UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

SERVIÇO WEB PARA ACESSO A DADOS NO *FRAMEWORK OBINJECT*

Wellington Openheimer Ribeiro

UNIFEI
Itajubá
Dezembro, 2014

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Wellington Openheimer Ribeiro

SERVIÇO WEB PARA ACESSO A DADOS NO *FRAMEWORK OBINJECT*

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciência e Tecnologia da Computação.

Área de Concentração: Sistemas de Computação

Orientador: Prof. Dr. Enzo Seraphim

UNIFEI
Itajubá
Dezembro, 2014

© 2014
Wellington Openheimer Ribeiro

Dados Internacionais de Catalogação na Publicação (CIP)

C331o

Ribeiro, Wellington Openheimer.
Serviço web para acesso a dados no *framework* *ObInject* / Wellington Openheimer Ribeiro. – Itajubá: UNIFEI, 2014.
xx + 54 f. : il. ; 29.7

Orientador: Enzo Seraphim.
Dissertação (Mestrado em Ciência e Tecnologia da Computação) – Universidade Federal de Itajubá, Itajubá, 2014.
1. Persistência. 2. Indexação. 3. Framework. 4. Serviços WEB. 5. CRUD 6. Serviço I. Seraphim, Enzo, orient. II. Universidade Federal de Itajubá. III. Título.

CDU 004.65

Exemplar correspondente à versão final do texto, após a realização da defesa, incluindo, portanto, as devidas considerações dos examinadores.

Typeset in pdfLaTeX

Folha de Aprovação

Dissertação **aprovada** pela Banca Examinadora em 12 de dezembro de 2014, conferindo ao autor o título de **Mestre em Ciência e Tecnologia da Computação**.

Dr. Enzo Seraphim ______________________________

Orientador

Universidade Federal de Itajubá

Dr^{a}. Melise Maria Veiga de Paula ______________________________

Membro da Banca

Universidade Federal de Itajubá

Dr. Humberto Luiz Razente ______________________________

Convidado

Universidade Federal de Uberlândia

O pensamento positivo pode vir naturalmente para alguns,
mas também pode ser aprendido e cultivado,
mude seus pensamentos e você mudará seu mundo.

Norman Vincent Peale

A minha querida esposa Andressa.

Agradecimentos

Agradeço a Deus. Refúgio que permite-me a calma, tranquilidade e confiança.

A minha querida esposa, *Andressa Gabrielle Pereira Openheimer*. Por sua amável companhia em todos os momentos desta etapa de minha vida, sempre me motivando e reforçando minhas qualidades. Nos momentos de alegria e realizações, mas principalmente nos momentos de derrota e cansaço. Por sua paciência em superar minhas ausências e contratempos por conta dos trabalhos. Esta conquista é dedicada a você.

A meus pais, *Silvio Torquato Ribeiro* e *Abigail Openheimer Ribeiro*, e minha irmã *Deise Maria Openheimer Ribeiro*. Por serem os meus maiores exemplos de dignidade, respeito, amor, cuidado e dedicação. Credito em especial esta conquista a meu pai, que apesar de sua humildade é um grande exemplo de homem trabalhador, inteligente e determinado. Sempre procuro me inspirar em sua experiência e sabedoria.

A meu professor, orientador e grande amigo, *Prof. Dr. Enzo Seraphim*. Ao passar do tempo aprendi a respeitar e admirar tamanho profissionalismo, maturidade, experiência, conhecimento e além de tudo sua paixão por ensinar e formar grandes profissionais e pesquisadores da computação. Qualidades estas que refletem no reconhecimento de nossa instituição e da comunidade acadêmica. Além do empenho profissional e pessoal na orientação deste trabalho, agradeço em especial por todas as vezes que precisei de um conselho, seja ele profissional ou pessoal, de uma palavra amiga, onde sempre me encorajou e valorizou minhas habilidades. Professor, o senhor é um exemplo para mim, de profissional e pai de família. Espero que possamos trabalhar juntos em outras oportunidades.

A todos os docentes do programa de pós-graduação em Ciência e Tecnologia da Computação por compartilharem seus conhecimentos e experiências acadêmicos e profissionais entre nós alunos e contribuir com nossa formação.

À Universidade Federal de Itajubá e a Diretoria de Suporte à Informática pelo incentivo à qualificação de seus servidores e pelo apoio de todos os colegas de departamento.

Ao Instituto Federal de Educação, Ciência e Tecnologia do Sul de Minas por todos os programas de incentivos à qualificação de seus servidores, contribuindo para o desenvolvimento do servidor como profissional e cidadão.

Resumo

A tecnologia de serviço web tem se mostrado uma importante ferramenta para integração de dados entre plataformas heterogêneas devida a disponibilidade de interfaces padronizadas e comunicação através da web. Esta integração de dados envolve armazenamento e recuperação de informação em sistemas gerenciadores de bancos de dados (SGBD). Os SGBD's possuem um conjunto de interfaces padronizadas, como por exemplo o ODBC *(Open Database Connectivity)*, para acesso a dados sem a necessidade de que aplicações clientes codifiquem métodos de acesso especializados para tratar aspectos de persistência. Recentemente, vários *frameworks* para persistência de dados têm sido propostos, dentre eles, o *framework ObInject* (CARVALHO et al., 2013) que apresenta uma abordagem de indexação de objetos. No entanto, sua utilização está limitada em armazenar persistência em um sistema de arquivos local e somente para modelos de dados desenvolvidos em Java. Este trabalho desenvolve uma proposta baseada em serviços web para criação de um provedor de serviços de armazenamento e recuperação de dados para o *framework ObInject*. O provedor de serviços disponibiliza serviços CRUD *(Create, Retrieve, Update, Delete)* para acesso a dados de objetos de aplicações clientes. A partir da interoperabilidade que os serviços web oferecem, potenciais aplicações clientes escritas em qualquer linguagem de programação que desejem persistir dados, podem se tornar consumidoras do serviço de forma padronizada.

Palavras-chave: Persistência, Indexação, *Framework*, CRUD, Serviço Web.

Abstract

Webservice data access in framework ObInject

The web service technology has been an important tool for data integration between heterogeneous platforms due to the availability of standardized interfaces and communication on the web. This data integration involves storage and information retrieval in database management systems (DBMS). DBMSs have a set of standard interfaces such as ODBC (Open Database Connectivity), that access data without requiring that client applications encode specialized access methods to address aspects of persistence. Recently, several frameworks for data persistence have been proposed, among them the ObInject framework that presents an approach for indexing objects. However, its use is limited to persist in a local system and only for data models developed in Java. This paper shows a web service based proposal to create a service provider of storage and data retrieval for ObInject framework. The service provider offers CRUD (Create, Retrieve, Update, Delete) services for data access of objects from client applications. Based on the interoperability that Web services offers, potential customers applications written in any programming language who want to persist data, can become service consumers in a standard way.

Keywords: *Persistence, Indexing, Framework, CRUD, Web Services.*

Sumário

Lista de Figuras **xviii**

Abreviaturas e Siglas **xix**

1 Introdução **1**

1.1 Objetivo 3

1.2 Organização do trabalho 3

2 Revisão Bibliográfica **5**

2.1 Arquitetura Orientada a Serviços 5

2.1.1 Serviços 6

2.1.2 Serviços Web 9

2.1.2.1 *Simple Object Access Protocol* (SOAP) 10

2.1.2.2 *Web Services Description Language* (WSDL) 11

2.1.2.3 *Universal Description, Discovery and Integration* (UDDI) . 11

2.1.3 *Java API for XML Web Services* (JAX-WS) 12

2.2 *Framework ObInject* 13

2.2.1 Módulos 14

2.2.2 Metaprogramação para geração de classes 16

2.3 Trabalhos Correlacionados 19

3 Serviço web para acesso a dados **23**

3.1 Arquitetura 24

3.1.1 Provedor de serviço de acesso a dados 24

3.1.2 Consumidor de Serviço 30

3.2 Fluxo de Funcionamento 30

3.3 Considerações Finais 32

4 Experimentos **33**

4.1 Objetivo 33

4.2 Caso de Uso: Piratas do Caribe 34

4.2.1 Análise Funcional: Aplicação consumidora em Java 35

4.2.2 Análise Funcional: Aplicação consumidora em C++ 38

4.2.3 Análise Funcional: Aplicação consumidora em PHP 39

4.3 Análise de Desempenho 40

4.4 Resultados 42

5 Conclusão **43**

Referências Bibliográficas **45**

A Exemplo de log no provedor de serviços **49**

B Código C++ para registro da aplicação “Piratas” **51**

C Código PHP para registro da aplicação “Piratas” **53**

Lista de Figuras

2.1 Arquitetura Orientada a Serviços 6

2.2 Pilha conceitual de serviços web 9

2.3 Interação SOAP, WSDL e UDDI em SOA 10

2.4 Ilustração dos pacotes do framework Object-Inject 14

2.5 Framework Object-Inject: pacote Meta 15

2.6 Diagrama de classes para geração de classes 17

3.1 Arquitetura do serviço web para acesso a dados 24

3.2 Diagrama de Classes DaoService 25

3.3 Caso de Uso de uma Escola 26

3.4 Classe Pessoa(a) e classe Aluno(b) geradas pelo método *create* 27

3.5 Classe de serviços “Escola” gerada pelo método *commit* 29

3.6 Classe de publicação dos serviços web “Escola” gerada pelo método *commit* . 30

3.7 Fluxo de Funcionamento 31

4.1 Diagrama de Classes - Caso de Uso “Piratas do Caribe” 35

4.2 Código Java para registro de aplicação “Piratas do Caribe” 36

4.3 Diagrama de classes pacote “Piratas” representando a entidade Costa 37

4.4 Armazenamento(a) e recuperação(b) de um objeto da entidade “Costa” em Java 38

4.5 Armazenamento(a) e recuperação(b) de um objeto da entidade “Costa” em C++ 39

4.6 Armazenamento(a) e recuperação(b) de um objeto da entidade “Costa” em PHP 39

4.7 Gráfico de tempo de processamento para inserção de objetos 41

4.8 Gráfico de consumo de espaço em disco para inserção de objetos 42

A.1 Log de saída para registro da aplicação “Piratas do caribe” 49

B.1 Código C++ para registro de aplicação “Piratas do Caribe” 51

C.1 Código PHP para registro de aplicação “Piratas do Caribe” 53

Abreviaturas e Siglas

API	–	Application Programming Interface
CORBA	–	Common Object Request Broker Architecture
CRTP	–	Curiously Recurring Template Pattern
CRUD	–	Create, Retrieve(Read), Update, Delete
DAO	–	Data Access Object
DCOM	–	Distributed Component Object Model
FTP	–	File Transfer Protocol
GIS	–	Geographic Information System
HTTP	–	Hipertext Transfer Protocol
JAX-WS	–	Java API for XML Web Services
JSON	–	JavaScript Object Notation
LOD	–	Linked Open Data
ODBC	–	Open Database Connectivity
OGSA	–	Open Grid Services Architecture
ORM	–	Object-Relational Mapping
PHP	–	Hypertext Preprocessor
RDF	–	Resource Description Framework
REST	–	Representational State Transfer
RPC	–	Remote Procedure Call
SGBD	–	Sistema Gerenciador de Bancos de Dados
SMTP	–	Simple Mail Transfer Protocol
SOA	–	Service-Oriented Architecture
SOAP	–	Simple Object Access Protocol
SQL	–	Structured Query Language
TI	–	Tecnologia da Informação
UDDI	–	Universal Description, Discovery and Integration
UML	–	Unified Modeling Language
URI	–	Uniform Resource Identifier
URL	–	Uniform Resource Locator
UUID	–	Universally Unique IDentifier
W3C	–	World Wide Web Consortium
WFS	–	Web Feature Service

WS-DAI – Web Services Data Access and Integration
WSDL – Web Services Description Language
XML – Extensible Markup Language

Capítulo 1

Introdução

A Internet tem se mostrado uma ferramenta imprescindível na vida das pessoas. Atualmente a web é amplamente utilizada nas mais diversas finalidades como entretenimento, negócios, trabalho, informação, dentre outras. Um único tipo de aplicação, o navegador web, é suficiente para ter acesso a uma infinidade de serviços disponibilizados na internet. Por outro lado, a comunicação entre aplicações heterogêneas apresentava desafios, tais como, a possibilidade de integração de dados e o reaproveitamento de funcionalidades. Para superar estes desafios, surgiram tecnologias padronizadas para disponibilização de funcionalidades de softwares como serviços pela web, que possibilitaram comunicação entre aplicações.

Assim, surgiram os serviços web que tornaram possível que aplicações distintas, executando em diferentes plataformas pudessem estabelecer comunicação e sincronização através da web de uma forma padronizada e normalizada. A adoção de serviços web permite que diversas aplicações contidas em um mesmo processo de negócio possam se interagir e permitir o fluxo dos dados de forma dinâmica, onde as aplicações podem tornar público um serviço e qualquer aplicação pode também consumir este serviço.

A abordagem de software como serviço é uma tendência que se consolida a cada dia (TSAI et al., 2014). A transformação de simples programas de computador em serviços de tecnologia da informação elevou a computação a níveis estratégicos dentro das organizações, capazes de interferir nos negócios e no sucesso das empresas. Estes programas de computador disponibilizados como serviços podem ser acessados de qualquer lugar onde a aplicação cliente utiliza software sob sua demanda.

Uma vantagem dos serviços web é permitir flexibilidade para integração de dados de aplicações (HANSEN et al., 2003) que normalmente estão armazenados em bancos de dados.

O conceito de banco de dados foi concebido e aprimorado desde a década de 60 (CODD, 1970). A evolução do armazenamento e recuperação de dados contempla desde a simples operação de leitura e escrita em arquivos de campos de tamanho fixo até os mais modernos

bancos de dados relacionais que provêm manipulação de grande volume de transações com altos índices de escalabilidade e confiabilidade.

A manipulação de dados em bancos de dados relacionais pode ser feita com a linguagem de consulta SQL (*Structured Query Language*) (CHAMBERLIN; BOYCE, 1974). No entanto, a linguagem não possibilitava a comunicação entre aplicações distintas, sendo necessário padronizações tais como ODBC (*Open Database Connectivity*) e o DAO *(Data Access Object).*

O ODBC (SIGNORE et al., 1995) foi desenvolvido para atuar como uma camada *middleware* entre a aplicação e o sistema de banco de dados, de forma que a aplicação e seu ambiente de execução se tornam independentes do banco de dados e do sistema operacional através de uma Interface de Programação de Aplicações (Application Programming Interface - API) comum. Esta API é implementada pelos diversos fabricantes de bancos de dados através do driver ODBC.

Uma outra padronização é o DAO (MICROSYSTEMS, 2001) que isola a aplicação dos detalhes das soluções de bancos de dados. Dentre estes detalhes, as consultas de SQL para manipulação de dados, bem como as funções específicas da linguagem de programação da aplicação para conexão com o banco de dados. O DAO fornece quatro operações básicas de manipulação de dados conhecidas como CRUD *(Create, Retrieve(Read), Update, Delete)* (MARTIN, 1981). A operação *Create* é responsável por persistir dados, a operação *Retrieve(Read)* é responsável por consultar e recuperar dados persistidos, a operação *Update* é responsável por atualizar dados persistidos e a operação *Delete* é responsável por excluir dados persistidos.

Consequentemente, as operações CRUD envolvem os resultados esperados por clientes que são representados por objetos que concentram o modelo de negócio. Dentro dos modelos de negócio, a informação se apresenta como fator de extrema relevância capaz de agregar valor (O'REILLY, 2007).

Estas padronizações não tratam o problema de diferentes paradigmas utilizados no armazenamento de dados e no desenvolvimento de aplicações. O armazenamento de dados tradicional utiliza o paradigma relacional enquanto que aplicações modernas são desenvolvidas utilizando o paradigma orientado a objeto.

Uma solução que vem se popularizando são os chamados *frameworks* de mapeamento objeto-relacional (*Object-relational mapping* - ORM) (BARRY; STANIENDA, 1998). A técnica de ORM mapeia os objetos em relações dos bancos de dados, bem como as consultas SQL. No entanto, os *frameworks* ORM apresentam uma camada adicional de processamento para traduzir objetos em relações de banco de dados e vice-versa.

Outra solução é adotar uma forma de armazenamento que utiliza o paradigma orientado a objeto, como o *framework ObInject* (CARVALHO et al., 2013). Esta solução não contém uma camada extra de tradução de objetos para relações em bancos de dados pois gerencia os objetos em suas estruturas de armazenamento. Com a utilização deste *framework*, qualquer

objeto definido em linguagem de programação pode ser indexado por estruturas de dados que podem estar armazenadas em diferentes dispositivos. No entanto, sua utilização está limitada em armazenar persistência e indexações em um sistema de arquivos local para modelos de dados desenvolvidos na linguagem de programação Java.

1.1 Objetivo

O objetivo deste trabalho é desenvolver um provedor de serviços de armazenamento e recuperação de dados para o *framework ObInject* através de serviços web. O provedor de serviços deve disponibilizar serviços CRUD *(Create, Retrieve, Update, Delete)* para acesso a dados de objetos de aplicações clientes. Ao final deste trabalho será possível criar um servidor de armazenamento e recuperação de dados e disponibilizar este serviço. A partir da interoperabilidade que os serviços web oferecem, potenciais aplicações clientes escritas em qualquer linguagem de programação que desejem persistir dados, poderão se tornar clientes do serviço de forma padronizada.

Espera-se que com um serviço web para operações CRUD, as aplicações tendem a se concentrar no modelo de negócio do cliente e não em detalhes da tecnologia de armazenamento e recuperação de dados.

Finalmente, este trabalho visa prover acesso remoto ao *framework* utilizando o conceito de serviços web que estabelece uma prestação de serviço à aplicação cliente. Dessa maneira, o serviço web encapsula os detalhes da implementação de armazenamento do *framework* e prevê o consumo do serviço sob demanda da aplicação cliente.

1.2 Organização do trabalho

Este documento está organizado da seguinte maneira:

- *Capítulo 2 - Revisão bibliográfica*: discute os principais conceitos utilizados neste trabalho.
- *Capítulo 3 - Serviço web para acesso a dados*: descreve o trabalho desenvolvido, mostrando seu funcionamento e detalhes de implementação.
- *Capítulo 4 - Experimentos*: apresenta os experimentos realizados com o serviço web desenvolvido, discutindo sua metodologia e resultados.
- *Capítulo 5 - Conclusão*: apresenta as conclusões do trabalho e propostas de metas futuras.

Capítulo 2

Revisão Bibliográfica

Neste capítulo serão apresentados tópicos sobre tecnologias diretamente relacionadas com o desenvolvimento do serviço web para acesso a dados. Na seção 2.1 são apresentados tópicos relacionados a arquitetura orientada a serviços como serviços web, tecnologias SOAP, WSDL, UDDI e a biblioteca JAX-WS. Na seção seção 2.2 é apresentado o *framework ObInject* (CARVALHO et al., 2013) . Finalmente, na seção 2.3 são apresentados os trabalhos correlacionados.

2.1 Arquitetura Orientada a Serviços

A arquitetura de software orientada a serviços define que uma aplicação pode disponibilizar funcionalidades em forma de serviços. Segundo Josuttis (2007) SOA é um paradigma para a compreensão e manutenção de processos de negócio que abrangem sistemas grandes.

Para Josuttis (2007), SOA é baseado em três conceitos principais: serviço, interoperabilidade e baixo acoplamento. O serviço é uma representação de TI de alguma funcionalidade de negócio, seja esta funcionalidade simples ou complexa. Serviços desempenham o papel de estruturar sistemas distribuídos baseados em abstrações de regras e processos de negócios. A interoperabilidade é a capacidade de conectar sistemas heterogêneos de forma fácil. Para isso é necessário definições formais que independam da plataforma e linguagem de programação, criando uma interface que esconda a implementação, tornando assim os sistemas SOA independente da tecnologia de desenvolvimento ou plataforma. O baixo acoplamento é a minimização de dependências entre sistemas. Quando as dependências são minimizadas, modificações têm efeitos minimizados permitindo que sistemas continuem rodando quando parte dele falha. Isto proporciona flexibilidade, escalabilidade e tolerância a falhas.

Pulier (2006) adiciona um quarto conceito à definição de SOA, chamado de aspecto negocial. Neste conceito, a interoperabilidade entre sistemas heterogêneos permite flexibilidade e dinamismo no processamento das regras de negócio.

Para atingir estes conceitos, é definida uma arquitetura baseada em 3 entidades, como pode ser vista na figura 2.1: o provedor do serviço(servidor) que fornece as interfaces de serviços; o consumidor de serviços(cliente) que descobre e invoca serviços e o serviço de registro que atua como um repositório ou catálogo de serviços publicados.

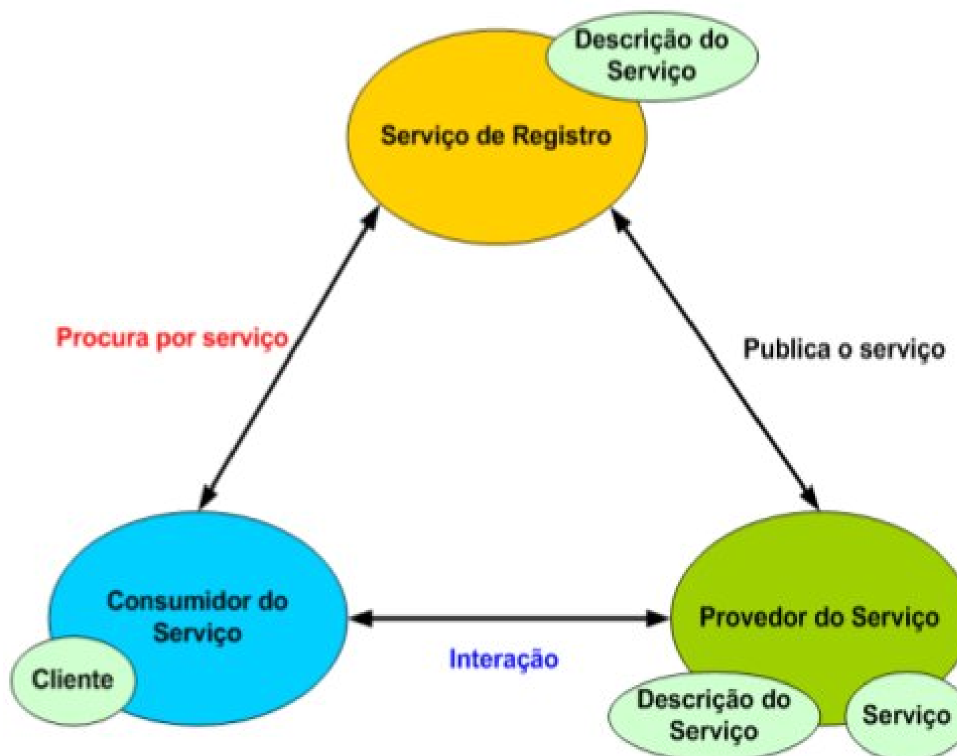

Figura 2.1: Arquitetura Orientada a Serviços
Fonte: (KREGER et al., 2001)

2.1.1 Serviços

Um serviço é uma funcionalidade de negócio auto-contida que não mantém estado, que aceita uma ou mais requisições e retorna uma ou mais respostas através de uma interface bem definida e padronizada (POMBINHO et al., 2012). Baseado nesta definição, Josuttis (2007) afirma que o objetivo primário de um serviço é o de representar naturalmente uma parte de uma funcionalidade de negócio, de acordo com o domínio cujo o qual está inserido.

Erl (2005) descreve um conjunto das principais características associadas a serviços, listadas a seguir:

- **Reusabilidade**: A orientação a serviços alavanca o reuso em todos os serviços, mesmo se já houverem requisitos para tal. Aplicando padrões de projeto que tornam cada projeto potencialmente reutilizável, aumentam-se as chances de acomodar futuros requisi-

tos com menos esforço de desenvolvimento. Além disso, serviços reutilizáveis reduzem a necessidade da criação de serviços que encapsulam outro serviços por conveniência.

- **Obediência a contratos**: Para a interação de serviços, eles precisam apenas de compartilhar e seguir um contrato formal que descreva cada serviço e defina as questões de troca de informação. Contratos de serviços fornecem uma definição formal para o destino final do serviço, para cada operação do serviço, para toda mensagem de entrada/saída e para regras/características do serviço e suas operações. Bons contratos de serviço podem também prover informações semânticas que explicam como um serviço pode se comportar quando for cumprir uma determinada tarefa.

- **Baixo acoplamento**: Serviços devem ser projetados para interagir sem a necessidade de dependências fortes entre serviços. Baixo acoplamento é uma condição onde um serviço adquire conhecimento de outro serviço enquanto permanece independente do mesmo. Baixo acoplamento é alcançado pelo uso de contratos de serviço que permitem a interação entre serviços por parâmetros pré-definidos.

- **Abstração de lógica de negócio**: A única parte de um serviço que é visível para o mundo exterior é o que está exposto via contrato de serviço. Lógica de negócio, além do que estiver descrito nas descrições do contrato, é invisível e irrelevante aos consumidores do serviço. Desta forma, serviços podem ser concebidos como caixas pretas, escondendo seus detalhes do mundo exterior.

- **Composição**: Serviços podem compor outros serviços. Isto permite à logica ser representada em níveis diferentes de granularidade e promove reusabilidade e a criação de camadas de abstração.

  Uma extensão SOA comum que envolve composição é o conceito de orquestração, onde um processo orientado a serviços (que pode ser uma composição) é controlado por um processo pai que compõe processos participantes.

- **Autonomia**: A lógica governada por um serviço é circundada por limites explícitos. O serviço possui controle dentro destes limites e não depende de outros serviços para executar sua governança. Isto elimina dependências em outros serviços, o que libera um serviço de laços que poderiam inibir sua implantação e evolução. Autonomia de serviços é uma consideração primária quando se decide como a lógica de aplicação deve ser dividida em serviços e quais operações devem ser agrupadas em um contexto de serviço.

- **Sem armazenar estado**: Serviços não devem ser requisitados para gerenciar informações de estado, pois isto pode impedir sua habilidade de permanecer com baixo acoplamento. Serviços devem ser projetados para maximizar a capacidade do não-armazenamento de estado, mesmo que isto signifique a delegação do gerenciamento do estado para outro lugar.

A condição do não-armazenamento de estado é fundamental e promove reusabilidade, escalabilidade e idempotência. Para que um serviço retenha o mínimo estado possível, suas operações individuais devem ser projetadas com considerações de processamento para o não-armazenamento de estado.

- **Capacidade de ser descoberto**: Serviços devem permitir às suas descrições serem descobertas e entendidas por humanos e requisitantes de serviços que possam utilizar sua lógica. Esta capacidade ajuda a evitar a criação acidental de serviços redundantes ou serviços que implementam lógica redundante. Devido a cada operação prover um pedaço de lógica de processamento com potencial reusabilidade, metadados anexados a um serviço devem descrever suficientemente as funcionalidades oferecidas por suas operações, além de seu propósito geral.

Josuttis (2007) descreve uma comum classificação técnica de serviços que permite a introdução de camadas SOA e seus estágios de expansão. Esta classificação baseia-se em três categorias: serviços básicos, serviços compostos e serviços processo.

O primeiro estágio de expansão compreende apenas serviços básicos, que provêem funcionalidades de negócio básicas. Há dois tipos de serviços básicos: serviços básicos envolvendo dados e serviços básicos envolvendo lógica.

Serviços básicos envolvendo dados realizam leitura ou escrita de dados de ou para um sistema. Estes serviços tipicamente representam uma operação de negócio fundamental, de forma a encapsular aspectos específicos a plataformas e detalhes de implementação do mundo exterior. Tais serviços seguem necessidades de negócio.

Serviços básicos envolvendo lógica representam regras de negócio fundamentais. Estes processos usualmente processam alguma entrada e retornam resultados correspondentes. Exemplo típico: Retornar se um ano é bissexto.

O segundo estágio de expansão acrescenta serviços compostos. Estes representam a primeira categoria de serviços que são compostos por outros serviços (básicos e/ou outros serviços compostos). Na terminologia SOA, compor novos serviços com serviços existentes é chamado orquestração. Estes serviços operam em um nível mais alto que serviços básicos, mas ainda caracterizam-se por execução curta e conceitualmente não armazenam estado.

O terceiro estágio de expansão acrescenta serviços processo, que representam fluxos de trabalho mais longos ou processos de negócio. Diferentemente de serviços básicos e compostos, um serviço processo usualmente mantêm estado que permanece estável através de diversas chamadas.

2.1.2 Serviços Web

A definição do W3C (W3C, 2004) para um serviço web é: um sistema de software projetado para suportar interoperabilidade através de uma rede. Possui uma interface descrita em formato processável por máquina(WSDL). Outros sistemas interagem com o serviço web de forma prescrita por sua descrição utilizando mensagens SOAP, normalmente transmitidas usando HTTP com XML serializado em conjunto com outros padrões web. Os serviços web permitem que desenvolvedores de aplicações escolham e utilizem, de maneira conveniente, funções existentes na web (GURUGE, 2004).

Em (KREGER et al., 2001) é apresentada a pilha conceitual de serviços web que utiliza as tecnologias:

- *Simple Object Access Protocol* (SOAP), que descreve o formato da mensagem;
- *Web Services Description Language* (WSDL), que é um documento que descreve um serviço e como invocá-lo;
- *Universal Discovery, Description, and Integration* (UDDI), que é um diretório do serviço que está disponível para ser usado.

A pilha conceitual pode ser vista na figura 2.2:

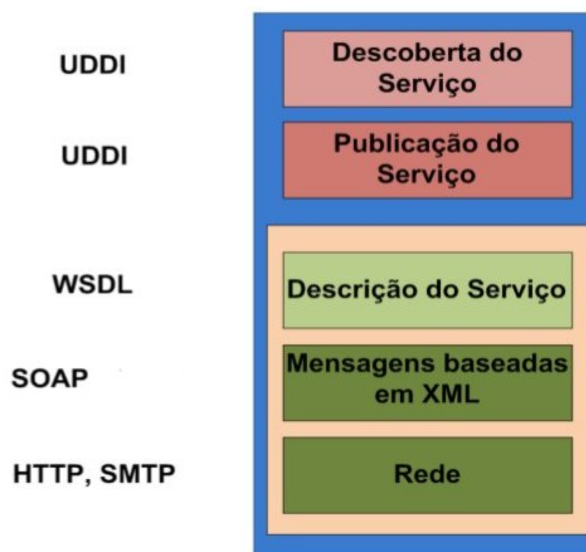

Figura 2.2: Pilha conceitual de serviços web
Fonte: (KREGER et al., 2001)

A base da pilha é a rede, camada acessível pelos clientes através do protocolo HTTP. A camada de mensagens representa o uso de XML como base para o protocolo de mensagem SOAP. A camada de descrição de serviços apresenta a WSDL onde são definidos a interface

e os mecanismos de interação entre os serviços. A camada de publicação e descoberta do serviço pode ser realizada por meio de um repositório de descrição de serviços como o UDDI. Esta camada inclui a descrição do serviço e sua publicação.

Finalmente, estas três tecnologias, SOAP, WSDL e UDDI, envolvidas na construção de um serviço web, interagem na arquitetura orientada a serviço como pode ser visto na figura 2.3.

Figura 2.3: Interação SOAP, WSDL e UDDI em SOA
Fonte: (KREGER et al., 2001)

2.1.2.1 *Simple Object Access Protocol* (SOAP)

A W3Schools (2006) define SOAP como um protocolo simples baseado em XML que permite aplicações trocarem informações através do protocolo HTTP.

De acordo com Josuttis (2007), SOAP foi o primeiro padrão real de serviço web que foi desenvolvido. Originalmente SOAP era acrônimo de "Simple Object Access Protocol". Entretanto, notou-se que SOAP não era simples e que não se tratava de acesso a objetos.

Pulier (2006) afirma que SOAP é a linguagem mais usada em serviços web, a estrutura XML na qual todas a mensagens dos serviços web são construídas. Quando se diz que serviços web são baseados em XML, na verdade se quer dizer que serviços web são baseados em mensagens SOAP, que são escritas em XML. O que torna SOAP especial e o difere do simples XML é que cada mensagem SOAP segue um padrão que foi especificado pelas normas da W3C.

SOAP pode ser referido como um "envelope de dados". Cada mensagem SOAP começa com uma tag que lê `<SOAP-ENV:envelope>`. A tag envelope sinaliza a mensagem ao destinatário que ele está prestes a receber uma mensagem SOAP. O que se segue é um cabeçalho, que contém a informação crítica sobre origem e destino das mensagens. E depois, há o corpo da

mensagem SOAP, que define o dado atual ou instruções de operação exigidos pelo computador consumidor.

Uma mensagem SOAP é formatada como um "envelope" de código XML que define o seu início e fim. O "cabeçalho" descreve de onde a mensagem veio, para onde ela está indo, e como ela está indo. O "corpo" da mensagem SOAP contém os dados pertinentes ou instruções processuais do pedido de resposta SOAP.

Guruge (2004) ainda afirma que SOAP é utilizado para enviar entradas, e receber saídas de um serviço web em XML convencional. Já que um serviço web requer parâmetros de entrada para ser ativado, SOAP também é considerado o agente que invoca um serviço web.

SOAP também é uma ferramenta para realizar chamadas remotas de procedimentos. Possui a vantagem de ser baseado em XML e ser transportado por HTTP e não precisar de portas específicas abertas para invocar os métodos, como ferramentas como CORBA - Commom Object Request Broker Architecture e DCOM - Distributed Component Object Model.

2.1.2.2 *Web Services Description Language* (WSDL)

Segundo Pulier (2006), WSDL é um documento XML, feito de acordo com os padrões especificados pela W3C, que descreve exatamente como um serviço web específico trabalha. O documento WSDL descreve para o potencial consumidor do serviço uma explicação de como o serviço funciona e como acessá-lo. O WSDL descreve como criar uma solicitação SOAP que irá invocar aquele serviço específico.

Caso um desenvolvedor pretenda criar uma aplicação cliente de um serviço web, ele necessita deste documento descritivo. A WSDL fornece todas as informações que o desenvolvedor necessita para criar um programa que requisite um serviço web. Isto significa que os desenvolvedores podem criar aplicações consumidoras sem nunca terem conhecido ou falado com o desenvolvedor que criou o serviço web em questão. Por causa das capacidades da WSDL, os serviços web são vistos como elementos de software auto-descritivos.

2.1.2.3 *Universal Description, Discovery and Integration* (UDDI)

UDDI é um protocolo de serviços de diretório que contém as descrições dos serviços web. Opera como um registro de serviços. O UDDI possibilita potenciais clientes a localizarem serviços e descobrirem seus detalhes.

Pulier (2006) exemplifica o conceito dizendo que o UDDI pode ser comparado a uma lista telefônica, onde páginas brancas contêm endereço, contatos do fornecedor; as páginas amarelas contêm listagens categorizadas nos tipos de negócios e as páginas verdes indicam serviços oferecidos por cada negócio. O UDDI funciona como um catálogo de serviços web. Caso o

desenvolvedor, necessite de um serviço específico, o UDDI pode informar onde encontrar este serviço e direcionar ao documento WSDL do serviço.

2.1.3 *Java API for XML Web Services* (JAX-WS)

JAX-WS (KOHLERT; GUPTA, 2007) é uma API Java para criação de serviços web. Faz parte da plataforma Java *Enterprise Edition*. A especificação JAX-WS define um padrão de mapeamento entre código Java e WSDL. Métodos descritos em Java são mapeados em operações descritas em WSDL. Uma mensagem SOAP ao invocar uma operação WSDL, é direcionada para processamento por um método Java mapeado.

De acordo com a especificação da biblioteca JAX-WS, um serviço web pode ser publicado através de anotações da biblioteca inseridas em uma determinada classe e seus métodos. A anotação *@webservice* de JAX-WS determina que uma classe Java contém serviços web. A anotação *@webmethod* determina que um método de uma classe deve ser publicado na WSDL do serviço web. Esta metodologia facilita o desenvolvimento e publicação de serviços web em Java.

Para publicar os serviços, é necessário uma ação de *EndPoint*, em tradução literal, uma extremidade ou interface que publica por meio da linguagem WSDL o serviço na web em um endereço HTTP. Na biblioteca JAX-WS, essa ação é determinada pela execução do método estático *publish* da classe *EndPoint*.

Neste método *publish* são passados dois parâmetros: o objeto da classe anotada com *@webservice* e a URI onde estará publicado o serviço.

A partir desse momento, qualquer aplicação cliente pode consumir o serviço através do protocolo SOAP que transmite mensagens sobre o protocolo HTTP.

2.2 *Framework ObInject*

ObInject (CARVALHO et al., 2013) é um *framework* de indexação e persistência de objetos desenvolvido na linguagem de programação Java.

Este *framework* utiliza índices primários para realizar persistência e índices secundários para realizar indexação.

Índices primários são índices que determinam a localização de um registro em um arquivo de dados. Eles atuam sobre conjuntos de atributos que contém chaves-primárias que são definidas por um UUID.

Um Identificador Único Universal (*Universally Unique Identifier* - UUID) é uma sequência de 128 *bits*, sendo que cada *byte* pode ser codificado como um número inteiro. Esta sequência é representada por um conjunto de 32 dígitos hexadecimais divididos em cinco grupos separados por hífens, como 01234567-89AB-CDEF-0123-456789ABCDEF (ZAHN et al., 1990), (LEACH et al., 2005), (ISO/IEC, 1996), (ITU, 2004).

Para garantir a unicidade destes identificadores é utilizado o algoritmo de geração de números aleatórios proposto por (ITU, 2004). Neste algoritmo, o esgotamento de possibilidades, considerando uma taxa de geração de UUIDs de 100 trilhões por nanosegundo, ocorreria em aproximadamente 3,1 trilhões de anos.

Os identificadores UUIDs definem as chaves primárias que são utilizadas pelos índices primários para armazenamento e recuperação de dados. São disponibilizadas implementações de índices primários baseados nas estruturas de dados hash extensível e lista duplamente encadeada.

Índices secundários são estruturas empregadas para otimizar a busca de um índice primário. Eles são definidos por domínios de indexação e conjuntos de atributos-chave. Estes atributos e o UUID são replicados em chaves que são empregadas em uma estrutura de indexação apropriada para o domínio. Estruturas de indexação baseadas em árvores são disponibilizadas, com armazenamento de chaves de roteamento em nós internos e chaves em nós folha. Assim, a partir de um valor de atributo-chave, pode-se navegar entre chaves de roteamento para alcançar uma chave com um UUID. Este identificador é então utilizado para a recuperação do objeto em um índice primário (RAMAKRISHNAN; GEHRKE, 1999), (GARCIA-MOLINA et al., 2008). As implementações de estruturas de indexação disponíveis são Árvore B+ (domínio ordenável) (COMER, 1979), Árvore M (domínio métrico) (CIACCIA et al., 1997), Árvore R (GUTTMAN, 1984) (domínio espacial).

A seguir, serão detalhados os módulos utilizados para realizar a persistência e indexação do framework.

2.2.1 Módulos

O *framework* é organizado em 4 módulos: Dispositivos, Blocos, Armazenamento e Meta (figura 2.4).

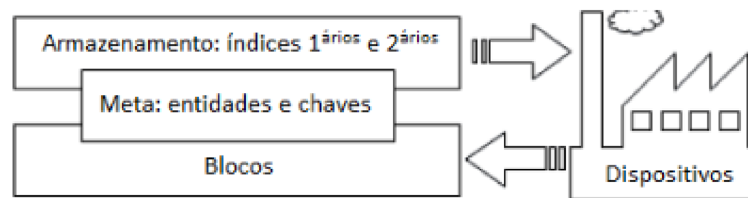

Figura 2.4: Ilustração dos pacotes do framework Object-Inject
Fonte: (CARVALHO et al., 2013)

O módulo Dispositivos define o gerenciamento de recursos de hardware para unidades de armazenamento. Esta é uma camada de abstração sobre meios de armazenamento, responsável por gerenciar alocação, manipulação e liberação de recursos, dispondo unidades de armazenamento para estruturas do módulo Armazenamento.

O módulo Blocos define unidades de armazenamento para estruturas do módulo Armazenamento, possibilitando a divisão dos dados armazenados. Estas unidades são independentes de meio de armazenamento e são disponibilizadas pelo módulo Dispositivos.

O módulo Armazenamento define estruturas de dados para índices primários e secundários, responsáveis por gerenciar entidades e chaves sobre unidades de armazenamento do módulo Blocos.

O módulo Meta define interfaces de entidades (objetos persistentes), chaves (objetos indexáveis contendo atributos-chave de um objeto persistente), domínios de indexação e classes ajudantes de (des)serialização.

Por meio de entidades e chaves(associadas a domínios de indexação), o módulo Meta define o vínculo entre aplicação e *framework*. A garantia de transparência e retrocompatibilidade para classes de uma aplicação é realizada com adoção de funcionalidades especificadas em interfaces. Através de classes empacotadoras dedicadas ao vínculo com o framework, é possível eximir classes existentes da aplicação de modificações. As interfaces *Entity* ou *Key* podem ser implementadas por uma classe empacotadora de persistência ou indexação especializando uma classe da aplicação. Desta forma a classe empacotadora se torna compatível, em tipos, atributos e métodos, com a classe da aplicação e interface. O *framework* realiza automaticamente conversão de tipo entre classe da aplicação e classe empacotadora, o que promove a transparência deste arranjo à classe da aplicação. Outro benefício é que as classes da aplicação, por vezes indisponíveis em formato fonte, não precisam ser modificadas, tornando o *framework* retrocompatível com classes da aplicação.

A fim de garantir restrições de comparação entre entidades ou chaves, as interfaces *Entity* e *Key* adotam o padrão de projeto CRTP (*Curiously Recurring Template Pattern*; padrão com template curiosamente recursivo). A própria classe é passada por parâmetro neste padrão de projeto. Estas interfaces requerem um parâmetro de tipo, empregado em métodos de comparação. Assim, a assinatura dos métodos de comparação restringe comparações de igualdade à mesma classe de entidades ou chaves. Também é definido que a comparação de igualdade entre classes distintas tem resultado falso em qualquer caso.

O *framework* adquire significado através do módulo Meta. Em tempo de compilação são garantidas classes da aplicação com resolução e segurança de tipos (*type safety*). As meta-classes do *framework* são as interfaces *Entity, Key* e interfaces relacionadas a domínios de indexação *Order, Metric, Point, Rectangle, e Edition* (figura 2.5).

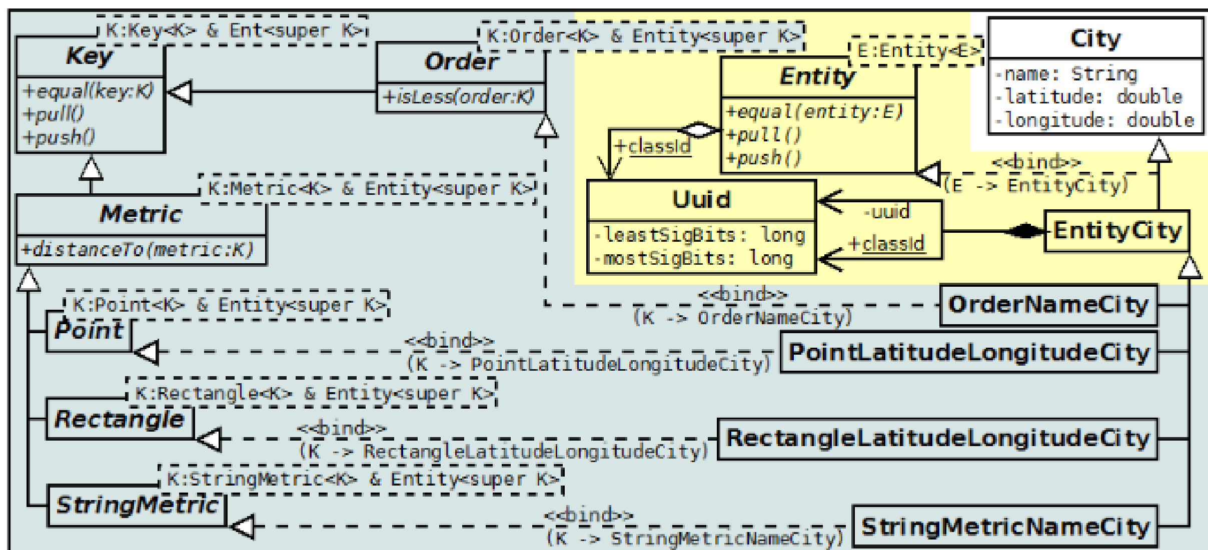

Figura 2.5: Framework Object-Inject: pacote Meta
Fonte: (CARVALHO et al., 2013)

Um objeto persistente é representado por uma Entidade, definida na interface *Entity*. Um objeto indexável e persistente é representado por uma chave através de atributos-chave, definidos na interface *Key*. Em uma aplicação, classes devem implementar a interface Entity para se tornarem persistentes e a interface *Key* para se tornarem indexáveis.

Métodos de identificação, comparação de igualdade e (des)serialização de entidades são especificados na interface *Entity*. Também há atributo no escopo da classe para identificação e comparação de igualdade entre classes de entidades.

Métodos de comparação de igualdade e de (des)serialização de chaves são especificados na interface *Key*. Ela é especializada em domínios de indexação, com especificações de métodos particulares à cada domínio, através das interfaces *Order* (domínio ordenável), *Edition* (domínio métrico textual) e *Metric* (domínio métrico), especializada em *Point* e *Rectangle* (domínio métrico espacial).

Um método de comparação de ordem entre chaves é especificado na interface *Order*. Este método também tem a função de implementar a ordenação de chaves que satisfaça as propriedades de relação de ordem total (transitividade, anti-simetria, totalidade) (ZEZULA et al., 2007). Métodos de acesso para a string (texto) de indexação são especificados pela interface *Edition*. Um método de comparação de distância entre chaves é especificado pela interface *Metric*. Este método é empregado em chaves com relação de (dis)similaridade, porém sem relação de ordem, satisfazendo propriedades de função de distância (não-negatividade, simetria, reflexividade) (ZEZULA et al., 2007). Métodos de acesso à coordenadas dimensionais são especificados pelas interfaces *Point* e *Rectangle*. Além disso, a interface *Rectangle* especifica métodos de acesso à coordenadas de origem e extensão em dimensões.

A figura 2.5 ilustra o módulo Meta apresentando o cenário de uma aplicação. *City* é uma classe da aplicação, representando uma cidade. *EntityCity* é uma classe empacotadora de persistência. São definidas as classes empacotadoras de indexação *OrderNameCity* e *OrderMayorCity* (chaves para indexação em domínio ordenável, através dos atributos name e mayor, respectivamente), *PointLatitudeLongitudeCity* e *RectangleLatitudeLongitudeCity* (chaves para indexação em domínio métrico espacial, através dos atributos latitude e longitude, simultaneamente), e *EditionNameCity* e *EditionMayorCity* (chaves para indexação em domínio métrico textual, através dos atributos name e mayor, respectivamente).

2.2.2 Metaprogramação para geração de classes

A automatização da implementação de interfaces de persistência e indexação foi introduzida em (OLIVEIRA, 2012) através da metaprogramação, combinando metaclasses do *framework* e metadados da aplicação (anotação).

A adoção de metadados e classes empacotadoras conduz à uma abordagem conveniente e pouco intrusiva para a aplicação. Usabilidade e intrusão são critérios relevantes para adoção e desenvolvimento de aplicações (HOU et al., 2008); (CORRITORE; WIEDENBECK, 2001).

A figura 2.6 mostra um diagrama de classes em UML que é responsável pela geração automática do código-fonte das classes empacotadoras para o framework.

A classe CodeGenerator(figura 2.6) delega a responsabilidade de gerar as strings que definem as classes empacotadoras. Suas especializações tratam da geração de classes: entidades persistentes (*EntityGenerator*) e chave-primária (*PrimaryKeyGenerator*). A generator class *EntityGenerator*, cria o código-fonte que descende da classe de aplicação e implementa a interface *Entity*(figura 2.5). De forma semelhante, uma nova especialização é feita a partir de *CodeGenerator* gerando as classes:

- a classe *PrimaryKeyGenerator* que implementa as responsabilidades definidas pela interface *Order*(figura 2.5);

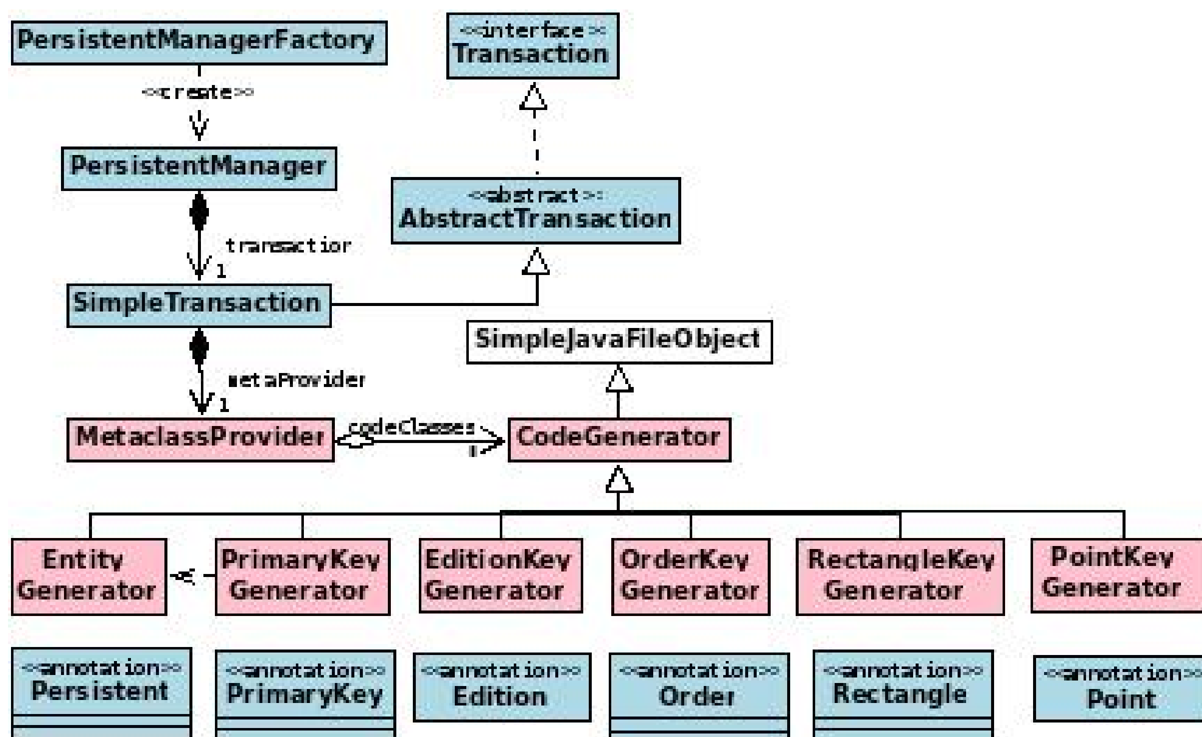

Figura 2.6: Diagrama de classes para geração de classes

- a classe *EditionKeyGenerator* que implementa as responsabilidades definidas pela interface *Edition*(figura 2.5);
- a classe *OrderKeyGenerator* que implementa as responsabilidades definidas pela interface *Order*(figura 2.5);
- a classe *RectangleKeyGenerator* que implementa as responsabilidades definidas pela interface *Rectangle*(figura 2.5);
- a classe *PointKeyGenerator* que implementa as responsabilidades definidas pela interface *Point*(figura 2.5);

A classe *MetaclassProvider*(figura 2.6) utiliza reflexão para identificar as informações dos campos para cada definição de entidade ou da chave primária.

As classes *PersistentManagerFactory, PersistentManager, Transaction, AbstractTransaction e SimpleTransaction* apresentadas na figura 2.6 oferecem uma abstração de gerenciamento de objetos semelhante a padronização *Java Persistence* API (KEITH; SCHINCARIOL, 2009).

Finalmente neste diagrama são apresentadas as anotações: *@Persistent, @PrimaryKey, @Order, @Spatial, @Edition e @Point*. A anotação para persistência de classe é *@Persistent* que é aplicável a tipos. A seguir são descritas as anotações de indexação, aplicáveis a atributos (campos):

- *@PrimaryKey*: atributo é chave primária da classe, com relação de ordem, indexado

em árvore B+.

- *@Order*: atributo com relação de ordem, indexado em árvore B+.
- *@Rectangle*: atributo com relação espacial, indexado em árvore R.
- *@Edition (distance=levenshtein, dameraulevenshtein, xuDamerau)*: atributo com relação de distância entre cadeias de caracteres, indexado em árvore M; funções de distância podem ser levenshtein e variantes damerauLevenshtein (transposição) e xuDamerau (proteínas).
- *@Point (distance=euclidean, manhattan, earthSpherical)*: atributo com relação de distância, indexado em árvore M; função de distância pode ser euclidean (euclidiana), manhattan, e esférica terrestre.

A partir de uma anotação de indexação em um atributo de classe, o gerador de classes cria o código-fonte de uma classe empacotadora com as implementações da interface Key e subinterface correspondente à anotação e um índice secundário para chaves. Algumas anotações são marcadoras e outras contém elementos para configuração de indexação.

2.3 Trabalhos Correlacionados

Um dos primeiros trabalhos que relacionam acesso a dados por meio de serviços web foi (RAMAN et al., 2002). Neste trabalho já se nota a crescente evolução das bases de dados corporativas e sua dificuldade de interoperação e distribuição, causando muitas vezes replicação de dados e dificuldade de dinamismo e autonomia. (RAMAN et al., 2002) propõe que a complexidade de comunicação entre bases de dados distintas deve ser transparente para as aplicações. Para isto propõe uma camada de serviços intermediária operando sobre a web por meio de OGSA(Open Grid Services Architeture) (TALIA, 2002). OGSA é baseada em tecnologias de serviços web, tais como WSDL e SOAP.

Em (HANSEN et al., 2003) também é observada a necessidade de acesso a dados em bases heterogêneas buscando a flexibilidade dos serviços web para armazenar e recuperar dados de diferentes plataformas. São destacadas no trabalho as tecnologias XML, HTTP, WSDL, SOAP, UDDI.

Em (ANTONIOLETTI et al., 2006) é proposto o WS-DAI (Web Service Data Access and Integration), um framework para acesso a dados e integração, ao qual define interfaces de serviços web para acessar fontes de dados. São descritas propriedades que podem ser usadas para descrever acesso e armazenamento em um serviço de dados. WS-DAI também é baseado na descrição e publicação de serviços por meio de WSDL e comunicação por meio de SOAP e HTTP.

Recentemente o termo "Dados como Serviço" ou "Data-as-a-service", sigla "DaaS" tem ganhado espaço no meio científico. Em (RAJESH et al., 2012) é abordado o tema destacando a importância da arquitetura orientada a serviços para manipulação de dados. Neste trabalho, vários "dados" chamados de produtos são oferecidos aos clientes como serviços. Além disso é discutido temas como segurança dos dados, privacidade, governança de dados e métodos CRUD(Create, Read, Update e Delete).

Em (LIU et al., 2014) é proposto uma arquitetura de serviço de dados para o sistema de informação na empresa moderna. O trabalho de (LIU et al., 2014) é motivado pela integração de uma variedade de sistemas de informação com vários repositórios de dados heterogêneos. Para isto, é proposto a criação de um serviço de dados operando sobre a internet. O objetivo é compor um serviço capaz de obter dados de fontes heterogêneas e também compatilhá-los.

(LIU et al., 2014) define um serviço de dados como uma especialização de serviços web que podem ser implantados sobre repositórios de dados ou outros serviços que emcapsulam grande quantidade de dados. Estes serviços de dados foram divididos em 5 camadas:

- Camada de Aplicação: Interface que provê o serviço de dados para a aplicação. A requisição de dados é submetida para camada de serviço e os resultados são retornados para a aplicação.

- Camada de Serviço: Publica uma variedade de funções baseadas no modelo de dados. Todas as funções são publicadas na linguagem WSDL.

- Camada de Integração: Camada que contém o mapeamento do esquema de dados e gerencia as consultas de dados.

- Camada Semântica: Responsável por estabelecer associação entre relacionamentos semânticos no mapeamento do esquema de dados.

- Camada de Dados: Contém um gerenciador de fontes de dados e metadados.

Ainda neste trabalho, a associação semântica dos dados em bases heterogêneas é realizada de forma manual.

Em (PRETTY; BLACKWELL, 2014) é proposto uma metodologia de acesso a dados via serviços web. O repositório de dados usado nesse trabalho é o MDSPlus (STILLERMAN et al., 1997) que é baseado no modelo cliente / servidor e que apresenta significativa complexidade de implementação por parte de novos usuários e colaboradores. Motivado por estas características, (PRETTY; BLACKWELL, 2014) propôs um sistema de acesso a dados com uma interface intuitiva sem modificar a configuração existente de MDSPlus. Para essa proposta a abordagem escolhida foi a de web services que prove um método prático de acesso utilizando somente da infraestrutura web.

Uma característica desse trabalho está na utilização de REST (RICHARDSON; RUBY, 2008) ao invés de SOAP para prover uma familiaridade com URL's e navegação em browser. Uma limitação nesse trabalho está no repositório de dados MDSPlus que não provê mecanismos de acesso a dados de outros bancos de dados.

Em (JONES et al., 2014) é proposto um adaptador que atua como um serviço web para realizar a interoperabilidade entre bases Linked Open Data (LOD) e Sistema de Informação Geográfico (Geographic Information System - GIS) usando Serviços de Características na Web (WFS). Bases Linked Open Data (LOD) (BIZER et al., 2009) usam Resource Description Framework (RDF) para representar uma informação na Internet.

Serviços de Características na Web (WFS) são um padrão de serviço web para requisição de dados geográficos que usa protocolo HTTP com respostas em documentos XML. Sua contribuição está na utilização de padrões para disponibilizar informação de dados LOD de um GIS com Serviços de Características na Web (WFS).

Em (WANGUE; ZHAO, 2014) é proposto um método de segurança de acesso a banco de dados por meio de serviços web. É discutido no trabalho a vulnerabilidade de controle de acesso aos bancos de dados e propoem um middleware entre as bases de dados e as aplicações clientes. A abordagem é baseada em uma política de segurança que usa uma matriz de controle de

acesso implementada em um serviço web para permitir ou negar acesso a um banco de dados por uma aplicação.

(WANGUE; ZHAO, 2014) propõe um serviço web entre os clientes e os repositórios de dados. Os clientes requisitam uma conexão de acesso ao banco de dados ao serviço web. O serviço web autentica a conexão baseado em um repositório de usuários e uma matriz de controle de acesso.

Capítulo 3

Serviço web para acesso a dados

Este capítulo apresenta a principal contribuição deste trabalho, que é o desenvolvimento de um serviço web para armazenamento e recuperação de dados utilizando-se do *framework ObInject* (CARVALHO et al., 2013) para persistência.

Este serviço web deve oferecer as seguintes funcionalidades:

- Registro de novos modelos de negócios. Novos modelos de negócio podem ser registrados a qualquer momento no serviço.

- Disponibilização de serviços CRUD. Criação e disponibilização de serviços CRUD para armazenamento e recuperação de dados para cada entidade do modelo de negócios registrado.

- Portabilidade. Garantir portabilidade para qualquer linguagem de programação e sistema operacional utilizados pelas aplicações clientes. A assinatura dos serviços CRUD deve garantir a manipulação de dados utilizando somente tipos comuns a todas linguagens de programação.

Esta metodologia transfere a responsabilidade do desenvolvimento da camada de acesso a dados (DAO) para o provedor de serviço, garantindo que os detalhes de implementação de persistência e indexação de dados fiquem transparentes para as aplicações clientes. Isto também permite uma maior agilidade no desenvolvimento de aplicações clientes, vistas que, estas não necessitam se concentrar nos detalhes de persistência e consomem os serviços (CRUD) sob demanda.

Nas próximas seções serão apresentadas a arquitetura do serviço web para acesso a dados e o fluxo de ações para uma aplicação cliente registrar seu modelo de negócios e consumir serviços CRUD.

3.1 Arquitetura

A arquitetura para o serviço web para acesso a dados apresentada na figura 3.1 é dividida em consumidores de serviço e provedores de serviços. Os potenciais consumidores de serviço são aplicações que necessitam de persistência de dados que podem ter sido desenvolvidas em qualquer linguagem de programação, tais como, Java, C++ e PHP *(Hypertext Preprocessor)*. O provedor de serviços hospeda o serviço web para acesso a dados que possibilita o registro de aplicações clientes e disponibiliza serviços CRUD dos quais as aplicações clientes tornam-se consumidoras.

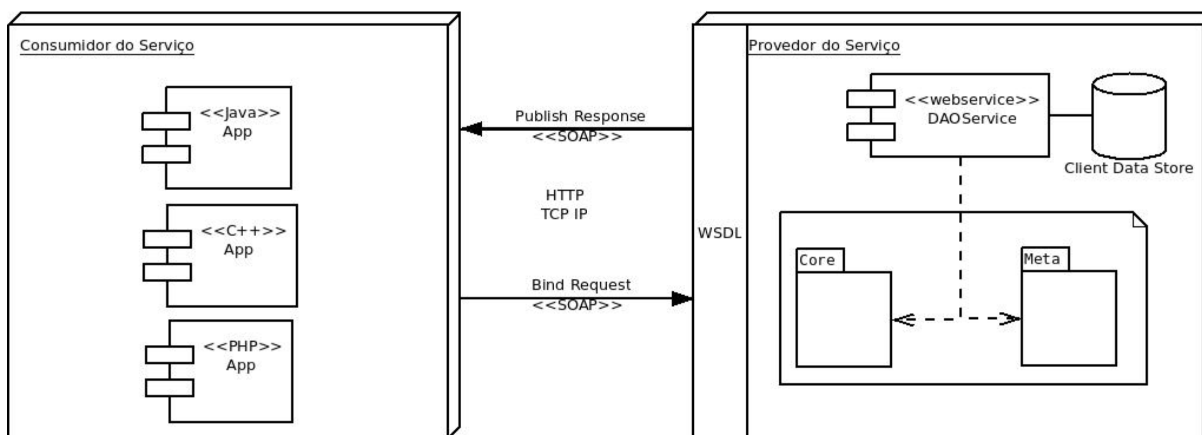

Figura 3.1: Arquitetura do serviço web para acesso a dados
Fonte: O autor

3.1.1 Provedor de serviço de acesso a dados

O provedor de serviço hospeda o serviço web para acesso a dados onde potenciais aplicações clientes podem se tornar consumidoras deste serviço.

O provedor de serviço foi desenvolvido na linguagem de programação Java, a mesma utilizada pelo *framework ObInject* que é responsável pela persistência e indexação dos dados. Além disso, três bibliotecas foram utilizadas: JAX-WS *(Java API for XML Web Services)* (KOHLERT; GUPTA, 2007) que permite a criação e publicação de serviços web baseados nas tecnologias XML, WSDL, SOAP (seção 2.1.2); Java *Reflection* (FORMAN et al., 2004) que é responsável pela instrumentação dos métodos e atributos das classes em tempo de execução; Java *Compiler* (ADL-TABATABAI et al., 1998) que é responsável por compilar e disponibilizar um objeto Java em tempo de execução.

Inicialmente o provedor recebe uma requisição para registrar o modelo de negócio da aplicação cliente. Neste registro são geradas:

1. Classes de modelo de negócio. Estas classes representam o modelo de negócio que a

aplicação cliente deseja persistência.

2. Classes de persistência. Estas classes são responsáveis pelo acoplamento das classes do modelo de negócio com o *framework ObInject*. Sua principal característica é mapear atributos do objeto em formato serializável.

3. Classe de serviços CRUD. Classe responsável por disponibilizar métodos CRUD para cada entidade do modelo de negócio.

A classe responsável pela geração automática dos três tipos de classes mencionados anteriormente é ***DAOService***, ilustrada na figura 3.2. Esta classe é anotada por *@webservice* (JAX-WS) indicando que contém serviços web. Os métodos *begin, create e commit* anotados por *@webmethod* (JAX-WS) são serviços web publicados na WSDL do serviço. Este serviço é publicado por um *EndPoint* através da URL http://*hostname*:8080/daoservice?wsdl.

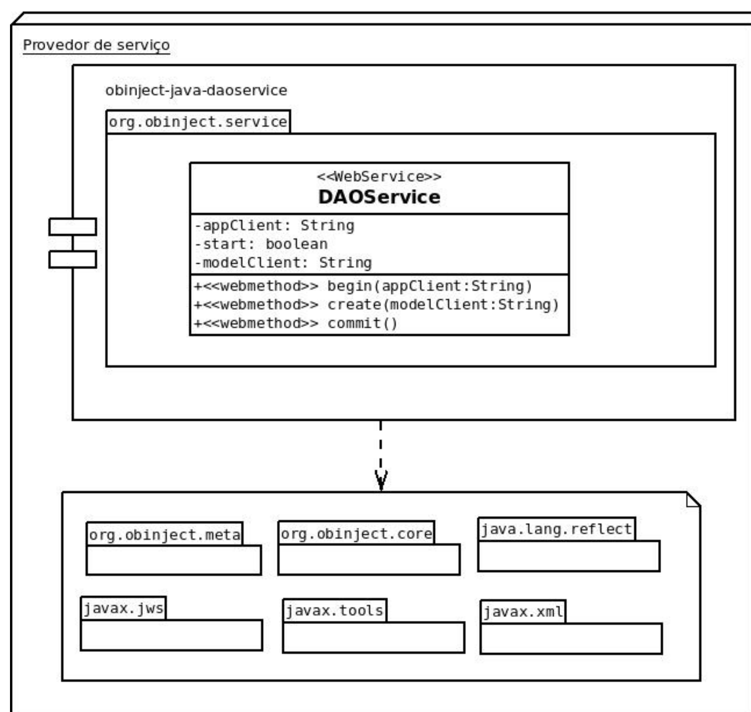

Figura 3.2: Diagrama de Classes DaoService
Fonte: O Autor

A aplicação cliente que deseja utilizar os serviços web de armazenamento e recuperação de dados deve invocar os métodos na seguinte ordem: *begin, create e commit.*

O método ***begin*** inicia o registro de uma nova aplicação cliente no provedor de serviço. Este método recebe uma *string* por parâmetro que é utilizado em três definições do serviço: o nome do pacote que contém as classes geradas, o nome da classe que define os serviços CRUD e o nome da URI de publicação dos serviços CRUD.

O método ***create*** é responsável pelo registro no provedor de serviço do modelo de negócio da aplicação cliente. Este registro se dá por meio da criação das classes Java de modelo de negócio e de persistência. Este método recebe uma *string* por parâmetro que deve conter a definição do modelo de negócio da aplicação cliente. O modelo de negócio deve ser definido através de entidades com seus respectivos atributos e relacionamentos. É possível definir herança e atributos de relacionamento entre entidades. Cada entidade deve possuir ao menos um atributo-chave. A *string* da definição do modelo de negócio deve obedecer a seguinte gramática:

```
<Entidade[:SuperEntidade]>[,][<[#]atributo:tipo >][ ,][...][;][...]
```

Nesta gramática o nome da entidade e seus atributos são separados por vírgula(,) e as entidades são separadas por ponto-e-vírgula(;). Caso a entidade herde de outra entidade, deve-se incluir dois-pontos(:) a frente da entidade com o nome de sua entidade-mãe. Os atributos e seus tipos são separados por dois-pontos(:). Os atributos que são chaves da entidade devem iniciar com malha(#).

Uma restrição nesta gramática é que somente são mapeados relacionamentos com multiplicidade um. No caso de multiplicidades n para n entre entidades, deve-se definir uma entidade de relacionamento entre estas entidades. Como exemplo, suponha uma aplicação cliente chamada "Escola". Seu modelo de negócio é apresentado na figura 3.3. Neste caso de uso, as entidades professor e aluno herdam de pessoa. Todo aluno está vinculado a um curso, toda disciplina está vinculada a um professor e toda turma está vinculada a uma disciplina. Cada aluno pode estar vinculado a n turmas e cada turma pode ter n alunos vinculados.

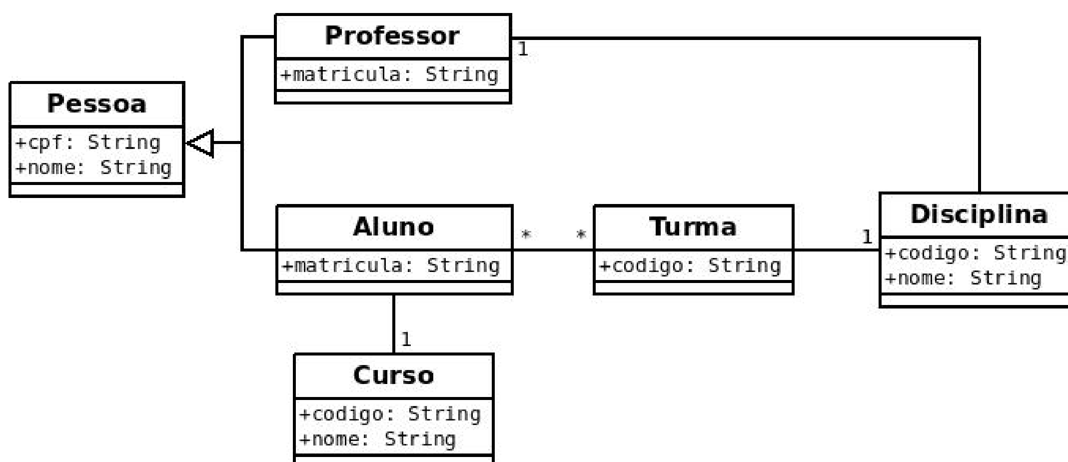

Figura 3.3: Caso de Uso de uma Escola
Fonte: O Autor

Neste exemplo, as entidades aluno e turma possuem um relacionamento de n para n. Assim, nesta definição deverá ser criada uma entidade de relacionamento entre as duas entidades,

que será chamada AlunoTurma que deverá conter referências para aluno e turma. Baseado na gramática apresentada, a *string* da definição deste modelo de negócio a ser passada no método *create* deverá ser:

```
Curso,#codigo:String,nome:String;
Pessoa,#cpf:String,nome:String;
Aluno:Pessoa,matricula:String,curso:Curso;
Professor:Pessoa,matricula:String;
Disciplina,#codigo:String,nome:String,professor:Professor;
Turma,#codigo:String,disciplina:Disciplina;
AlunoTurma,#codigo:String,aluno:Aluno,turma,Turma
```

Após receber o parâmetro, o método processa a *string* para criar as classes em Java do modelo de negócio da aplicação cliente. Todas as classes geradas são anotadas por *@Persistent* do módulo *meta* do *framework ObInject* (seção 2.2.2), indicando que as mesmas serão persistidas e indexadas pelo *framework*. Os atributos-chave são anotados por *@PrimaryKey* do módulo *meta* do *framework ObInject* (seção 2.2.2). Através de metaprogramação por reflexão, as classes são geradas em tempo de execução pelo método *create*. As figuras 3.4a e 3.4b apresentam as classes Pessoa e Aluno respectivamente geradas no provedor de serviço pelo método *create*.

```java
package Escola;
import org.obinject.annotation.Persistent;
import org.obinject.annotation.PrimaryKey;
@Persistent
public class Pessoa {
    @PrimaryKey
    private String cpf;
    public String getCpf() {
        return cpf;
    }
    public void setCpf(String cpf) {
        this.cpf = cpf;
    }
    private String nome;
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

(a)

```java
package Escola;
import org.obinject.annotation.Persistent;
import org.obinject.annotation.PrimaryKey;
@Persistent
public class Aluno extends Pessoa {
    private String matricula;
    public String getMatricula() {
        return matricula;
    }
    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }
    private Curso curso;
    public Curso getCurso() {
        return curso;
    }
    public void setCurso(Curso curso) {
        this.curso = curso;
    }
}
```

(b)

Figura 3.4: Classe Pessoa(a) e classe Aluno(b) geradas pelo método *create*
Fonte: O Autor

Todas as classes geradas pelo provedor de serviço para a aplicação cliente são armazenadas em um pacote com o nome da aplicação cliente definido no método *begin*. A partir das anotações definidas nas classes de modelo, o método *create* também gera as classes de ligação entre o

modelo de negócio e o módulo *core* do *framework ObInject*. Estas classes serão responsáveis pela persistência e indexação de cada entidade da aplicação cliente. Para cada entidade, serão geradas as classes *Entity<Entidade>* e *PrimaryKey<Entidade>*.

Finalmente, o método ***commit*** é responsável por finalizar o processo de registro da aplicação cliente através da criação e disponibilização dos serviços web (CRUD) para manipulação de dados de cada entidade definida no método *create*. Para cada aplicação cliente registrada no provedor de serviço, é gerada através de metaprogramação, uma classe de serviços com os respectivos métodos CRUD para cada entidade da aplicação cliente. Esta classe é nomeada com o nome da aplicação cliente definido no método *begin* e é anotada por *@webservice* indicando que contém serviços web.

Para cada entidade do modelo de negócio da aplicação cliente, são criados os quatro métodos CRUD anotados por *@webmethod*:

- ***create<Entidade>***: Método para persistir um novo objeto desta entidade. Este método recebe por parâmetro todos os atributos da entidade em ordem de definição. Em caso de entidade-filha, os atributos da entidade-mãe vêm primeiro e em seguida os atributos da própria entidade. Para atributos de relacionamentos, deve-se informar o atributo-chave da entidade relacionada.

- ***retrieve<Entidade>***: Método para consultar e recuperar um determinado objeto desta entidade. Este método recebe por parâmetro o atributo-chave da entidade para realizar a consulta. Caso encontre o objeto, é retornada uma uma *string* JSON *JavaScript Object Notation* (CROCKFORD, 2001) contendo os dados de todos atributos da entidade, inclusive os de herança. para atributos de relacionamento, é retornado o valor do atributo-chave da entidade relacionada.

- ***update<Entidade>***: Método para atualizar um determinado objeto de desta entidade. Este método recebe os mesmos parâmetros do método *create<Entidade>*.

- ***delete<Entidade>***: Método para excluir um determinado objeto de desta entidade. Este método recebe por parâmetro o atributo-chave da entidade a ser excluída.

Exemplo: A figura 3.5 representa a classe de serviços CRUD gerada para a aplicação cliente "Escola". Para ilustração, nesta figura somente estão os métodos *create* e *retrieve* da entidade "Aluno".

Pode ser visto na figura que o método *createAluno* recebe por parâmetro os atributos de Pessoa e Aluno. Aluno possui uma referência para Curso, onde no caso, é passado por parâmetro o atributo-chave de Curso. Através do atributo-chave de Curso o método busca o objeto referenciado para repassar ao atributo de relacionamento de Aluno. No método *retrieveAluno*, deve-se informar o atributo-chave de aluno para consulta, onde no caso é

```
@WebService
public class Escola {
    @WebMethod
    public boolean createAluno(String cpf, String nome, String matricula, String curso) {
        Aluno aluno = new Aluno();
        aluno.setCpf(cpf);
        aluno.setNome(nome);
        aluno.setMatricula(matricula);
        PrimaryKeyCurso pkCurso2 = new PrimaryKeyCurso();
        pkCurso2.setCodigo(curso);
        Uuid uuidCurso2 = EntityCurso.primaryKeyCursoStructure.find(pkCurso2);
        aluno.setCurso((Curso) EntityCurso.entityStructure.find(uuidCurso2));
        Entity newEntity = new EntityAluno(aluno);
        return newEntity.inject();
    }
    @WebMethod
    public String retrieveAluno(String cpf) throws JSONException {
        PrimaryKeyAluno pkp = new PrimaryKeyAluno();
        pkp.setCpf(cpf);
        Uuid uuid = EntityAluno.primaryKeyAlunoStructure.find(pkp);
        Aluno aluno = (Aluno) EntityAluno.entityStructure.find(uuid);
        JSONArray ja = new JSONArray();
        JSONObject jo = new JSONObject();
        jo.put("matricula", aluno.getMatricula());
        jo.put("curso", aluno.getCurso().getCodigo());
        jo.put("cpf", aluno.getCpf());
        jo.put("nome", aluno.getNome());
        ja.put(jo);
        return "{rows: " + ja.toString() + "}";
    }
```

Figura 3.5: Classe de serviços "Escola" gerada pelo método *commit*

Fonte: O Autor

"cpf". A partir do atributo-chave, o método busca o objeto e retorna seus atributos através de uma *string* JSON. Para o atributo de referência "curso", é retornado o atributo-chave de Curso.

A classe criada que contém os serviços web CRUD da aplicação cliente, atuará como um novo serviço a ser executado no provedor de serviço. Para isto, ela deverá ser executada em paralelo com o serviço *DAOService*, assim como outras possíveis aplicações clientes do provedor de serviço. Para isto ser possível, este serviço deve ser executado como *Thread*, possibilitando que o provedor de serviço hospede múltiplos serviços de armazenamentos. Assim, o método *commit* também gera a classe responsável por publicar o serviço, *(EndPoint)*, que será executado como *Thread*. A figura 3.6 mostra a classe de publicação dos serviços CRUD para a aplicação cliente "Escola". Neste exemplo, o método estático *publish* da classe *Endpoint* publica os serviços web da classe "Escola" por meio da URL "http://*hostname*:8080/Escola".

Ao finalizar a criação da classe de serviços, o método *commit* efetua a compilação de todas as classes geradas e inicia por meio de reflexão, a *thread* dos serviços CRUD para a aplicação cliente. Após o término da execução do método *commit*, os serviços CRUD da aplicação cliente já estão disponíveis para utilização por meio da URL http://*hostname*:8080/*nome da aplicação cliente*?wsdl.

```
package Escola;
import javax.xml.ws.Endpoint;
public class EscolaRun extends Thread {
    public void run() {
        Escola escola = new Escola();
        Endpoint.publish("http://hostname:8080/Escola", escola);
    }
}
```

Figura 3.6: Classe de publicação dos serviços web "Escola" gerada pelo método *commit*
Fonte: O Autor

3.1.2 Consumidor de Serviço

Qualquer aplicação que necessite persistir dados poderá ser um potencial cliente do serviço. Não há restrição de linguagem de programação e nem de sistema operacional, já que a assinatura dos métodos de serviços CRUD somente utilizam tipos primitivos compatíveis em todas linguagens. Cada linguagem de programação possui seus próprios mecanismos para acesso a serviços web através do protocolo SOAP.

Para uma aplicação registrar seu modelo de negócio no provedor de serviço e tornar-se cliente, é necessário executar os três métodos do serviço web *begin, create e commit* apresentados na seção anterior.

3.2 Fluxo de Funcionamento

O fluxo de funcionamento desde o registro de uma aplicação cliente até o consumo dos serviços CRUD é mostrado no diagrama 3.7. Os itens de 1 a 7 correspondem à sequência de passos para registro de uma aplicação cliente e são executados apenas uma única vez. Uma vez registrada a aplicação, somente são executados os itens de 8 a 12 conforme demanda da aplicação cliente.

O fluxo de funcionamento é descrito da seguinte forma:

1. Aplicação cliente estabelece conexão com o serviço web para acesso a dados;
2. Aplicação cliente executa o método *begin* passando por parâmetro o nome da aplicação;
3. Aplicação cliente executa o método *create* passando por parâmetro o modelo de negócio;
4. Provedor de serviço gera classes do modelo de negócio do cliente e gera classes de acoplamento com *framework ObInject*;
5. Aplicação cliente executa o método *commit*;

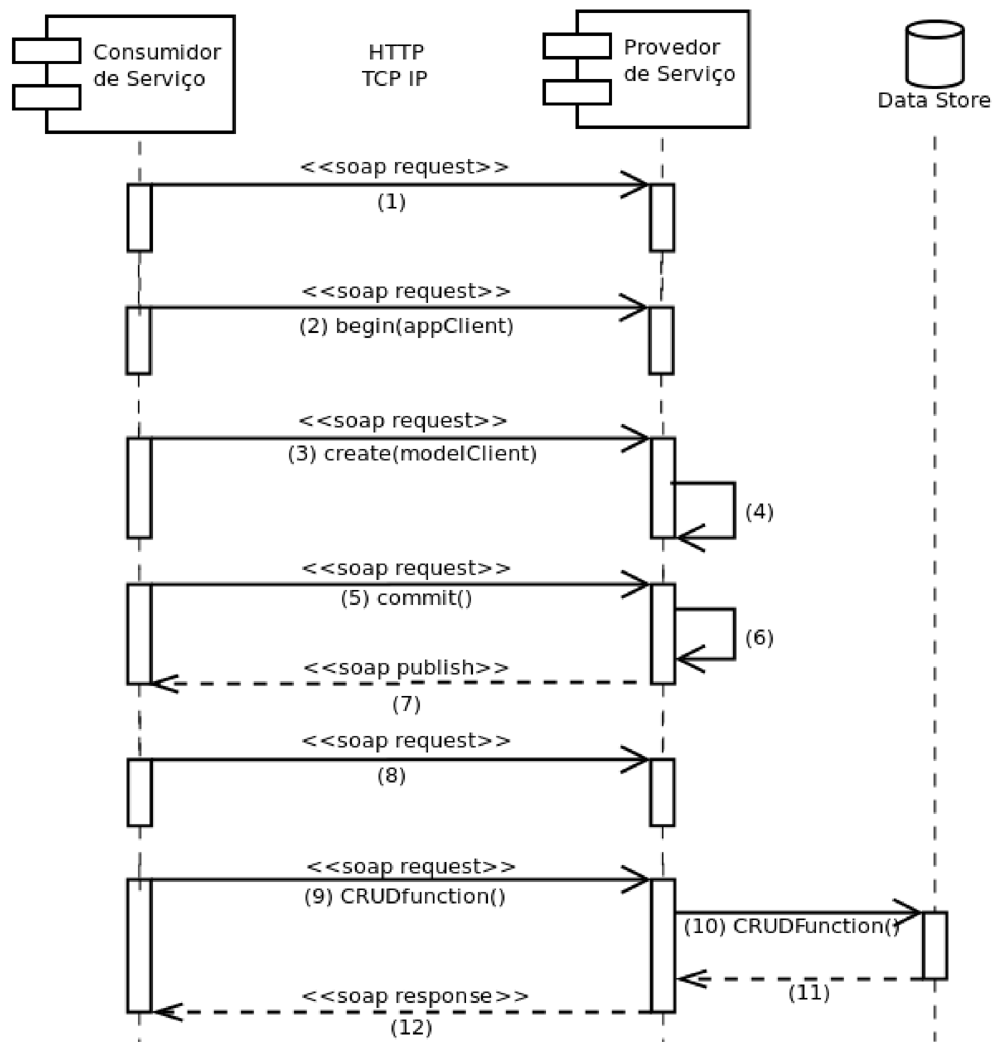

Figura 3.7: Fluxo de Funcionamento
Fonte: O autor

6. Provedor de serviço gera classe de serviços web CRUD e compila todas classes geradas.

7. Provedor de serviço inicia serviços CRUD para aplicação cliente e publica a WSDL.

8. Aplicação cliente estabelece conexão com o serviço web CRUD;

9. Aplicação cliente executa um método CRUD;

10. Provedor de serviço executa método CRUD nas estruturas de armazenamento de dados do *framework ObInject*;

11. *Framework ObInject* envia para provedor de serviço a resposta da operação CRUD executada;

12. Provedor de serviço envia para aplicação cliente a resposta da operação CRUD executada;

3.3 Considerações Finais

O *framework ObInject* foi desenvolvido na linguagem de programação Java e fornece mecanismos de persistência e indexação para objetos de classes Java. Esta característica previa que o modelo de negócio da aplicação cliente estivesse definido em classes Java. Isto limitava a interoperabilidade de aplicações desenvolvidas em outras linguagens para persistirem com o *framework*. Neste contexto, a solução de se criar uma linguagem de registro de objetos genérica, baseada no conceito chave-valor, se mostrou simples e eficaz. Desta forma, o servidor se encarregou de criar o modelo de negócio em Java da aplicação cliente e oferecer métodos CRUD com parâmetros de tipos primitivos, possibilitando que os objetivos iniciais do trabalho pudessem ser atingidos.

Capítulo 4

Experimentos

Este Capítulo descreve os experimentos realizados no serviço web para acesso a dados no *framework ObInject*. A primeira seção deste capítulo descreve os objetivos dos experimentos. A segunda seção descreve análises funcionais do serviço realizadas com aplicações clientes escritas nas linguagens de programação Java, C++ e PHP. A seção também descreve o cenário de um caso de uso do filme "Piratas do Caribe" para realização dos experimentos. Na terceira seção é apresentada uma análise de desempenho do serviço para inserção de uma carga de objetos da aplicação cliente "Piratas do Caribe". A quarta seção apresenta a conclusão dos resultados dos experimentos.

4.1 Objetivo

O objetivo deste capítulo é avaliar se os propósitos deste trabalho foram atingidos. Para isto, serão realizados dois tipos de experimentos. O primeiro experimento é uma análise funcional que avalia a eficácia das funcionalidades do serviço web de acesso a dados. Dentre elas pode-se destacar: registro de um modelo de negócio, disponibilização e execução de serviços CRUD e portabilidade do serviço, que garante que aplicações desenvolvidas em várias linguagens de programação possam se tornar clientes do serviço. Para realizar este experimento é utilizado um caso de uso do filme "Piratas do Caribe" como modelo de negócio para registro no provedor de serviço. Serão analisados casos de aplicações clientes nas linguagens Java, C++ e PHP. O segundo experimento é uma análise de desempenho que avalia o comportamento do serviço web de acesso a dados em um cenário com uma carga de inserção de objetos. Para efeito de comparação, além de realizar persistência com o *framework Obinject*, este experimento também realiza persistência utilizando o *framework Hibernate* com banco de dados MySQL. Serão comparados tempos de inserção e utilização de espaço em disco.

4.2 Caso de Uso: Piratas do Caribe

A seguir é descrito o cenário do filme “Piratas do Caribe” que será usado como classes de negócio para esse caso de uso:

Um navio é descrito por seu nome, quando foi construído, seu tamanho, a quantidade de mastros e a quantidade de velas. Todo navio é identificado por seu nome. Todo navio é um navio pirata ou navio de guerra ou um navio mercante. Navios piratas ou de guerras são descritos também pela quantidade de canhões. Navios mercantes são descritos também por sua capacidade. Um navio pirata pode combater com vários navios de guerra e/ou vários navios mercantes. Navios de guerra e navios mercantes podem ser combatidos por muitos navios piratas. Navios mercantes podem transportar muitas mercadorias. Uma mercadoria pode ser transportada por vários navios mercantes. Uma mercadoria é descrita por seu código, preço e peso. Toda mercadoria é identificada por seu código. Um tripulante é descrito por seu nome, nacionalidade, apelido e a qual navio pertence. Todo tripulante é identificado por seu nome. Todo tripulante é um tripulante auxiliar ou um navegador. Tripulantes auxiliares também são descritos por suas horas de navegação. Tripulantes navegadores podem ser capitães ou não. Um local é descrito por seu código, país, latitude e longitude. Todo local é identificado por seu código. Todo local é uma costa ou uma ilha. Costas também são descritas por sua cidade e seu estado. Ilhas também são descritas por seu nome e área. Um deslocamento é descrito por seu código, a data de saída, a data de chegada, a distância, o navio que executou o deslocamento, o local de origem e o local de destino. Todo deslocamento é identificado por seu código. Um mapa é descrito por seu nome, a quantidade de rotas, a escala e o navegador responsável. Todo mapa é identificado por seu nome. Todo mapa é um mapa de navegação ou um mapa de tesouro. Mapas de navegação são descritos pelo oceano e pela costa correspondente. Mapas de tesouro são descritos pelo valor do tesouro e a ilha correspondente.

A figura 4.1 representa o diagrama de classes para o cenário “Piratas do Caribe”.

Para uma aplicação cliente registrar seu modelo de negócio no serviço web para acesso a dados, é necessário invocar os métodos *begin, create e commit* do serviço. No método *create*, deve ser informado o modelo de negócio da aplicação cliente usando a gramática estabelecida na seção 3.1.1. Segue abaixo a definição do modelo de negócio para o cenário:

```
Local,#codigo:int , pais:String , lat:double , lont:double;
Costa:Local , cidade:String , estado:String;
Ilha:Local ,nome:String , area:float;
Navio,#nome:String , construido:String ,tamanho:int , qtdMastro:int , qtdVelas:int;
Tripulante,#nome:String , nacionalidade:String , apelido:String , navio:Navio;
Auxiliar:Tripulante , horasNavegacao:int;
```

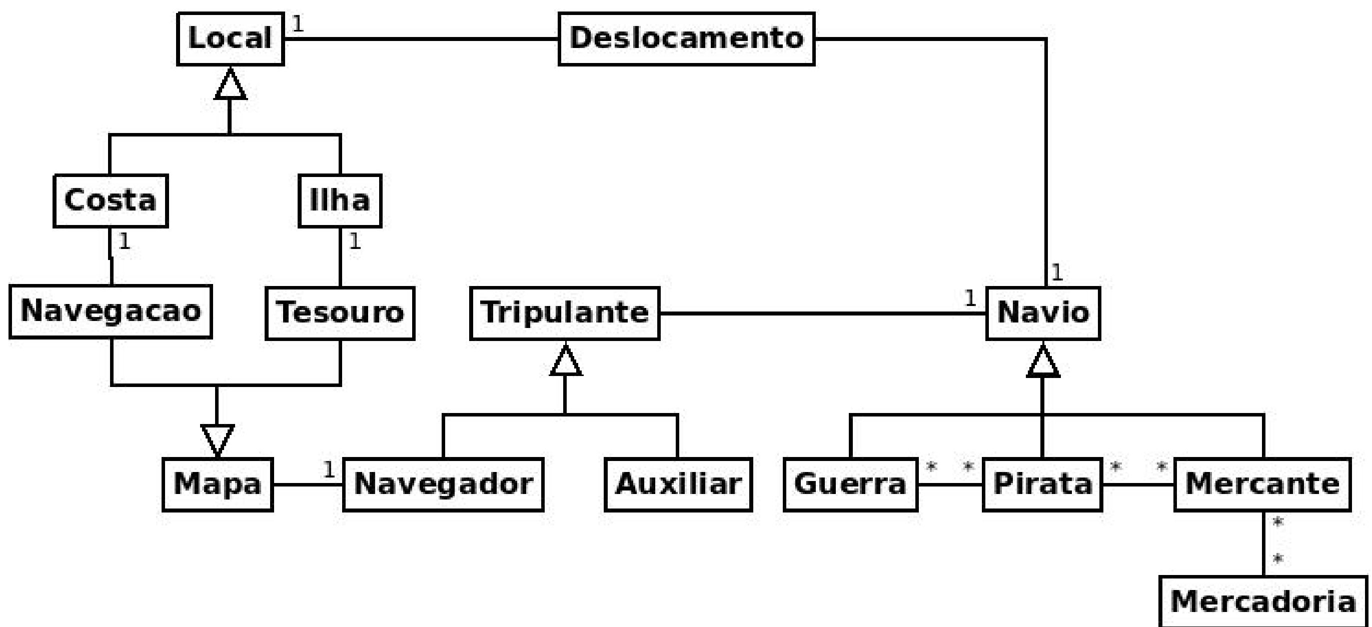

Figura 4.1: Diagrama de Classes - Caso de Uso "Piratas do Caribe"
Fonte: O Autor

```
Navegador:Tripulante, capitao:String;
Deslocamento,#codigo:int, dtOrigem:String, dtDestino:String, distancia:float,
    navio:Navio, localOrigem:Local, localDestino:Local;
Pirata:Navio, qtdCanhoes:int;
Guerra:Navio, qtdCanhoes:int;
Mercante:Navio, capacidade:int;
Mercadoria,#codigo:int, preco:float, peso:float;
PirataGuerra,#codigo:int, pirata:Pirata, guerra:Guerra;
PirataMercante,#codigo:int, pirata:Pirata, mercante:Mercante;
MercadoriaMercante,#codigo:int, mercante:Mercante, mercadoria:Mercadoria;
Mapa,#nome:String, qtdRota:int, escala:double, navegador:Navegador;
Navegacao:Mapa, oceano:String, costa:Costa;
Tesouro:Mapa, valorTesouro:float, ilha:Ilha
```

4.2.1 Análise Funcional: Aplicação consumidora em Java

A ferramenta *wsimport* da API JAVA JAX-WS (seção 2.1.3), é capaz de criar uma infraestrutura local para consumir serviços web através do WSDL. Esta ferramenta gera código-fonte local baseado no JAX-WS, causando o mínimo esforço para aplicações Java consumirem serviços web. Desta forma, a aplicação cliente executa métodos locais que encapsulam a conexão HTTP e executam os serviços web.

Para criar a infraestrutura necessária para execução dos métodos *begin, create e commit*, deve-se executar o comando *wsimport*, descrito na sintaxe abaixo:

```
wsimport -keep -s pastaaplicacao/src http://hostname:8080/daoservice?wsdl
```

Ao executar este comando, é criado na pasta /src da aplicação cliente um pacote "obinject.java.daoservice" com classes consumidoras dos serviços especificados na WSDL. Nesta infraestrutura, os métodos *begin, create e commit* estão encapsulados na classe *DAOService* gerada pela importação e estão prontos para serem executados. A figura 4.2 apresenta o código Java que executa os métodos *begin, create e commit* após importação pela ferramenta *wsimport*. É definido no método *begin* o nome "Piratas" para a aplicação.

```java
DAOService daoservice = new DAOServiceService().getDAOServicePort();
daoservice.begin("Piratas");
daoservice.create("Local,#codigo:int,pais:String,lat:double,lont:double;"
+ "Costa:Local,cidade:String,estado:String;"
+ "Ilha:Local,nome:String,area:float;"
+ "Navio,#nome:String,construido:String,tamanho:int,qtdMastro:int,qtdVelas:int;"
+ "Tripulante,#nome:String,nacionalidade:String,apelido:String,navio:Navio;"
+ "Auxiliar:Tripulante,horasNavegacao:int;"
+ "Navegador:Tripulante,capitao:String;"
+ "Deslocamento,#codigo:int,dtOrigem:String,dtDestino:String,"
+ "distancia:float,navio:Navio,localOrigem:Local,localDestino:Local;"
+ "Pirata:Navio,qtdCanhoes:int;"
+ "Guerra:Navio,qtdCanhoes:int;"
+ "Mercante:Navio,capacidade:int;"
+ "Mercadoria,#codigo:int,preco:float,peso:float;"
+ "PirataGuerra,#codigo:int,pirata:Pirata,guerra:Guerra;"
+ "PirataMercante,#codigo:int,pirata:Pirata,mercante:Mercante;"
+ "MercadoriaMercante,#codigo:int,mercante:Mercante,mercadoria:Mercadoria;"
+ "Mapa,#nome:String,qtdRota:int,escala:double,navegador:Navegador;"
+ "Navegacao:Mapa,oceano:String,costa:Costa;"
+ "Tesouro:Mapa,valorTesouro:float,ilha:Ilha");
daoservice.commit();
```

Figura 4.2: Código Java para registro de aplicação "Piratas do Caribe"
Fonte: O Autor

O apêndice A apresenta o log de saída no provedor de serviço para o registro da aplicação cliente "Piratas".

Em qualquer que seja a plataforma da aplicação cliente, com o registro da aplicação "Piratas", no servidor é criado um pacote "Piratas" contendo todas as classes de estrutura e serviços geradas para a aplicação cliente. A figura 4.3 apresenta o diagrama de classes do pacote "Piratas" no provedor de serviços após o registro de "Piratas do Caribe". Para ilustração, somente está representado a entidade "Costa". Em amarelo, estão as classes de modelo de negócio criadas, "Local" e sua classe filha "Costa". Em rosa, estão as classes de acoplamento com o *framework ObInject*, "EntityCosta" e "PrimaryKeyCosta". Em azul estão as classes de serviços CRUD, classe "Piratas" que contém os métodos de serviços CRUD para todas as entidades e a classe "PiratasRun" que herda de "Thread", responsável pela execução dos serviços.

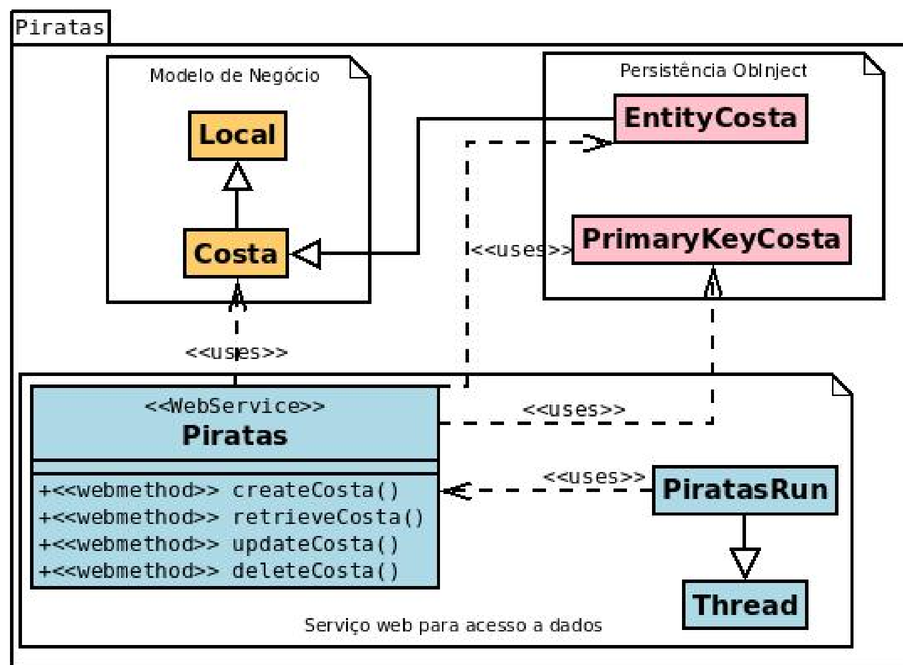

Figura 4.3: Diagrama de classes pacote “Piratas” representando a entidade Costa
Fonte: O Autor

Após o registro, os serviços CRUD para a aplicação “Piratas” estão disponíveis para acesso pela URL http://*hostname*:8080/Piratas?wsdl. A aplicação cliente pode também utilizar a ferramenta *wsimport* agora para importar os serviços CRUD, conforme exemplo abaixo:

```
wsimport -keep -s pastaaplicacao/src http://hostname:8080/Piratas?wsdl
```

Após a execução de *wsimport*, é criado na aplicação cliente um pacote “Piratas” contendo classes consumidoras dos serviços CRUD para a aplicação “Piratas”. Neste momento, é possível que a aplicação cliente execute os serviços CRUD para cada entidade definida no registro do modelo de negócio. A execução dos métodos CRUD para cada entidade segue como na execução dos métodos *begin, create e commit* . Como exemplo, será armazenado no servidor um objeto da entidade “Costa”. A entidade “Costa” herda de “Local”, neste caso, a ordem dos parâmetros do método “createCosta” inicia com os atributos de “Local” e posteriormente os atributos de “Costa”. Pretende-se armazenar um objeto “Costa” com código “1”, país “Brasil”, latitude “10”, longitude “10”, cidade “Salvador”, estado “Bahia”. As figuras 4.4a e 4.4b representam respectivamente a criação e recuperação de um objeto da entidade “Costa” em Java. A seguir é apresentado em JSON, o retorno da função *retrieveCosta* com código “1” passado por parâmetro.

```
{rows: [{"codigo":1,"cidade":"Salvador","estado":"Bahia","pais":"Brasil","lont
    ":10,"lat":10}]}
```

```java
Piratas piratas = new PiratasService().
        getPiratasPort();
piratas.createCosta(1, "Brasil",
        10, 10, "Salvador", "Bahia");
```
(a)

```java
Piratas piratas = new PiratasService().
        getPiratasPort();
System.out.println(piratas.retrieveCosta(1));
```
(b)

Figura 4.4: Armazenamento(a) e recuperação(b) de um objeto da entidade “Costa” em Java
Fonte: O Autor

4.2.2 Análise Funcional: Aplicação consumidora em C++

Em C++, a ferramenta *gsoap* (ENGELEN; GALLIVAN, 2002) executa a mesma funcionalidade de *wsimport* em Java. Ela cria uma infraestrutura local para consumir serviços web através do WSDL. Os comandos *wsdl2h* e *soapcpp2* são responsáveis por importar os serviços e compilar as classes consumidoras geradas.

Inicialmente, é necessário criar a infraestrutura cliente para executar os métodos *begin, create e commit*. O comando *wsdl2h* deve ser executado para importação dos serviços. No exemplo a seguir, a infraestrutura cliente para consumo dos serviços será gerada no arquivo “daoservice.h”:

```
wsdl2h -s -o daoservice.h http://hostname:8080/daoservice?wsdl
```

Após a geração do arquivo “daoservice.h”, é necessário executar o compilador *soapcpp2* para gerar os métodos e parâmetros para a aplicação cliente interagir com os serviços:

```
soapcpp2 -C daoservice.h
```

Após a importação e compilação das classes consumidoras dos serviços, deve-se executar os métodos *begin, create e commit* para registro da aplicação cliente “Piratas” (Apêndice B). Após o registro, a estrutura criada no servidor segue como mostrado na figura 4.3 como exemplificado na seção anterior. Assim, os serviços CRUD já estarão disponíveis para acesso pela URL http://*hostname*:8080/Piratas?wsdl. A aplicação cliente pode também utilizar as ferramentas *wsdl2h* e *soapcpp2* para importar os serviços CRUD.

```
wsdl2h -s -o piratas.h http://hostname:8080/Piratas?wsdl
soapcpp2 -C piratas.h
```

As figuras 4.5a e 4.5b representam respectivamente a criação e recuperação de um objeto da entidade “Costa” em C++. O retorno para a recuperação do objeto Costa com valor de chave 1 é um JSON como mostrado em Java (seção 4.2.1)

```cpp
#include "soapPiratasPortBindingProxy.h"
#include "PiratasPortBinding.nsmap"
using namespace std;
int main(int argc, char** argv) {
    PiratasPortBinding piratas;
    ns1__createCosta costa;
    costa.arg0 = 1;
    costa.arg1 = "Brasil";
    costa.arg2 = 10;
    costa.arg3 = 10;
    costa.arg4 = "Salvador";
    costa.arg5 = "Bahia";
    piratas.__ns1__createCosta(costa, NULL);
}
```

(a)

```cpp
#include "soapPiratasPortBindingProxy.h"
#include "PiratasPortBinding.nsmap"
using namespace std;
int main(int argc, char** argv) {
    PiratasPortBinding piratas;
    ns1__retrieveCosta costa;
    costa.arg0 = 1;
    ns1__retrieveCostaResponse response;
    piratas.__ns1__retrieveCosta(costa, response);
    cout << response.return_;
}
```

(b)

Figura 4.5: Armazenamento(a) e recuperação(b) de um objeto da entidade “Costa” em C++
Fonte: O Autor

4.2.3 Análise Funcional: Aplicação consumidora em PHP

Em PHP, a classe nativa da linguagem, *SoapClient*, é responsável por implementar clientes de serviços web. Diferente dos exemplos citados de Java e C++, em PHP não é necessário utilizar uma ferramenta para gerar classes clientes consumidoras. A classe *SoapClient*, a partir do endereço da WSDL, se encarrega de invocar os métodos web.

Inicialmente, é necessário executar os métodos *begin, create e commit* para registrar a aplicação cliente. O construtor da classe *SoapClient* recebe por parâmetro a URL do serviço. O método *__soapCall* da classe *SoapClient*, é reponsável por executar os métodos publicados na WSDL. O primeiro parâmetro de *__soapCall* é o nome do método e o segundo parâmetro é um *array* dos parâmetros do método. O apêndice C apresenta o código PHP que executa os métodos *begin, create e commit* para registro da aplicação cliente “Piratas”.

Após o registro, a estrutura criada no servidor segue como mostrado na figura 4.3 como mostrado anteriormente. Os serviços CRUD já estarão disponíveis para acesso pela URL http://*hostname*:8080/Piratas?wsdl. As figuras 4.6a e 4.6b representam respectivamente a criação e recuperação de um objeto da entidade “Costa” em PHP.

```php
<?php
$client = new
        SoapClient("http://hostname:8080/Piratas?wsdl");
$client->__soapCall("createCosta",
        array("parameters" => array("arg0" => "1",
        "arg1" => "Brasil", "arg2" => "10",
        "arg3" => "10", "arg4" => "Salvador",
        "arg5" => "Bahia"
)));
?>
```

(a)

```php
<?php
$client = new
        SoapClient("http://hostname:8080/Piratas?wsdl");
$response = $client->__soapCall("retrieveCosta",
        array("parameters"=>array("arg0" => "1")));
print_r($response);
?>
```

(b)

Figura 4.6: Armazenamento(a) e recuperação(b) de um objeto da entidade “Costa” em PHP
Fonte: O Autor

4.3 Análise de Desempenho

Esta seção descreve uma análise de desempenho para inserção de objetos da aplicação cliente do cenário "Piratas do Caribe" utilizando o serviço web para acesso a dados. Para efeito de comparação, foram efetuados experimentos realizando persistência através do *framework ObInject* e através do *framework Hibernate* com banco de dados MySQL. Para os experimentos com o *framework Hibernate*, foram criados serviços web CRUD para cada entidade de "Piratas do Caribe". As assinaturas dos métodos CRUD para armazenamento com o *framework Hibernate* foram idênticas às assinatura dos métodos CRUD para o *framework ObInject* conforme descritos na seção 3.1.1.

As análises foram realizadas com a persistência de um total de 44.000 objetos de "Piratas do Caribe". Todos os dados de objetos foram gerados automaticamente a partir de valores aleatórios e salvos em arquivos de texto. Os mesmos arquivos de dados foram utilizados nos experimentos no *framework ObInject* e no *framework HIbernate*.

A quantidade específica de objetos de cada entidade foram: Costa: 4.000 objetos; Ilha: 4.000 objetos; Navio de guerra: 200 objetos; Navio mercante: 200 objetos; Navio pirata: 200 objetos; Tripulante auxiliar: 10.000 objetos relacionados com algum navio; Tripulante navegador: 10.000 objetos relacionados com algum navio; Deslocamento: 6.000 objetos relacionados com algum navio e locais de origem e destino; Mercadoria: 2.500 objetos; Transporte de mercadorias entre navios mercantes: 2.500 objetos relacionados entre mercadorias e navios mercantes; Mapa de navegação: 2.000 objetos relacionados com algum navegador e alguma costa; Mapa de tesouro: 2.000 objetos relacionados com algum navegador e alguma ilha; Confrontos Piratas/Guerra: 200 objetos relacionados entre navios piratas e navios de guerra; Confrontos Piratas/Mercante: 200 objetos relacionados entre navios piratas e navios mercantes;

Os experimentos foram realizados com as aplicações consumidora e provedora de serviço executando na mesma máquina. Embora não seja uma restrição da tecnologia, isso foi necessário minimizar os impactos de rede sobre os valores medidos. Especificação da máquina: processador Intel Core 2 Duo T6400 2.0GHz x 2, memória RAM de 4.0GB, sistema operacional Ubuntu 12.04 LTS 64 bits. A linguagem de programação para aplicação consumidora foi Java, embora pudesse ser qualquer linguagem de programação que tenha suporte a tecnologia SOAP.

A aplicação provedora de serviço foi configurada para não persistir as conexões HTTP (*keepAlive false*) tanto na persistência com *HIbernate* quanto com o *framework ObInject*.

Foram realizadas análises de tempo de processamento para inserção de objetos e análises de consumo de espaço em disco para as duas abordagens: *framework ObInject* e *Hibernate*/MySQL. Os 44.000 objetos foram divididos em cinco experimentos de 8.800 objetos, 17.600 objetos, 26.400 objetos, 35.200 objetos e 44.000 objetos. As figuras 4.7 e 4.8 repre-

sentam os gráficos para análise de tempo de processamento de inserção e consumo de disco respectivamente.

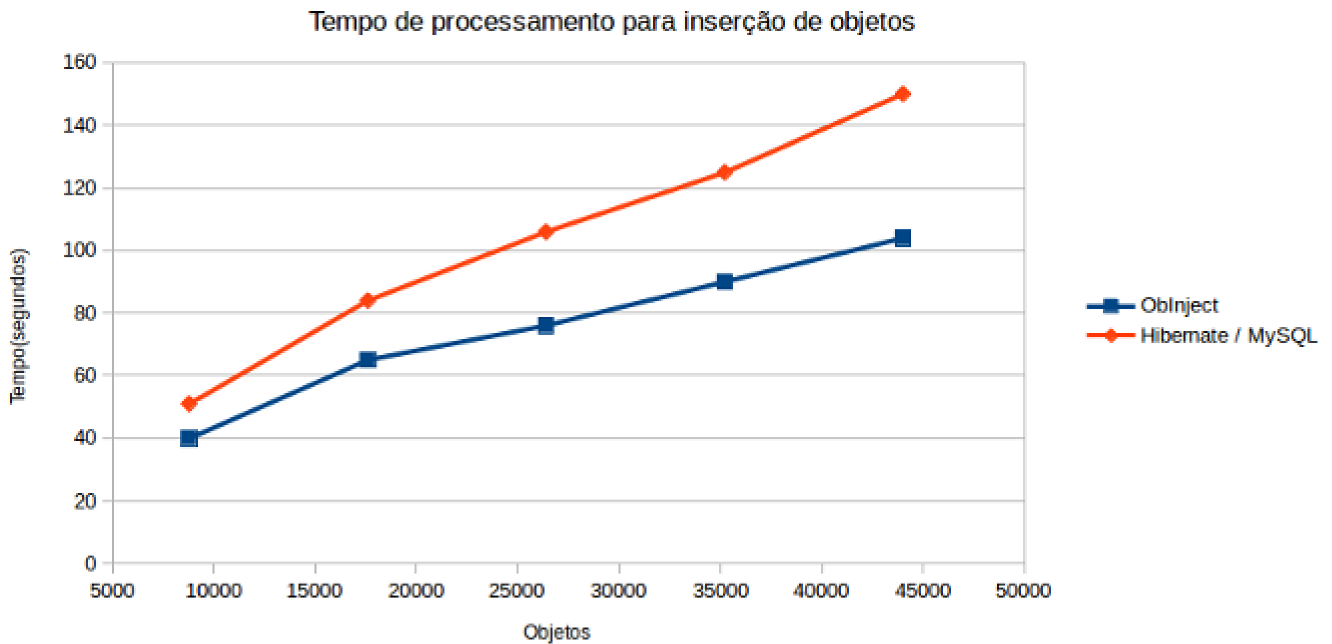

Figura 4.7: Gráfico de tempo de processamento para inserção de objetos
Fonte: O Autor

O desempenho para processamento de inserções de objetos utilizando o serviço web para acesso a dados pode ser observado na figura 4.7. Em média, o experimento utilizando o *framework ObInject* inseriu 330,46 objetos por segundo. O experimento utilizando o *Hibernate*/MySQL inseriu em média 241,21 objetos por segundo. Em todos as bases de teste, o *framework ObInject* foi mais rápido para persistir os objetos, sendo que: para a inserção de 8.800 objetos, o ganho foi de 27,5%; para a inserção de 17.600 objetos, o ganho foi de 29,23%; para a inserção de 26.400 objetos, o ganho foi de 39,47%; para a inserção de 35.200 objetos, o ganho foi de 38,89%; para a inserção de 44.000 objetos, o ganho foi de 44,23%.

Na figura 4.8 pode ser observado o consumo de espaço em disco para armazenamento dos objetos dos experimentos. O comportamento do gráfico mostra que o crescimento do espaço consumido utilizando o *framework ObInject* cresce linearmente, visto que a quantidade de objetos de cada experimento também é linear(crescimento de 8.800 objetos). Já o MySQL com 17.600 objetos apresentou um menor consumo de espaço em disco e praticamente a mesma quantidade de espaço com 26.400 objetos. A partir de 26.400 objetos, o MySQL consumiu mais espaço que o *framework ObInject*. O espaço utilizado para armazenar os objetos, comparando o *framework ObInject* com o MySQL tem-se que: para a inserção de 8.800 objetos, o ganho no *framework ObInject* foi de 22,88%; para a inserção de 17.600 objetos, o ganho no MySQL foi de 15,56%; para a inserção de 26.400 objetos, o ganho no

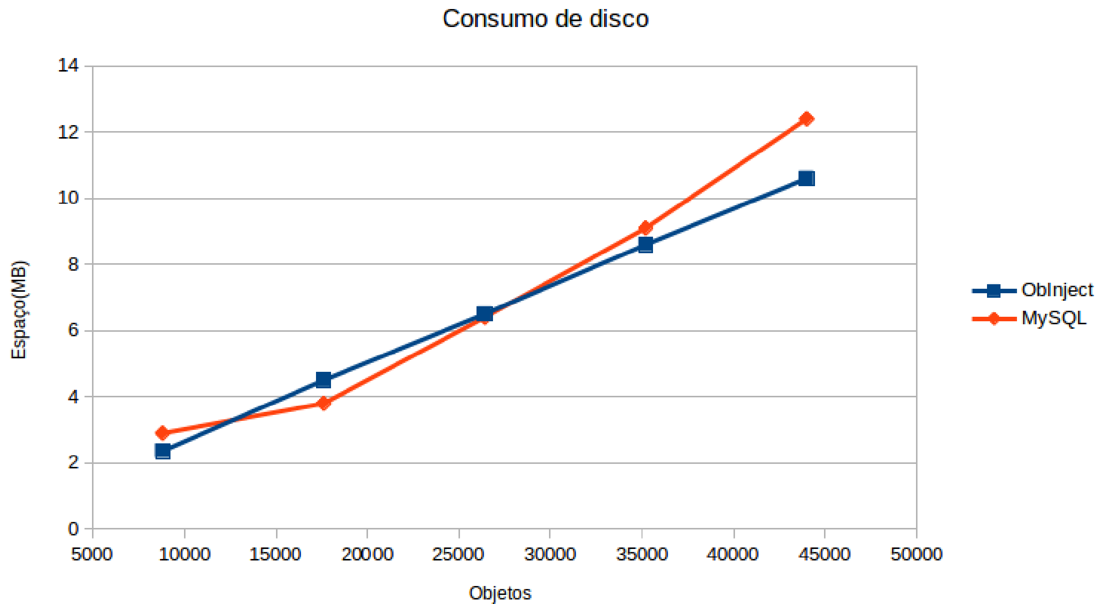

Figura 4.8: Gráfico de consumo de espaço em disco para inserção de objetos
Fonte: O Autor

MySQL foi de 1,54%; para a inserção de 35.200 objetos, o ganho no *framework ObInject* foi de 5,81%; para a inserção de 44.000 objetos, o ganho no *framework ObInject* foi de 16,98%.

4.4 Resultados

Os experimentos de análise funcional demonstraram a eficácia do serviço web de acesso a dados para registro de um modelo de negócio contendo heranças e associações. Através de seus próprios mecanismos de acesso a serviços web, as três linguagens de programação avaliadas conseguiram estabelecer conexão com os serviços e executá-los. Os serviços CRUD foram executados com sucesso nas três linguagens de programação. Os experimentos de análise de desempenho demonstraram que a técnica de serviços web CRUD pode ser utilizada em qualquer mecanismo de persistência, sendo que foram avaliados o *framework Obinject* e o *framework Hibernate*. A análise de tempo de inserção através dos mecanismos de serviços CRUD demonstrou que a persistência com o *framework Obinject* é mais rápida em relação ao *framework Hibernate* com MySQL. Isto se deve às técnicas de persistência de objetos implementadas pelo *framework Obinject*, pois não há uma camada extra de tradução de objetos em relações de bancos de dados, o que o torna mais rápido que o *framework Hibernate*. A análise de consumo de espaço em disco demonstrou que em quatro de cinco análises, o *framework Obinject* consumiu menos espaço que o MySQL. Esta redução no consumo de espaço também se deve aos mecanismos de estruturas de dados para persistência e indexação implementados pelo *framework Obinject*.

Capítulo 5

Conclusão

Este trabalho desenvolveu uma proposta baseada em serviços web para criação de um provedor de serviços de armazenamento e recuperação de dados para o *framework ObInject*. O provedor de serviços disponibiliza serviços CRUD *(Create, Retrieve, Update, Delete)* para acesso a dados de objetos de aplicações clientes. A partir da interoperabilidade que os serviços web oferecem, potenciais aplicações clientes escritas em qualquer linguagem de programação que desejem persistir dados, podem se tornar consumidoras do serviço de forma padronizada.

O principal resultado deste trabalho foi prover acesso remoto ao *framework ObInject* utilizando o conceito de serviço web que estabelece uma prestação de serviço a uma aplicação cliente.

Como contribuições secundárias deste trabalho, pode-se destacar:

- Criação de uma linguagem para definição de classes de negócio para persistência de dados no provedor de serviço.
- Novos modelos de negócios podem ser registrados a qualquer momento através de um serviço web *(begin, create, commit).*
- Criação e disponibilização de serviços CRUD *(Create, Retrieve, Update, Delete)* para armazenamento e recuperação de dados para cada entidade do modelo de negócios registrado.
- As análises funcionais realizadas nos experimentos demonstraram que estes serviços CRUD podem ser consumidos por aplicações desenvolvidas em qualquer linguagem de programação. Para tanto, as assinaturas dos serviços CRUD utilizam somente tipos de dados comuns a todas linguagens de programação.
- A análise de desempenho realizada no experimento demonstrou que a abordagem de serviço CRUD utilizada neste trabalho pode utilizar qualquer técnica de persistência,

sendo que foram avaliados o *framework Hibernate* com banco de dados MySQL e o *framework ObInject*. Estes experimentos demonstraram que a inserção de objetos no *framework ObInject* foi sempre mais rápida que o *framework Hibernate*.

Como propostas futuras podemos destacar os seguintes trabalhos:

- Definição de uma linguagem de consulta flexível e semelhante com SQL disponibilizada através de serviços web.
- Adicionar aos serviços funcionalidades de segurança e autenticação para estabelecer privilégios de acesso.
- Utilizar uma abordagem diferente de SOAP para definição de serviços web.

Referências Bibliográficas

ADL-TABATABAI, A.-R.; CIERNIAK, M.; LUEH, G.-Y.; PARIKH, V. M.; STICHNOTH, J. M. Fast, effective code generation in a just-in-time java compiler. In: ACM. **ACM SIGPlAN Notices**. [S.l.], 1998. v. 33, n. 5, p. 280–290.

ANTONIOLETTI, M.; KRAUSE, A.; PATON, N. W.; EISENBERG, A.; LAWS, S.; MALAIKA, S.; MELTON, J.; PEARSON, D. The ws-dai family of specifications for web service data access and integration. **ACM SIGMOD Record**, ACM, v. 35, n. 1, p. 48–55, 2006.

BARRY, D.; STANIENDA, T. Solving the java object storage problem. **Computer**, IEEE, v. 31, n. 11, p. 33–40, 1998.

BIZER, C.; HEATH, T.; BERNERS-LEE, T. Linked data-the story so far. **International journal on semantic web and information systems**, v. 5, n. 3, p. 1–22, 2009.

CARVALHO, L. O.; SERAPHIM, T. F.; JÚNIOR, C. T.; SERAPHIM, E. Obinject: a noodmg persistence and indexing framework for object injection. **Journal of Information and Data Management**, v. 4, n. 3, p. 220, 2013.

CHAMBERLIN, D. D.; BOYCE, R. F. Sequel: A structured english query language. In: ACM. **Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control**. New York, NY, USA, 1974. p. 249–264.

CIACCIA, P.; PATELLA, M.; ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In: **Proceedings of 23rd International Conference on Very Large Data Bases**. Athens, Greece: Morgan Kaufmann Publishers Inc., 1997. (VLBD '97), p. 426–435.

CODD, E. F. A relational model of data for large shared data banks. **Commun. ACM**, ACM, New York, NY, USA, v. 13, n. 6, p. 377–387, jun. 1970. ISSN 0001-0782.

COMER, D. The ubiquitous B-tree. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 11, n. 2, p. 121–137, jun. 1979.

CORRITORE, C. L.; WIEDENBECK, S. An exploratory study of program comprehension strategies of procedural and object-oriented programmers. **International Journal of Human-Computer Studies**, Elsevier, v. 54, n. 1, p. 1–23, 2001.

CROCKFORD, D. **Json**. [S.l.]: Technical report, RFC 4627, 2006, 2006. UR L http://www. ietf. org/rfc/rfc4627. txt, 2001.

ENGELEN, R. A. V.; GALLIVAN, K. A. The gsoap toolkit for web services and peer-to-peer computing networks. In: IEEE. **Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on**. Berlin, Germany, 2002. p. 128–128.

ERL, T. **Service-Oriented Architecture: Concepts, Technology, and Design**. 1. ed. Upper Saddle River, NJ, USA: Prentice Hall, 2005. 792 p.

FORMAN, I. R.; FORMAN, N.; IBM, J. V. Java reflection in action. Citeseer, 2004.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **Database Systems: The Complete Book**. 2nd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. 1241 p.

GURUGE, A. **Web Services Theory and Practice**. 1. ed. [S.l.]: Digital Press, 2004. 371 p.

GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. **ACM SIGMOD Record**, ACM, New York, NY, USA, v. 14, n. 2, p. 47–57, jun. 1984.

HANSEN, M.; MADNICK, S.; SIEGEL, M. **Data integration using web services**. [S.l.]: Springer, 2003.

HOU, D.; RUPAKHETI, C. R.; HOOVER, H. J. Documenting and evaluating scattered concerns for framework usability: A case study. In: IEEE. **Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific**. Beijing, China, 2008. p. 213–220.

ISO/IEC. Norm 11578, **Information technology - Open Systems Interconnection - Remote Procedure Call (RPC)**. Genève, Switzerland: International Organization for Standardization / International Electrotechnical Commission, 1996.

ITU. Recommendation, **OSI networking and system aspects - Naming, Addressing and Registration**. Genève, Switzerland: International Telecommunication Union, set. 2004. (Series X: Data Networks And Open System Communications).

JONES, J.; KUHN, W.; KESSLER, C.; SCHEIDER, S. Making the web of data available via web feature services. In: **Connecting a Digital Europe Through Location and Place**. [S.l.]: Springer, 2014. p. 341–361.

JOSUTTIS, N. M. **SOA in Practice – The Art of Distributed Systems Design**. 1. ed. Sebastopol, CA, USA: O'Reilly, 2007. 342 p.

KEITH, M.; SCHINCARIOL, M. **Pro JPA 2: Mastering the Java™ Persistence API**. 1. ed. [S.l.]: Apress, 2009.

KOHLERT, D.; GUPTA, A. The java api for xml-based web services (jax-ws) 2.1. **JCP Specification, Sun Microsystems**, 2007.

KREGER, H. et al. Web services conceptual architecture (wsca 1.0). **IBM Software Group**, v. 5, p. 6–7, 2001.

LEACH, P.; MEALLING, M.; SALZ, R. **A Universally Unique Identifier (UUID) URN Namespace**. Internet Engineering Task Force - IETF, 2005. RFC 4122. Disponível em: <`http://www.ietf.org/rfc/rfc4122.txt`>. Acesso em: 25 de maio de 2012.

LIU, X.; HU, C.; LI, Y.; JIA, L. The advanced data service architecture for modern enterprise information system. In: IEEE. **Information Science and Applications (ICISA), 2014 International Conference on**. Seoul, Korea, 2014. p. 1–4.

MARTIN, J. Managing the data base environment. Savant Res. Inst., 1981.

MICROSYSTEMS, S. **Core J2EE Patterns - Data Access Object**. 2001. `http://www.oracle.com/technetwork/java/dataaccessobject-138824.html`. Accessed: 2014-09-16.

OLIVEIRA, M. F. d. **Metaprogramação e Metadados de Persistência e Indexação para o Framework Object-Injection**. Dissertação (Mestrado) — Universidade Federal de Itajubá, UNIFEI, 2012.

O'REILLY, T. What is web 2.0: Design patterns and business models for the next generation of software. **Communications and Strategies**, v. 65, n. 1, p. 17–37, 2007.

POMBINHO, J.; AVEIRO, D.; TRIBOLET, J. Business service definition in enterprise engineering-a value-oriented approach. In: IEEE. **Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2012 IEEE 16th International**. [S.l.], 2012. p. 70–79.

PRETTY, D.; BLACKWELL, B. H1ds: A new web-based data access system. **Fusion Engineering and Design**, Elsevier, 2014.

PULIER, H. T. E. **Understanding Enterprise SOA**. 1. ed. Greenwich, CT, USA: Manning, 2006. 280 p.

RAJESH, S.; SWAPNA, S.; REDDY, P. S. Data as a service (daas) in cloud computing. **Global Journal of Computer Science and Technology**, v. 12, n. 11-B, 2012.

RAMAKRISHNAN, R.; GEHRKE, J. **Database Management Systems**. 2nd. ed. USA: McGraw-Hill, 1999. 931 p.

RAMAN, V.; NARANG, I.; CRONE, C.; HAAS, L.; MALAIKA, S.; MUKAI, T.; WOLFSON, D.; BARU, C. Data access and management services on grid. **GGF, DAIS group**, 2002.

RICHARDSON, L.; RUBY, S. **RESTful web services**. [S.l.]: "O'Reilly Media, Inc.", 2008.

SIGNORE, R.; STEGMAN, M. O.; CREAMER, J. **The ODBC solution: Open database connectivity in distributed environments**. [S.l.]: McGraw-Hill, Inc., 1995.

STILLERMAN, J.; FREDIAN, T.; KLARE, K.; MANDUCHI, G. Mdsplus data acquisition system. **Review of Scientific Instruments**, AIP Publishing, v. 68, n. 1, p. 939–942, 1997.

TALIA, D. The open grid services architecture: where the grid meets the web. **IEEE Internet Computing**, IEEE Computer Society, v. 6, n. 6, p. 67–71, 2002.

TSAI, W.; BAI, X.; HUANG, Y. Software-as-a-service (saas): perspectives and challenges. **Science China Information Sciences**, Springer, v. 57, n. 5, p. 1–15, 2014.

W3C. **Web Services Architecture**. 2004. http://www.w3.org/TR/ws-arch/. Accessed: 2014-09-26.

W3SCHOOLS. **SOAP Reference**. [S.l.], 2006. 33 p. Http://www.w3schools.com/soap/default.asp.

WANGUE, R.; ZHAO, F. A method to secure data in web database with web services. In: ATLANTIS PRESS. **International Conference on Logistics Engineering, Management and Computer Science (LEMCS 2014)**. [S.l.], 2014.

ZAHN, L.; DINEEN, T.; LEACH, P. **Network computing architecture**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990. 224 p.

ZEZULA, P.; AMATO, G.; DOHNAL, V.; BATKO, M. **Similarity Search: The Metric Space Approach**. 2007. Free Course Slides. Disponível em: <http://www.nmis.isti.cnr.it/amato/similarity-search-book/slides/>. Acesso em: 11 de fevereiro de 2012.

Apêndice A

Exemplo de log no provedor de serviços

```
Definindo novo serviço de armazenamento de dados:
Nome: PIRATAS
Classes para persistência:
-> Local
   -> Gerando webmethods para acesso à dados...
-> Costa
   -> Gerando webmethods para acesso à dados...
-> Ilha
   -> Gerando webmethods para acesso à dados...
-> Navio
   -> Gerando webmethods para acesso à dados...
-> Tripulante
   -> Gerando webmethods para acesso à dados...
-> Auxiliar
   -> Gerando webmethods para acesso à dados...
-> Navegador
   -> Gerando webmethods para acesso à dados...
-> Deslocamento
   -> Gerando webmethods para acesso à dados...
-> Pirata
   -> Gerando webmethods para acesso à dados...
-> Guerra
   -> Gerando webmethods para acesso à dados...
-> Mercante
   -> Gerando webmethods para acesso à dados...
-> Mercadoria
   -> Gerando webmethods para acesso à dados...
-> PirataGuerra
   -> Gerando webmethods para acesso à dados...
-> PirataMercante
   -> Gerando webmethods para acesso à dados...
-> MercadoriaMercante
   -> Gerando webmethods para acesso à dados...
-> Mapa
   -> Gerando webmethods para acesso à dados...
-> Navegacao
   -> Gerando webmethods para acesso à dados...
-> Tesouro
   -> Gerando webmethods para acesso à dados...
Classes finalizado.
Gerando Thread de Endpoint para nova interface de terminal de serviço...
Compilando classes do cliente...
Compilando classes de ligação com o framework ObInject...
Compilando classes de webservices...
Executando Thread de novo Endpoint de terminal de serviço...
SERVIÇO DISPONÍVEL.
```

Figura A.1: Log de saída para registro da aplicação "Piratas do caribe"
Fonte: O Autor

Apêndice B

Código C++ para registro da aplicação “Piratas”

```
#include "soapDAOServicePortBindingProxy.h"
#include "DAOServicePortBinding.nsmap"
using namespace std;
int main(int argc, char** argv) {
    DAOServicePortBinding daoservice;
    ns1__begin begin;
    begin.arg0 = "Piratas";
    daoservice.__ns1__begin(begin, NULL);
    ns1__create create;
    create.arg0 = "Local,#codigo:int,pais:String,lat:double,lont:double;"
    + "Costa:Local,cidade:String,estado:String;"
    + "Ilha:Local,nome:String,area:float;"
    + "Navio,#nome:String,construido:String,tamanho:int,qtdMastro:int,qtdVelas:int;"
    + "Tripulante,#nome:String,nacionalidade:String,apelido:String,navio:Navio;"
    + "Auxiliar:Tripulante,horasNavegacao:int;"
    + "Navegador:Tripulante,capitao:String;"
    + "Deslocamento,#codigo:int,dtOrigem:String,dtDestino:String,"
    + "distancia:float,navio:Navio,localOrigem:Local,localDestino:Local;"
    + "Pirata:Navio,qtdCanhoes:int;"
    + "Guerra:Navio,qtdCanhoes:int;"
    + "Mercante:Navio,capacidade:int;"
    + "Mercadoria,#codigo:int,preco:float,peso:float;"
    + "PirataGuerra,#codigo:int,pirata:Pirata,guerra:Guerra;"
    + "PirataMercante,#codigo:int,pirata:Pirata,mercante:Mercante;"
    + "MercadoriaMercante,#codigo:int,mercante:Mercante,mercadoria:Mercadoria;"
    + "Mapa,#nome:String,qtdRota:int,escala:double,navegador:Navegador;"
    + "Navegacao:Mapa,oceano:String,costa:Costa;"
    + "Tesouro:Mapa,valorTesouro:float,ilha:Ilha";
    daoservice.__ns1__create(create, NULL);
    daoservice.__ns1__commit(NULL, NULL);
}
```

Figura B.1: Código C++ para registro de aplicação “Piratas do Caribe”

Fonte: O Autor

Apêndice C

Código PHP para registro da aplicação “Piratas”

```
<?php
$modelClient = "Local,#codigo:int,pais:String,lat:double,lont:double;"
. "Costa:Local,cidade:String,estado:String;"
. "Ilha:Local,nome:String,area:float;"
. "Navio,#nome:String,construido:String,tamanho:int,qtdMastro:int,qtdVelas:int;"
. "Tripulante,#nome:String,nacionalidade:String,apelido:String,navio:Navio;"
. "Auxiliar:Tripulante,horasNavegacao:int;"
. "Navegador:Tripulante,capitao:String;"
. "Deslocamento,#codigo:int,dtOrigem:String,dtDestino:String,distancia:float,"
        . "navio:Navio,localOrigem:Local,localDestino:Local;"
. "Pirata:Navio,qtdCanhoes:int;"
. "Guerra:Navio,qtdCanhoes:int;"
. "Mercante:Navio,capacidade:int;"
. "Mercadoria,#codigo:int,preco:float,peso:float;"
. "PirataGuerra,#codigo:int,pirata:Pirata,guerra:Guerra;"
. "PirataMercante,#codigo:int,pirata:Pirata,mercante:Mercante;"
. "MercadoriaMercante,#codigo:int,mercante:Mercante,mercadoria:Mercadoria;"
. "Mapa,#nome:String,qtdRota:int,escala:double,navegador:Navegador;"
. "Navegacao:Mapa,oceano:String,costa:Costa;"
. "Tesouro:Mapa,valorTesouro:float,ilha:Ilha";
$client = new SoapClient("http://hostname:8080/daoservice?wsdl");
$client->__soapCall("begin", array("parameters" => array("arg0" => "Piratas")));
$client->__soapCall("create", array("parameters" => array("arg0" => $modelClient)));
$client->__soapCall("commit", null);
?>
```

Figura C.1: Código PHP para registro de aplicação “Piratas do Caribe”
Fonte: O Autor