tabelas/Java, detalhes de

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

pooling de conexões, qual banco de dados será usado, dentre outras. Combinados, estes arquivos fornecem total configurabilidade à aplicação.

O *Hibernate* é a camada que conecta ao banco de dados. Para isso devemos inicialmente configurá-lo. As conexões com banco de dados são feitas através do driver JDBC (*Java Database Connection*) [HIBERNATE2002]. Estaremos utilizando o PostgreSQL como banco de dados para o projeto utilizado neste trabalho.

O arquivo de configuração do *Hibernate* pode ser uma *property* ou um XML. É preferível utilizar XML para melhor visualização e manutenção do arquivo posteriormente. Para o arquivo XML, deve-se utilizar um DTD diferente, o *hibernate-configuration-2.0.dtd*, o qual manterá a integridade do XML de configuração.

Abaixo segue o arquivo de configuração para nosso projeto. Este deve ter o sufixo cfg.xml. Neste caso ficará *hibernate.cfg.xml*:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 2.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
  <session-factory>

    <!-- Informações de conexão com o PostgreSQL -->
    <property
     name="connection.driver_class">org.postgresql.Driver</property>
    <property
     name="connection.url">jdbc:postgresql://localhost/formula1</property>
    <property name="connection.username">formula</property>
    <property name="connection.password">formula</property>
    <!-- SQL dialect, específico para o PostgreSQL -->
    <property
     name="dialect">net.sf.hibernate.dialect.PostgreSQLDialect</property>
    <!-- Mostrar os comandos SQL executados? -->
    <property name="show_sql">false</property>
    <!-- Apaga e recria o schema do BD ao iniciar -->
    <property name="hbm2ddl.auto">create</property>
    <property name="connection.pool_size">0</property>
    <!-- Arquivos de mapeamento das Classes -->
    <mapping resource="formula/Circuito.hbm.xml"/>
    <mapping resource="formula/Equipe.hbm.xml"/>
    <mapping resource="formula/Pais.hbm.xml"/>
    <mapping resource="formula/Piloto.hbm.xml"/>
    <mapping resource="formula/Prova.hbm.xml"/>
    <mapping resource="formula/Resultado.hbm.xml"/>
    <mapping resource="formula/Temporada.hbm.xml"/>

  </session-factory>
</hibernate-configuration>
```

Figura 5 – Exemplo do arquivo de configuração do *Hibernate* (hibernate.cfg.xml)

Algumas propriedades já estão comentadas na figura 5. Outras propriedades importantes estão abaixo com seus possíveis valores:

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

a) Dialect: É uma classe que faz o cruzamento das funções do *Hibernate* com as funções do banco de dados. Os possíveis valores podem ser vistos na figura 6.

```
DB2 - org.hibernate.dialect.DB2Dialect
HypersonicSQL - org.hibernate.dialect.HSQLDialect
Informix - org.hibernate.dialect.InformixDialect
Ingres - org.hibernate.dialect.IngresDialect
Interbase - org.hibernate.dialect.InterbaseDialect
Pointbase - org.hibernate.dialect.PointbaseDialect
PostgreSQL - org.hibernate.dialect.PostgreSQLDialect
Mckoi SQL - org.hibernate.dialect.MckoiDialect
Microsoft SQL Server - org.hibernate.dialect.SQLServerDialect
MySQL - org.hibernate.dialect.MySQLDialect
Oracle (any version) - org.hibernate.dialect.OracleDialect
Oracle 9 - org.hibernate.dialect.Oracle9Dialect
Progress - org.hibernate.dialect.ProgressDialect
FrontBase - org.hibernate.dialect.FrontbaseDialect
SAP DB - org.hibernate.dialect.SAPDBDialect
Sybase - org.hibernate.dialect.SybaseDialect
Sybase Anywhere - org.hibernate.dialect.SybaseAnywhereDialect
```

Figura 6 – Possíveis valores para a propriedade dialect

b) hbm2ddl.auto: Esta propriedade indica ao *Hibernate* que ele deverá atualizar o *schema* do banco de dados ao iniciar a aplicação, ou seja, atualizar tabelas, índices, entre outros, com seus arquivos de mapeamento (*mapping resource*) que veremos mais adiante. Por exemplo, se no arquivo de mapeamento tiver um campo e no banco de dados não tiver, ele fará um ALTER TABLE para adicionar este campo. Os possíveis valores são:

- **create** - apenas cria as tabelas e índices, se já tiverem sido criados não faz nada;
- **create-drop** - apaga o *schema* e o recria;
- **update** - faz sincronismo entre XML de mapeamento e banco de dados.

c) show_sql: Esta propriedade indica ao *Hibernate* se deverá ou não registrar todos os comandos SQL. Os possíveis valores são: *true* ou *false*.

5.2.2 Criação do Mapeamento

O primeiro mapeamento abordado é o da classe País e do seu relacionamento com as classes Equipe, Piloto e Circuito. No modelo que foi mostrado na figura 2, um país tem várias equipes, pilotos e circuitos. Vejamos o mapeamento para essa classe:

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

```
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>

  <class name="formula.Pais" table="pais">

    <id name="codPais" column="codPais" type="long">
      <generator class="increment" />
    </id>

    <property name="nome" type="string" />
    <property name="populacao" type="long" />

    <set name="equipeH" inverse="true" lazy="true" order-by="nome asc">
      <key column="codPais"/>
      <one-to-many class="formula.Equipe"/>
    </set>

    <set name="pilotoH" inverse="true" lazy="true" order-by="nome asc">
      <key column="codPais"/>
      <one-to-many class="formula.Piloto"/>
    </set>

    <set name="circuitoH" inverse="true" lazy="true" order-by="nome asc">
      <key column="codPais"/>
      <one-to-many class="formula.Circuito"/>
    </set>

  </class>
</hibernate-mapping>
```

Figura 7 – Exemplo de mapeamento da classe Pais (Pais.hbm.xml)

O arquivo de mapeamento também é um arquivo XML. Ele define as propriedades e os relacionamentos de uma classe para o *Hibernate*. Este arquivo pode conter classes, classes componentes e queries em HQL (*Hibernate Query Language*) ou em SQL. No exemplo da figura 7, temos apenas uma classe sendo mapeada no arquivo, a classe Pais. O arquivo XML começa normalmente com as definições da DTD (*Document Type Definitions*) e do nó raiz, o *<hibernate-mapping>*, depois vem o nó que nos interessa neste caso, *<class>*.

No nó *<class>* nós definimos a classe que está sendo mapeada e para qual tabela ela vai ser mapeada. O único atributo obrigatório deste nó é "*name*", que deve conter o nome completo da classe (com o pacote, se ele não tiver sido definido no atributo "*package*" do nó *<hibernate-mapping>*), se o nome da classe for diferente do nome da tabela, você pode colocar o nome da tabela no atributo "*table*".

Mais adiante, temos o nó *<id>* que é o identificador dessa classe no banco de dados. Neste nó, nós definimos a propriedade que guarda o identificador do objeto, no atributo "*name*", que nesse caso é "codPais". Se o nome da coluna no

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

banco de dados for diferente da propriedade do objeto, este deve ser definido no atributo "*column*" (por exemplo, codPais é diferente de codpais). Ainda dentro deste nó, nós encontramos mais um nó, o *<generator>*. Este nó guarda a informação de como os identificadores (as chaves do banco de dados) são gerados. Existem diversas classes de geradores, que são definidas no atributo "*class*" do nó. Nesse exemplo o gerador usado é o "*increment*", que incrementa um ao valor da chave sempre que insere um novo objeto no banco. Esse gerador costuma funcionar normalmente em todos os bancos.

Os próximos nós do arquivo são os *<property>* que indicam propriedades simples dos nossos objetos, como *Strings*, *Long*, *Date*, *Timestamp*, *Integer*, *Currency* e outros.

Neste nó os atributos mais importantes são "*name*", que define o nome da propriedade, "*column*", para quando a propriedade não tiver o mesmo nome da coluna na tabela, e "*type*" para definir o tipo do objeto que a propriedade guarda. Normalmente, o próprio *Hibernate* é capaz de descobrir qual é o tipo de objeto que a propriedade guarda não sendo necessário ter o atributo "*type*" no arquivo de configuração. Nós definimos as duas propriedades simples da nossa classe: "nome" e "populacao".

Os últimos três nós do arquivo são os *<set>*, que definem os relacionamentos de 1-para-N que a classe Pais tem com as classes Equipe, Piloto e Circuito. Uma "set" ou "conjunto" representa uma coleção de objetos não repetidos, que podem ou não estar ordenados (dependendo da implementação da interface *java.util.Set* escolhida). Quando você adiciona um nó deste tipo em um arquivo de mapeamento, você está indicando ao *Hibernate* que o seu objeto tem um relacionamento 1:N ou N:N com outro objeto.

No nó *<set>* o primeiro atributo a ser encontrado é "*name*" que assim como nos outros nós, define o nome da propriedade que está sendo tratada. Já o outro atributo, "*inverse*", define como o *Hibernate* vai tratar a inserção e retirada de objetos nessa associação. Quando um lado da associação define o atributo "*inverse*" para "*true*", ele está indicando que apenas quando um objeto for inserido do "outro lado" da associação ele deve ser persistido no banco de dados.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

O atributo “*lazy*” é uma outra característica importante do *Hibernate*. Por exemplo, a classe Pais refere-se a uma coleção de Equipe. Quando quiser exibir somente o nome dos países, não é necessário que as equipes sejam carregadas. Marcando esse atributo igual a “*true*”, a coleção Equipe não será carregada enquanto ela não receber nenhuma requisição. Seguindo no código, mais a frente tem o atributo “*order-by*” que serve para ordenar a coleção.

Dentro do nó *<set>*, aparecem mais dois nós desconhecidos: *<key>* e *<one-to-many>*. O nó *<key>* representa a coluna, no atributo *<column>*, da tabela relacionada que guarda a chave primária da tabela Pais. No outro nó, *<one-to-many>*, nós definimos a classe (com o pacote) a qual pertence essa coleção de objetos.

O mapeamento da classe Pais, faz referência a outras classes que o *Hibernate* ainda não conhece: as classes Equipe, Piloto e Circuito. A figura 8 mostra como fica o mapeamento de uma delas, pois as outras são semelhantes.

```
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>

  <class name="formula.Equipe" table="equipe">

      <id name="codEquipe" column="codEquipe" type="long">
        <generator class="increment" />
      </id>

      <property name="nome" type="string" />

      <many-to-one name="pais" column="codPais"
                   class="formula.Pais" not-null="true" />

      <set name="pilotoH" inverse="true" lazy="true" order-by="nome asc">
        <key column="codEquipe" />
        <one-to-many class="formula.Piloto" />
      </set>

  </class>
</hibernate-mapping>
```

Figura 8 – Exemplo de mapeamento da classe Equipe (Equipe.hbm.xml)

A maior parte do código já é conhecida, mas ainda existem algumas coisas que precisam ser esclarecidas. Equipe possui um relacionamento muitos-para-um, com a classe Pais. Para tornar essa associação bidirecional nós vamos utilizar o nó *<many-to-one>*. Neste nó, nós inserimos os atributos “*name*”, que recebe o nome da propriedade, “*class*”, que recebe o nome da classe da

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

propriedade, "*column*", que recebe o nome da coluna nesta tabela que guarda a chave estrangeira para a outra tabela do relacionamento e "*not-null*", que indica a obrigatoriedade de toda Equipe ter um Pais.

O mapeamento do resto das classes do modelo mostrado na figura 2 é semelhante ao mapeamento das classes que já foi mostrado.

5.2.3 Criação das Classes de Persistência

Agora que o *Hibernate* está mapeado e configurado, o próximo passo é criar todas as classes, as quais serão baseadas nos arquivos XML que foram criados anteriormente. Na figura 9 está um exemplo da classe Equipe, a qual pode ser tomada como base para criação de todas as outras classes do exemplo que está sendo exposto.

```java
package formula;

import java.rmi.RemoteException;
import java.util.HashSet;
import java.util.Set;

public class Equipe extends ImplBO
       implements EquipeInterface {
  private Long codEquipe;
  private String nome;
  private PaisInterface pais;
  private Set pilotoH;

  public Equipe() throws
                  RemoteException {
    super();
    pilotoH = new HashSet();
  }

  public Long getCodEquipe() {
    return codEquipe;
  }

  public void setCodEquipe(Long
                       codEquipe) {
    this.codEquipe = codEquipe;
  }

  public String getNome() {
    return nome;
  }

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

  public PaisInterface getPais() {
    return pais;
  }
```

```java
  public void setPais(PaisInterface
       pais) throws RemoteException {

    if (pais.getClass().getName()
            .endsWith("_Stub")) {

      Pais novoPais = new Pais();
      novoPais = (Pais) novoPais
        .findById(pais.getCodPais());
      pais = novoPais;
    }

    if (pais != null) {
      if (this.pais != null) {
        this.pais.getEquipeH()
                 .remove(this);
      }
      pais.getEquipeH().add(this);
    }
    this.pais = pais;
  }

  public Set getPilotoH() {
    return pilotoH;
  }

  public void setPilotoH(Set pilotoH){
    this.pilotoH = pilotoH;
  }

  public Set getPiloto() {
    return returnRealSet(pilotoH);
  }

  public void setPiloto(Set piloto) {
    setPilotoH(piloto);
  }
}
```

Figura 9 – Exemplo do arquivo Equipe.java que possui a implementação da classe Equipe

Posteriormente serão aplicados os conceitos para distribuição dos objetos persistentes, então um detalhe que deve ser observado em todas as classes é que

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

elas devem implementar uma interface. No exemplo da figura 9 é implementado a interface “EquipeInterface”. Outro detalhe a ser observado é que a classe “Equipe” herda os métodos da classe “ImplBO” (figura 10) que será mostrada mais na frente.

Como a classe “Equipe” terá seus objetos acessados remotamente, os métodos construtores devem ter um tratamento de exceções realizadas através da classe “java.rmi.RemoteException”, para que as exceções sejam repassadas remotamente. Isso é feito com a expressão “*trows RemoteException*”.

Os métodos que foram implementados na classe Equipe, são os “*getters*” e “*setters*” que serão usados pelo mapeamento do *Hibernate* para a criação dos objetos. Como o *Hibernate* cria os objetos usando a *API* R*eflection*, devem ser seguidos os padrões de visibilidade de métodos, construtores, etc. para que não ocorram erros.

Na figura 10 é apresentado um exemplo da classe “ImpBO” que tem seus métodos herdados pelas classes do mapeamento. Essa classe possui os métodos responsáveis por salvar, remover e pesquisar por um objeto.

```java
package formula;
import java.rmi.RemoteException;
import java.rmi.server
               .UnicastRemoteObject;
import java.util.*;
public abstract class ImplBO
       extends UnicastRemoteObject
       implements ImplBOInterface {
  private DAO dao;
  public ImplBO() throws
                  RemoteException {
    super();
    dao = new DAO();
  }
  public void save() {
    dao.save(this);
  }
  public void saveNew() {
    dao.saveNew(this);
  }
  public void delete() {
    dao.delete(this);
  }
  public List findAll() {
    Iterator i;
    i = dao.findByParam(null,
                   this.getClass());
    List lista = new ArrayList();
    if (i != null) {
      while (i.hasNext()) {
        lista.add(i.next());
      }
    }
    return lista;
  }

  public Object findById(Long id) {
    return dao.findById(id,
                   this.getClass());
  }

  public List findByParam(List p) {
    Iterator i;
    i=dao.findByParam(p.iterator(),
                   this.getClass());

    List lista = new ArrayList();
    if (i != null) {
      while (i.hasNext()) {
        lista.add(i.next());
      }
    }
    return lista;
  }

  public Set returnRealSet(Set
                   hibernateSet) {
    Iterator i;
    Set realSet;

    i = hibernateSet.iterator();
    realSet = new LinkedHashSet();

    while (iterator.hasNext()) {
      realSet.add(iterator.next());
    }
    return realSet;
  }
}
```

Figura 10 – Exemplo do arquivo ImplBO.java que possui a implementação da classe ImplBO

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

ImpBO também deve implementar uma interface e deve ser subclasse da classe UnicastRemoteObject que é derivada da classe RemoteServer. A RemoteServer é uma classe abstrata, utilizada para a implementação de RMI pela linguagem Java. Esta classe inicia uma *thread*, que mantém os objetos das classes do mapeamento ativos, a fim de que os clientes possam se conectar ao programa.

No *Hibernate*, assim como no JDBC, existem os conceitos de sessão e transação. Uma sessão é uma conexão aberta com o banco de dados, onde nós podemos executar queries, inserir, atualizar e deletar objetos. Já a transação é a demarcação das ações, ela faz o controle do que acontece. Se forem encontrados problemas, as transações do *Hibernate* também podem fazer um *roolback*, assim como uma transação do JDBC.

Para que a aplicação não fique criando e fechando sessões para cada requisição que é feita, é criada uma classe (figura 11) com métodos auxiliares para fazer o controle de sessão, fazendo com que seja aberta apenas uma sessão.

```
package formula;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import net.sf.hibernate.HibernateException;
import net.sf.hibernate.Session;
import net.sf.hibernate.SessionFactory;
import net.sf.hibernate.cfg.Configuration;

public class HibernateUtil {
  private static Log log = LogFactory.getLog(HibernateUtil.class);
  private static final SessionFactory sessionFactory;

  static {
    try {
      sessionFactory = new Configuration().configure()
                                          .buildSessionFactory();
    } catch (Throwable ex) {
      log.error("Initial SessionFactory creation failed.", ex);
      throw new ExceptionInInitializerError(ex);
    }
  }

  public static final ThreadLocal session = new ThreadLocal();

  public static Session currentSession() throws HibernateException {
    Session s = (Session) session.get();
    if (s == null) {
      s = sessionFactory.openSession();
      session.set(s);
    }
    return s;
  }

  public static void closeSession() throws HibernateException {
    Session s = (Session) session.get();
    session.set(null);
    if (s != null) s.close();
  }
}
```

Figura 11 – Exemplo do arquivo HibernateUtil.java que possui a classe que gerencia as sessões

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

O método “currentSession()” verifica se já possui alguma sessão aberta e, caso não possua, abre uma nova sessão e retorna ao cliente. O método “closeSession()” serve para fechar a sessão atual.

Essa classe também é responsável por carregar todos os arquivos XML de mapeamento e configuração do *Hibernate*. Caso um dos arquivos XML esteja com algum erro, essa classe gera uma exceção.

5.3 DISTRIBUINDO OS OBJETOS PERSISTENTES

Com RMI é possível transportar objetos pela rede e também chamar métodos que estejam em outro computador, mantendo o objeto na máquina remota. Assim o sistema de “Formula 1” poderá rodar toda sua lógica de negócio em uma única máquina denominada servidor, enquanto os clientes poderão acessar de qualquer parte da rede ou até pela internet. Ou seja, todas as classes do *Hibernate* e todas as classes responsáveis pelo mapeamento objeto-relacional estarão apenas no servidor, enquanto as máquinas clientes terão apenas as interfaces e os *Stub's* das classes que terão seus objetos distruibuidos.

Para um cliente invocar um método RMI, ele cria um objeto *Stub* que encapsula o pedido do cliente em um pacote de bytes e envia para o servidor. Já para o servidor receber a informação que está no *Stub*, ele cria um objeto *Skeleton*. O *Skeleton* decodifica os parâmetros, chama o método desejado, captura o valor de retorno ou exceção, codifica e retorna o valor codificado para o cliente. O *Stub* decodifica o valor de retorno do servidor, que pode ser uma exceção, mas que em geral é o retorno da chamada do método remoto.

Como já foi dito anteriormente, para realizar uma operação RMI, é preciso criar, tanto do lado do cliente, quanto do lado do servidor, uma interface para o objeto que será referenciado contendo todos os métodos que serão acessados remotamente. Isto porque o cliente, embora não tenha o objeto real, que está no servidor, deve saber as ações que pode executar sobre este objeto (o protocolo).

Na figura 12 é mostrado um exemplo de interface. É mostrado o código do arquivo “*EquipeInterface.java*” que é implementada pela classe Equipe que foi mostrado anteriormente na figura 9.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

```
package formula;
import java.rmi.RemoteException;
import java.util.Set;
public interface EquipeInterface extends ImplBOInterface {
        public Long getCodEquipe() throws RemoteException;
        public void setCodEquipe(Long codEquipe) throws RemoteException;
        public String getNome() throws RemoteException;
        public void setNome(String nome) throws RemoteException;
        public PaisInterface getPais() throws RemoteException;
        public void setPais(PaisInterface pais) throws RemoteException;
        public Set getPiloto() throws RemoteException;
        public void setPiloto(Set piloto) throws RemoteException;
        public Set getPilotoH() throws RemoteException;
}
```

Figura 12 – Exemplo da interface EquipeInterface que é implementada pela classe Equipe

Todas as interfaces das classes do mapeamento devem herdar os métodos da interface “ImplBOInterface” que aparece na figura 13, pois os métodos dessa interface também deveram ser acessíveis para o cliente.

```
package formula;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;
public interface ImplBOInterface extends Remote {
        public void save() throws RemoteException;
        public void saveNew() throws RemoteException;
        public void delete() throws RemoteException;
        public List findAll() throws RemoteException;
        public Object findById(Long Id) throws RemoteException;
        public List findByParam(List params) throws RemoteException;
}
```

Figura13 – Exemplo da interface ImplBOInterface que é implementada pela classe ImplBO

Um detalhe importante é que essa interface deve ser derivada da classe “java.rmi.Remote” e todas as outras devem ter o tratamento de suas exceções realizadas através da “java.rmi.RemoteException”, para que as exceções sejam repassadas remotamente.

As outras interfaces do exemplo que está sendo exposto deverão ser semelhantes à da figura 12, ou seja, devem possui todos os métodos que serão acessados remotamente e deve ser implementada por uma classe que terá o objeto transportado do servidor para o cliente.

5.3.1 Criação da camada de distribuição dos objetos persistentes

O primeiro passo para iniciar a distribuição dos objetos é a criação de uma classe semelhante a um *Design Pattern Factory Method*, onde será a única responsável pela instanciação dos objetos que serão distribuídos. Essa classe (figura 14) se chama “Formula” e ela será o único ponto de acesso a todos os

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

objetos persistentes e todas as classes que foram criadas anteriormente. Formula também é uma subclasse da classe “UnicastRemoteObject”.

```
package formula;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Formula extends UnicastRemoteObject implements
                                                       FormulaInterface {
  public Formula() throws RemoteException {
    super();
  }

  public Object getObject(String classe) throws RemoteException {
    Object obj = null;
    try {
      obj = (Object) Class.forName(classe).newInstance();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    } catch (InstantiationException e) {
      e.printStackTrace();
    } catch (IllegalAccessException e) {
      e.printStackTrace();
    }
    return obj;
  }
}
```

Figura 14 – Exemplo do arquivo Formula.java responsável pela instanciação dos objetos distribuídos

Essa classe implementa uma interface chamada FormulaInterface. Essa interface possui apenas o método “getObject(String classe)” que será o responsável por retornar uma instancia da classe com nome igual ao parâmetro “classe”.

A classe Formula fará o uso da *API Reflection* para a instanciação das classes remotas. Após essa classe retornar uma nova instancia da classe que foi passada pelo parâmetro, será possível criar, alterar, pesquisar ou fazer qualquer tipo de manipulação do objeto em questão.

O segundo passo para a distribuição dos objetos é criar uma classe servidor (figura 15), a qual registrará no sistema operacional uma instancia da classe Formula que foi mostrada na figura 14, para que possa ser acessado remotamente pelos clientes.

```
package formula;

import java.rmi.Naming;

public class Server {
  public static void main (String[] argv) {
    try {
      Naming.rebind ("Formula", new Formula());
      System.out.println ("Servidor iniciado!");
    } catch (Exception e) {
      System.out.println ("Server erro: " + e.toString());
    }
  }
}
```

Figura 15 – Exemplo da classe Server (Server.java) que registra o objeto Formula

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

O objeto “new Formula()”, passa a ser construído, com o construtor da classe, e cadastrado no sistema operacional, através do método rebind(String str, Remote r) da classe Naming. A classe Naming fornece um mecanismo para obter referencias a objetos remotos baseados na sintaxe da URL. Onde uma URL para objetos remotos é usualmente especificada pelo h*ost*, porta e nome do objeto remoto: rmi://host:porta/nome.

Continuando com a distribuição dos objetos, será necessária a criação de uma classe que faça o gerenciamento da conexão do cliente com o servidor.

Essa classe terá uma função parecida com a classe HibernateUtil que foi criada anteriormente para gerenciamento da conexão do *Hibernate* com o banco de dados, ou seja, ela fará com que o cliente não abra uma conexão, sempre que precisar, do objeto Fomula que foi criado anteriormente.

Abaixo está um exemplo do arquivo “*ObjetoRemoto.java*” que é o arquivo que contém a classe que fará o gerenciamento das conexões dos clientes:

```
package formula;

import java.rmi.Naming;

public class ObjetoRemoto {
  private static ObjetoRemoto objeto = null;
  private static FormulaInterface formula;

  public static FormulaInterface instancia() {
    if (objeto == null)
      objeto = new ObjetoRemoto();
    return ObjetoRemoto.formula;
  }

  private ObjetoRemoto() {
    try {
      formula = (FormulaInterface) Naming.lookup("//localhost/Formula");
    } catch (Exception e) {
      System.out.println("Erro: Classe não encontrada!");
    }
  }
}
```

Figura 16 – Classe ObjetoRemoto responsável pelo gerenciamento da conexão do cliente

O único método acessível dessa classe é o “instancia()”. Ele cria, caso ainda não exista, um novo objeto do tipo “ObjetoRemoto” e retorna o objeto remoto que foi criado pela classe Formula.

O construtor da classe ObjetoRemoto é o único responsável pela conexão do cliente ao servidor. O método lookup(String str) da classe Naming obtém uma instancia do objeto registrado em “localhost” que está “escutando” na porta *default*

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

com nome Formula. Note que o resultado de "*Naming.lookup*" deve ter um tratamento de exceção para o caso do endereço ou o objeto não ser localizado.

5.4 IMPLEMENTAÇÃO DO CLIENTE

Após ter criado a camada de distribuição o próximo passo é criar a interface gráfica do programa que ficará do lado do cliente. Para fazer isso pode ser usado o modo texto, web ou utilizando a biblioteca Swing. Usando qualquer um desses tipos terá que fazer o uso da classe ObjetoRemoto (figura 16) para conectar-se ao servidor, e usar o método getObject da classe remota Formula para instanciação dos objetos remotos.

A figura abaixo possui um pequeno exemplo, em modo texto, que cria um novo país e uma nova equipe que pertence ao país criado anteriormente. Depois lista todos os paises com suas respectivas equipes:

```
package formula;

import java.rmi.RemoteException;
import java.util.Iterator;
import java.util.List;
import java.util.Set;

public class Client {
  public static void main (String[] argv) throws RemoteException {

    FormulaInterface formula = ObjetoRemoto.instancia();

    PaisInterface pais = (PaisInterface) formula.getObject("pais");
    pais.setNome("ITÁLIA");
    pais.setPopulacao(new Long("56204592"));
    pais.save();

    EquipeInterface equipe = (EquipeInterface) formula.getObject("equipe");
    equipe.setNome("FERRARI");
    equipe.setPais(pais);
    equipe.save();

    //Lista todos os paises com suas equipes
    Iterator paises = ((List) pais.findAll()).iterator();
    while (paises.hasNext()) {
      pais = (PaisInterface) paises.next();
      System.out.println("Pais: " + pais.getNome());

      System.out.println("Equipes:");
      Iterator equipes = ((Set) pais.getEquipe()).iterator();
      while (equipes.hasNext()) {
        equipe = (EquipeInterface) equipes.next();
        System.out.println("    - " +equipe.getNome());
      }
    }

    System.exit(1);
  }
}
```

Figura 17 – Arquivo Client.java mostra um exemplo em modo texto de "cliente"

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

O primeiro método a ser chamado é o método "instancia()" da classe "ObjetoRemoto" que é o responsável por iniciar a conexão com o servidor. Caso ocorra erro ao tentar conectar-se, será retornada uma exceção. A partir daí é instanciado um novo objeto da classe Pais e usado o método "save()" para salvar o novo objeto no banco. A mesma coisa é feita com a classe Equipe.

Um detalhe importante a ser observado é quando vai listar as equipes de um determinado país. Em vez de usar "pais.getEquipeH()" deve ser usado o método "getEquipe()" pois "getEquipeH()", que é o método usado pelo *Hibernate*, retorna uma classe Set do *Hibernate*, ou seja, daria um erro de "*classe not found*" pois o cliente não conhece nenhuma classe do *Hibernate*. O método "getEquipe()" transforma o tipo Set do *Hibernate* em Set do Java. Após isso ele retorna a coleção Set ao cliente.

Para fazer com que o exemplo acima rode sem problemas primeiramente deve-se iniciar o "rmiregistry" no servidor. O *rmiregistry* faz o registro do objeto remoto usando uma porta especificada ou, se omitida, na porta padrão 1099. Após ter o *rmiregistry* iniciado, basta iniciar a classe Server:

```
Rmiregistry [porta]
java formula.Server
```

Se tudo iniciar sem retornar nenhuma exceção, falta apenas agora executar o cliente. Para fazer isso é simples:

```
java formula.Client
```

Após a execução, será criado no banco de dados um novo pais e equipe. Após isso, mostrará na tela todos os paises com suas respectivas equipes que estão atualmente no banco de dados.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

6 CONCLUSÃO

Após o estudo do Mapeamento Objeto-Relacional com distribuição dos objetos persistentes, com base em livros, em sites de universidades, fórum, etc, pôde-se constatar que ainda não existe um trabalho que mostra a eficiência de implementar um servidor de aplicação usando apenas MOR e RMI sem depender de *containers J2EE*. Na internet encontra-se uma vasta bibliografia, principalmente em inglês, sobre as duas tecnologias separadamente.

Outro detalhe observado durante o estudo, em relação ao mapeamento objeto-relacional, foi que trabalhar com MOR é um grande passo para iniciar o estudo com um Sistema Gerenciador de Banco de Dados Orientados a Objetos (SGBDOO).

Ficou claro também que, apesar da relutância de alguns em adotar esquemas de persistência, ficou evidente que sua utilização trás um ganho considerável de tempo na implementação de um sistema e eleva a qualidade do produto final, à medida que diminui a possibilidade de erros de codificação. O fraco acoplamento entre as camadas de dados e de lógica do sistema promovido pelas Camadas de Persistência é outro ponto que demonstra a sua utilidade. Além de fornecer um acesso mais natural aos dados, as Camadas de Persistência executam controle transacional, otimização de consultas e transformação automática de dados entre formatos distintos (tabelas relacionais para arquivos XML ou classes Java, por exemplo).

Sem dúvida, as Camadas de Persistência devem funcionar como a principal ponte de ligação entre sistemas Orientados a Objetos e repositórios de dados diversos: um conceito poderoso, com implementações estáveis e comprovadamente eficientes.

Um outro ganho considerável é a distribuição desses objetos persistentes, que incluem a possibilidade de seu crescimento incremental com mais facilidade e menor custo, e a possibilidade de implementação de tolerância a falhas. Lembrando também que ao invés de utilizar uma especificação J2EE, que implementa um servidor de aplicação que necessita de uma maquina com mais recursos, é muito mais simples fazer uma aplicação distribuída apenas em cima de RMI, deixando a aplicação muito mais leve.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

As desvantagens em usar MOR são muito poucas. Uma delas é ter que aprender um novo *framework* para poder iniciar o primeiro projeto. Outro detalhe é a performance que em alguns casos é inferior à utilização de instruções SQL diretamente pela API JDBC. Mais apesar disso, quando essas comparadas com as vantagens, esquecemos completamente as desvantagens. Além do que, usando MOR, o código será reduzido bastante ao deixar de lado todas as instruções SQL.

Em relação aos objetos distribuídos, a principal desvantagem no seu uso é a segurança. O acesso fácil aos dados também podem ser aplicados a dados secretos. Além disso, um outro ponto é que a rede pode saturar ou cometer erros fazendo com que ocorra erro na aplicação do cliente.

Espera-se que o trabalho contribua quanto às informações necessárias para a utilização das tecnologias citadas, bem como uma base de conhecimento sobre esta área que está crescendo a cada dia, e que promete ter um futuro promissor da tradicional tecnologia relacional e na distribuição dos dados.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

REFERÊNCIAS BIBLIOGRÁFICA

AMBLER, Scott W. **AmbySoft Home Page**. Toronto, 1997. Disponível em: <http://www.ambysoft.com/>. Acesso em: 15 out. 2005.

AMBLER, Scott W. **Mapping Objects to Relational Databases: O/R Mapping In Detail**. Toronto, 2005. Disponível em: <http://www.agiledata.org/essays/ mappingObjects.html>. Acesso em: 15 out. 2005.

AMBLER, Scott W. **The Design of a Robust Persistence Layer For Relational Databases**. Toronto, 2005. Disponível em: <http://www.ambysoft.com/downloads/ persistenceLayer.pdf>. Acesso em: 12 out. 2005.

AQUINO, Mario. **A Simple Data Access Layer using Hibernate**. [S.l., 2003. Disponível em: <http://www.ociweb.com/jnb/jnbNov2003.html>. Acesso em: 07 nov 2005.

BACLAWSKI, Kenneth. **Java RMI Tutorial**. Boston, 1998. Disponível em: <http:// www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html>. Acesso em: 18 out. 2005.

COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. **Distributed Systems**: *Concepts and Design*. [S.l], 2005.

CRAIG Larman's Home Page. [S.l], [200-]. Disponível em: <http://www.craiglarman. com/>. Acesso em: 13 out. 2005.

FORMAN, Ira R.; FORMAN, Nate. **Java Reflection in Action**. [S.l]: Manning, [19--]

GOLDMAN, Alfredo; KON, Fabio; REVERBEL, Francisco C.R. **Sistemas Distribuídos**. USP, São Paulo, 2002. Disponível em: <http://www.ime.usp.br/dcc/ areas/node12.html>. Acesso em: 22 out. 2005.

HIBERNATE - RELATIONAL PERSISTENCE FOR IDIOMATIC JAVA. Hibernate Reference Documentation. [S.l], 2002. Disponível em: <http://www.hibernate.org/ hib_docs/reference/en/html_single/>. Acesso em: 10 out. 2005.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531

IMASTERs. Java. [S.l], 2001. Disponível em: <http://www.imasters.com.br/secao.php?cs=28>. Acesso em: 15 out. 2005.

INTEGRAÇÃO: Spring e Hibernate. JavaFree, [S.l], 2005. Disponível em: <http://www.javafree.org/content/view.jf?idContent=46>. Acesso em: 18 out. 2005.

JERÔNIMO, Paulo. **Sistemas Distribuídos**. Brasília, 2004. Disponível em: <http://www.paulojeronimo.eti.br/download/2004/11/19/palestras-finom-pdf.zip>. Acesso em: 07 nov 2005.

JEVEAUX, Paulo César M. **Tutorial RMI - Invocação remota de métodos**. Portal Java, [S.l], 2002. Disponível em: <http://www.portaljava.com/home/modules.php?name=Content&pa=showpage&pid=8>. Acesso em: 12 out. 2005.

LARMAN, Craig. **Utilizando UML e Padrões**. *Uma Introdução à Análise e Projeto Orientados a Objetos*. Porto Alegre: Bookman, 2000.

LOZANO, Fernando. **Persistência Objeto-Relacional com Java**. [S.l], 2004. Disponível em: <http://www.lozano.eti.br/palestras/persistencia-oo.pdf>. Acesso em: 26 out. 2005.

MAPPING Objects to Relational Databases: O/R Mapping In Detail. [S.l], 2005. Disponível em: <http://www.agiledata.org/essays/mappingObjects.html>. Acesso em: 26 out. 2005.

FINN, Mary A. **Como Resolver a Impedância em Banco de Dados**. [S.l], 2003. Disponível em: <http://www.linhadecodigo.com.br/artigos.asp?id_ac=70&pag=1>. Acesso em: 12 out. 2005.

MUNDO OO. Curitiba, 2003. Disponível em: <http://www.mundooo.com.br/>. Acesso em: 15 out. 2005.

SISTEMAS Operacionais Distribuídos. [S.l], [19--]. Disponível em: <http://orbita.starmedia.com/~brodowski/sem2.htm>. Acesso em: 22 out. 2005.

Copyright © Leandro Sales – leandroasp@gmail.com – +55 86 94152531