UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

RODRIGO SCHEFFER LUMERTZ

Um Framework para Marts de Serviços Web Compostos

Trabalho de Graduação.

Prof. Dr. Leandro Krug Wives
Orientador

Porto Alegre, julho de 2010.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
BPEL	Business Process Execution Language
CWS	Composite Web Service
DM	Data Mart
DW	Data Warehouse
GUI	Graphical User Interface
HTML	HyperText Markup Language
MVC	Model-View-Controller
OLAP	On-Line Analytical Processing
SOA	Software Oriented Architecture
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WB	Web Service
WSDL	Web Services Description Language
XML	Extensible Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Serviços Web compostos são projetados para satisfazer requisições de usuários quando nenhum serviço Web individualmente é capaz de fazê-lo. Através de uma linguagem de composição, os serviços Web existentes são combinados gerando um novo serviço Web composto.

Este trabalho aborda o projeto e o desenvolvimento de um *framework* para *CWSMarts*. Estes são uma estrutura especialmente dedicada a serviços Web compostos. Eles são formados por um repositório, um *data mart*, e um conjunto de módulos capazes de prover operações no nível de composição. O conteúdo de um *CWSMart* é dinâmico: novos serviços Web compostos são adicionados e outros são removidos, serviços são executados, monitorados e os dados coletados podem ser analisados. Através do *framework* proposto é possível desenvolver *CWSMarts* implementando apenas algoritmos específicos para determinada área, em um tempo relativamente curto.

Palavras-Chave: serviços Web, serviços Web compostos, BPEL, *CWSMarts*, *framework*

A Framework for Marts of Composite Web Services

ABSTRACT

Composite Web services are designed to target users' requests that cannot be satisfied by any single, available Web service. Through a composition language, a composite Web service can be obtained by combining available Web services.

This work discusses the design and development of a framework for *CWSMarts*. These are structures specifically dedicated to composite Web services. They are composed by a repository, a data mart and a set of modules that provides operations at the composition level. The contents of the *CWSMarts* are expected to be dynamic: new composite Web services are added, others are expelled, services are deployed, monitored and the collected data can be analyzed. Through the proposed framework is possible to develop *CWSMarts* by implementing only specific algorithms in a relative short time.

Keywords: Web services, composite Web services, BPEL, *CWSMarts*, framework

1 INTRODUÇÃO

Está em andamento no Instituto de Informática um projeto denominado "Dodona: recomendação de serviços Web" (WIVES, 2008). Tal projeto é coordenado pelo prof. Leandro Krug Wives, e consiste em estudar como agregar técnicas de Sistemas de Recomendação com serviços Web. Uma dimensão importante do projeto é o estudo de Serviços Web Compostos. Serviços Web Compostos ou, do inglês, *Composite Web Services* (CWS) podem ser vistos como um grupo ou conjunto de serviços Web que são agregados para formar um processo de negócio (ZUROWSKA; DETERS, 2009).

Os CWS podem ser armazenados em *CWSMarts*, que são centros dedicados que contém serviços Web compostos, com funcionalidades similares, prontos para uso, combinados e empacotados, e que podem ser facilmente utilizados a qualquer momento (MAAMAR et al., 2010). *CWSMarts* baseiam-se nos conceitos de comunidade de serviços Web e de *data marts*. Comunidades de serviços Web agrupam serviços Web com funcionalidades específicas independentemente de como eles funcionam internamente, onde estão localizados, como são mantidos ou quem os provê. Um *CWSMart* eleva os conceitos e operações de comunidades para o nível de composição onde serviços Web componentes são substituídos por serviços Web compostos. Como resultado, serviços Web compostos são agrupados em "mercados" independentemente de quem os desenvolveu, como eles funcionam e quais componentes eles usam. *Data marts* ajudam na tomada de decisões e desenvolvimento de estratégias ao permitir a análise de conjuntos de dados que representam tendências e situações passadas. *CWSMarts* aplicam o mesmo conceito ao analisar dados do funcionamento e execução de serviços Web compostos para melhor gerenciá-los e recomendá-los ao usuário.

No projeto "Dodona" não existe uma prova de conceito para a ideia de *CWSMarts*. Este trabalho contribui na medida em que fornece um *framework* a partir do qual *CWSMarts* podem serem desenvolvidos para áreas específicas, tornando essa prova de conceito possível. Adicionalmente, foram propostas algumas extensões e refinamentos a ideia inicial.

O trabalho desenvolvido tem por objetivo especificar e construir um *framework* que permita a definição de *CWSMarts* específicos. O *framework* especifica módulos e funcionalidades básicas-padrão, que podem ser estendidas. Tal *framework* é uma importante contribuição para a área, servindo como base para trabalhos futuros na área de *CWSMarts* e para implementação de *CWSMarts* para áreas determinadas.

O trabalho envolveu o estudo do estado da arte na área de *CWSMarts* bem como estudo sobre serviços Web compostos e conceitos relacionados. A partir disso, foi feito o projeto do *framework* de modo a integrar cada um dos módulos que compõem o *CWSMart*, permitir módulos customizáveis e extensões para suporte as linguagens de composição de serviços Web existentes e mesmo futuras. Foi utilizado projeto orientado

a objetos e diversos padrões de projeto, a saber, Singleton, Factory Methods, Abstract Factory e Strategy (GAMMA et. al, 2000). A partir do projeto, foi desenvolvida uma implementação do *framework* em linguagem de programação Java e banco de dados PostgreSQL, com auxilio da ferramenta de modelagem de projetos Enterprise Architect.

A estrutura do texto está organizada da seguinte forma. A Introdução visa dar ao leitor uma ideia geral do trabalho desenvolvido, contextualizando-o no projeto denominado Dodona, coordenado pelo Professor Leandro Krug Wives e em andamento no Instituto de Informática da UFRGS. O capítulo 2 tem o objetivo de fazer uma breve descrição dos conceitos que estão relacionados com o desenvolvimento do *framework* para o *mart* de serviços Web compostos. Os principais conceitos contam com exemplos práticos. O capítulo 3 mostra alguns trabalhos relacionados a *framework*s, serviços Web e serviços Web compostos, indicando o aspecto inovador da proposta de *marts* de serviços Web compostos. O capítulo 4, Framework para Marts de Serviços Web Compostos, apresenta os detalhes da implementação do trabalho. Além disso, em especial, nesse capítulo, destaca-se como o *framework* deve ser utilizado e é apresentado um *CWSMart* exemplo desenvolvido utilizando-se o *framework*. O capitulo 5 apresenta as conclusões do trabalho, resultados alcançados, dificuldades encontradas assim como as possibilidades de extensões em trabalhos futuros.

2 CONCEITOS RELACIONADOS

Este capítulo tem o objetivo de fazer uma breve descrição dos conceitos que estão relacionados com o desenvolvimento do *framework* para o *mart* de serviços Web compostos.

2.1 Arquitetura Orientada a Serviços

Arquitetura Orientada a Serviços, originado do inglês SOA (*Software Oriented Architecture*), é uma solução lógica para o projeto de software com objetivo de prover serviços tanto para usuários finais quanto para outros serviços distribuídos em uma rede, via interfaces publicáveis e pesquisáveis (PAPAZOGLOU, 2008). É uma evolução do paradigma de projeto de computação distribuída baseada em requisições e respostas (KODALI, 2005). Nele, uma lógica de negócio ou funções individuais são modularizadas e disponibilizadas como um serviço. A chave desses serviços é a sua natureza fracamente acoplada, dado que a interface que define o serviço deve ser independente da implementação. Desenvolvedores de aplicações ou integradores podem construir suas aplicações compondo um ou mais serviços sem conhecer sua implementação. O objetivo essencial de uma aplicação SOA é possibilitar interoperabilidade entre tecnologias existentes e futuras.

São características de serviços SOA:

- possuem uma interface autodescritiva descrita em documentos XML independentes de plataforma. *Web Services Description Language* (WSDL) é o padrão atualmente usado para descrever os serviços.
- se comunicam através de mensagens formalmente definidas via XML Schema (também chamados XSD). Comunicação entre clientes e provedores ou serviços tipicamente ocorre em ambientes heterogêneos com pouco conhecimento sobre os provedores.
- são mantidos por um registro, que age como um diretório. Aplicações podem procurar os serviços no registro e invocá-los. *Universal Description, Definition and Integration* UDDI é o padrão usado para registro de serviços.

Cada serviço SOA tem um padrão de qualidade associado. Alguns dos principais atributos são requerimentos de segurança, como autenticação e autorização, confiabilidade nas mensagens trocadas, e regras sobre quem pode invocar os serviços.

2.2 Serviços Web

Um Serviço Web, em inglês *Web Service*, é uma aplicação independe de plataforma, fracamente acoplada, autocontida, habilitada para Web que pode ser descrita, publicada,

descoberta, coordenada e configurada usando artefatos XML (padrões abertos) com o objetivo de desenvolver aplicações distribuídas interoperáveis (PAPAZOGLOU, 2008). Web services possuem a habilidade de usar outros Web services em uma computação comum com objetivo de completar uma tarefa concreta, conduzir uma operação de negócio ou resolver um problema complexo.

2.2.1 SOA *versus* Web services

Web services são serviços implementados usando um conjunto de padrões enquanto que SOA é um padrão arquitetural. Web service é uma das maneiras de implementar SOA. Os benefícios de implementar SOA com Web services é que se atinge o acesso a serviços de uma maneira independente de plataforma e melhor interoperabilidade a medida que mais e mais provedores desenvolvem Web services como meio de disponibilização de seus serviços (KODALI, 2005).

2.3 WSDL

WSDL (Web Services Description Language) é uma linguagem baseada em XML usada para descrever um serviço Web (que segue o padrão SOA), especificar como acessá-lo e indicar suas operações disponíveis (W3C WSDL, 2001).

A versão mais utilizada na publicação de serviços Web é a WSDL 1.1, entretanto, a W3C já publicou uma atualização para expandir o seu uso que é a WSDL 2.0 (W3C WSDL 2.0, 2007). As mudanças foram substanciais a ponto de gerar o *wsdl* 2.0 e não o *wsdl* 1.2 como originalmente estava previsto. A figura 2.1 apresenta uma representação dos conceitos definidos nos documentos WSDL 1.1 e WSDL 2.0 comparando as duas versões. O elemento *port* foi renomado para *endpoint*, o mesmo ocorreu com o elemento *portType* que foi renomeado para *interface*. O elemento *message* foi removido e passa a ser definido por tipos *XML Schema*.

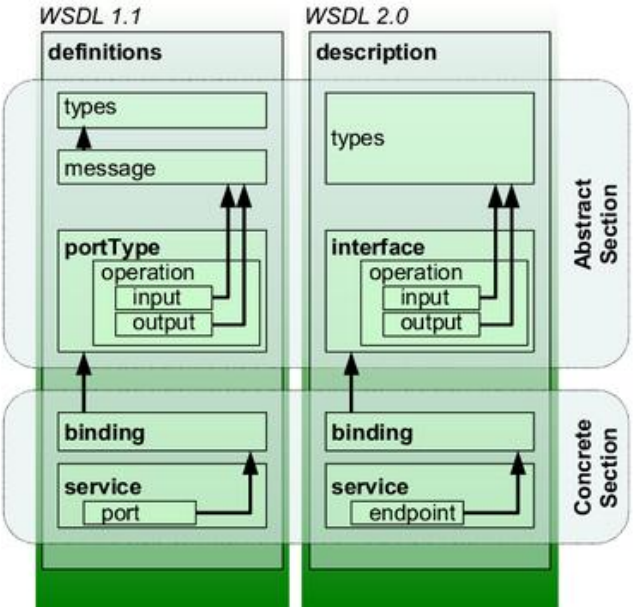

Figura 2.1: Representação dos conceitos definidos pelos documentos WSDL 1.1 e WSDL 2.0 (WIKIPEDIA WSDL)

No WSDL, a definição abstrata das operações e mensagens é separada das partes de uso concreto, ou instâncias, possibilitando o reuso das definições. Os principais conceitos são assim definidos:

- **Service** é um conceito que pode ser visto como um *container* para um conjunto de operações que foram expostas para protocolos baseados na web.
- **Port/Endpoint** define um endereço ou ponto de conexão para um serviço web. Tipicamente é representado por uma string com a URL HTTP.
- **Binding** especifica a interface, define o estilo de vinculação SOAP (RPC/ Document) e o transporte (protocolo SOAP). Essa seção também define as operações.
- **Portype**/**Interface** define um serviço web, as operações que podem ser executadas e as mensagens que são usadas para executar as operações. Na versão do WSDL 2.0 ocorreu a renomeação da *tag* <portType> para <interface>.
- **Operation** define as operações disponíveis. Cada operação pode ser comparada a um método ou função em uma linguagem de programação tradicional.
- **Message** é tipicamente correspondente à uma operação. O conceito foi removido do WSDL 2.0 onde simplesmente e diretamente se refere a tipos XML Schema para definir tais construções.
- **Type** tem o objetivo de descrever os dados. XML Schema é usado para esse propósito.

Para auxiliar na explicação e exemplificar os elementos mais importantes da gramática XML WSDL, será definido o serviço Web *EmployeeService* que disponibiliza a operação *EmployeeTravelStatus* cuja função é retornar a classe de viagem de um funcionário dado as informações desse funcionário. No texto que segue será apresentado trechos do documento *Employee.wsdl* (JURIC, s.d.) que descreve o serviço citado.

O elemento *definitions* é o elemento raíz de todos documentos WSDL. Ele define o nome do serviço e declara múltiplos *namespaces* usados no restante do documento:

```
<definitions name=”EmployeeService”
        xmlns:tns="http://packtpub.com/service/employee/"
        targetNamespace="http://packtpub.com/service/employee/"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

O elemento *types* descreve todos os tipos de dados usados entre o cliente e o servidor. Para a operação *EmployeeTravelStatus* foi definido dois tipos de dados, o tipo *EmployeeType* que contém três atributos do tipo string e define um funcionário, e o tipo *TravelClassType* que possui um atributo do tipo string que pode ser um dos valores da enumeração *Economy*, *Business* ou *First* e define a classe de viagem de um funcionário:

```
<types>
  <xs:complexType name="EmployeeType">
    <xs:sequence>
      <xs:element name="FirstName" type="xs:string" />
```

```
    <xs:element name="LastName" type="xs:string" />
    <xs:element name="Departement" type="xs:string" />
   </xs:sequence>
  </xs:complexType>

  <xs:simpleType name="TravelClassType">
   <xs:restriction base="xs:string">
    <xs:enumeration value="Economy"/>
    <xs:enumeration value="Business"/>
    <xs:enumeration value="First"/>
   </xs:restriction>
  </xs:simpleType>
</types>
```

O elemento *message* descreve uma mensagem de um sentido. Ele define o nome da mensagem e contém zero ou mais elementos *parts*. Cada elemento *part* é definido por um nome e por um tipo. A operação *EmployeeTravelStatus* recebe a mensagem *EmployeeTravelStatusRequestMessage* com um único elemento *part* de nome *employee* e tipo *EmployeeType* definido anteriormente e retorna a mensagem *EmployeeTravelStatusResponseMessage* composta de um elemento *part* de nome *travelClass* e de tipo *TravelClassType*.

```
<message name="EmployeeTravelStatusRequestMessage">
 <part name="employee" type="tns:EmployeeType" />
</message>

<message name="EmployeeTravelStatusResponseMessage">
 <part name="travelClass" type="tns:TravelClassType" />
</message>
```

O elemento *portType* tem um nome e define uma ou mais operações que estarão disponíveis através do serviço Web. A operação *EmployeeTravelStatus* é definida usando-se como entrada e saida as mensagens definidas anteriormente:

```
<portType name="EmployeeTravelStatusPT">
 <operation name="EmployeeTravelStatus">
  <input message="tns:EmployeeTravelStatusRequestMessage" />
  <output message="tns:EmployeeTravelStatusResponseMessage" />
 </operation>
</portType>
```

2.4 SOAP

SOAP é um protocolo baseado em XML utilizado para a troca de informações estruturadas entre sistemas distribuídos (W3C SOAP, 2007).

As mensagens SOAP podem trafegar sobre diversos protocolos. Por ser transportado também pelo protocolo HTTP, o SOAP é utilizado facilmente para comunicação através de proxy e firewall. Com esta tecnologia, é possível que novas aplicações possam interagir com as já existentes, independente de plataforma ou linguagem. Esses serviços são componentes que permitem enviar e receber dados em formato XML.

Uma vez que o cliente possui a descrição do serviço Web (WSDL), ele pode, então, invocar esse serviço usando SOAP.

2.5 UDDI

UDDI é um protocolo que especifica uma forma para publicar e descobrir serviços na Internet. Um serviço de registro UDDI é um Web Service que gerencia informação sobre provedores, implementações e meta-dados de serviços (OASIS, s.d.).

O serviço UDDI foi projetado para ser consultado através de mensagens SOAP e retornar documentos WSDL que descrevem o modo como se deve interagir com os serviços Web contidos no diretório. Dessa forma, provedores de serviços podem utilizar UDDI para publicar os serviços que eles oferecem, usuários de serviços podem usar UDDI para descobrir serviços que lhes interessem e obter os meta-dados necessários para utilizar esses serviços.

As operações suportadas pelo diretório UDDI são publicação (*publish*), eliminação (*delete*) e pesquisa (*query*). As pesquisas podem ser de três tipos: por área de negócio do serviço (componente chamado de páginas amarelas), por contatos das empresas (componente páginas brancas) e por endereços de acesso aos serviços (componente páginas verdes). A informação guardada no diretório UDDI é estruturada como na figura 2.2. A entidade principal do esquema é a organização. Uma organização pode ter pessoas de contato. Uma organização pode ter serviços. Os serviços podem ter implementações (*bindings*). Tanto as organizações quanto os serviços podem ter classificações que podem ser industriais, regionais ou uma classificação personalizada.

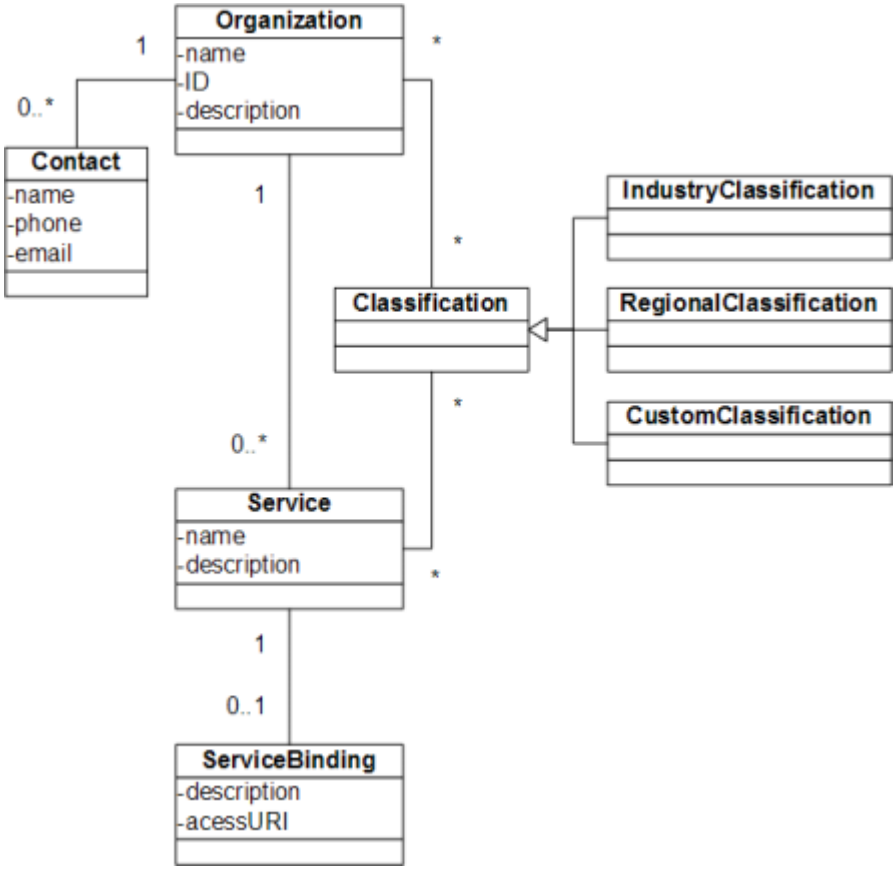

Figura 2.2: Esquema de dados de um registro UDDI (Docentes de Sistemas Distribuídos, DEI, IST, UTL)

Como será visto mais adiante, uma das funcionalidades do *CWSMart* é gerenciar um repositório de serviços, permitindo adição de novos serviços e a consulta de serviços

contidos na base, o que se assemelha as funcionalidades do UDDI. No entanto, os diretórios UDDI não foram planejados para serviços Web compostos e embora, em teoria, eles pudessem ser usados para esse fim uma vez que, na visão do cliente, um serviço Web composto não deixa de ser um serviço Web, na prática, se optou pelo projeto de um repositório especifico para serviços Web compostos, capaz de gerenciar uma série de informações extras necessárias para provimento das funcionalidades previstas pelo *mart*.

2.6 Comunidade de Web Services

Na área de Web services, Benatallah et al. (2003) definem comunidade como uma coleção de Web services com uma funcionalidade comum, embora estes Web services possuam propriedades não funcionais distintas. Medjahed e Bouguettaya (2005) consideram comunidade como um meio de organizar Web services que compartilham um mesmo domínio de interesse com respeito a uma ontologia. Finalmente, Maamar et al. (2007) definem comunidade através de um Web service abstrato representativo que conduz à comunidade, sem se referir explicitamente aos Web services concretos que populam essa comunidade e implementam essa funcionalidade em tempo de execução.

Um *CWSMart* eleva os conceitos e operações de comunidades para o nível de composição onde serviços Web componentes são substituídos por serviços Web compostos. Como resultado, serviços Web compostos com funcionalidades similares são agrupados em *marts* independentemente de quem os desenvolveu, como eles funcionam internamente, como tratam exceções, quais serviços componentes eles usam, etc.

2.7 Processos de Negócios

Um processo pode ser definido como qualquer sequência de passos que é iniciado por um evento, transforma informações, materiais, ou obrigações e produz uma saída (HARMON, 2003a). Um processo de negócio, do inglês *business process*, é um conjunto de tarefas relacionadas executadas para atingir um resultado bem definido (PAPAZOGLOU, 2008). Ele define os resultados a ser atingidos, o contexto das atividades, as relações entre as atividades e as interações com outros processos ou recursos. Um processo de negócio pode receber eventos que alteram o estado do processo e as atividades subsequentes. Um processo de negócio pode produzir eventos que servirão de entrada para outra aplicação ou processo. Pode também invocar outros processos para executar funções computacionais e pode também requisitar ações executadas por humanos. Processos de negócio podem ser medidos por diferentes medidas de desempenho, tais como custo, qualidade, tempo e satisfação do cliente. Processos de negócio podem ser modelados por Serviços Web Compostos.

2.8 Workflows

Um Workflow pode ser definido como uma sequência de passos de processamento (execução de regras de negócio, tarefas e transações), durante as quais informações e objetos físicos são passados de um passo para outro. Workflow envolve atividades, pontos de decisão, rotas, regras e papéis ou tarefas (PAPAZOGLOU, 2008). Modelos de fluxo, em inglês *flow model*, são usados para a especificação de interações

complexas entre serviços, de modo que a composição desses serviços provê a funcionalidade requerida para se atingir certo objetivo de negócio.

Workflows orientados a processos são usados para automatizar processos de negócio quando a estrutura é bem definida e estável no decorrer do tempo. O workflow do processo de negócio é construído de atividades que são implementadas como um conjunto de operações dentro de um escopo implementadas por um ou mais serviços Web.

2.9 Serviços Web Compostos

A composição de serviços Web visa satisfazer requisições que não podem ser satisfeitas por nenhum serviço Web sozinho. Nesse caso, um serviço Web composto, do inglês *Composite Web Service*, obtido pela combinação de Web services disponíveis, pode ser usado. Na literatura são reportadas várias linguagens para compor Web services, por exemplo, WS-BPEL (CURBERA, 2003), WS-CDL (KAVANTZAS et al., 2005) e XLANG (THATTE, 2001). Atualmente BPEL é o padrão *de facto* para a composição de Web Services (MAAMAR et al., 2010).

Chakraborty e Joshi (2001) diferenciam entre composição proativa e reativa. A primeira é um processo *offline* que junta *a priori* serviços Web componentes disponíveis para formar o serviço Web composto. Este é pré-compilado e está pronto para a execução quando o usuário faz a requisição. A última forma de composição cria serviços *Web* compostos *on-the-fly* no momento da requisição do usuário. Por causa da propriedade *on-the-fly* é necessário um módulo dedicado encarregado de identificar os serviços Web componentes necessários, fazê-los colaborarem, acompanhar sua execução e resolver conflitos caso eles aparecerem.

2.10 Orquestração e Coreografia

Normalmente, Serviços Web expõem operações de certas aplicações ou sistemas de informação. Consequentemente, combinar vários serviços Web para formar um serviço Web composto envolve a integração das aplicações subjacentes e suas funcionalidades (JURIC, s.d.). Serviços Web podem ser combinados de duas maneiras: Orquestração (*Orchestration*) e Coreografia (*Choreography*).

Na Orquestração, que é normalmente usada em processos de negócio privados, um processo central (que pode ser outro serviço Web) controla os demais serviços e coordena a execução das diferentes operações envolvidas no processo. Os serviços envolvidos não sabem, e não precisam saber, que estão envolvidos num processo de composição e que são parte de um processo de negócio de nível superior. Somente o coordenador da orquestração está ciente do objetivo, assim, a orquestração é centralizada com a definição explícita das operações e da ordem de invocação dos serviços Web componentes. A figura 2.3 ilustra o processo. Nela há um coordenador e alguns serviços Web usados na composição. A ordem pré-definida das operações é representada pelos números.

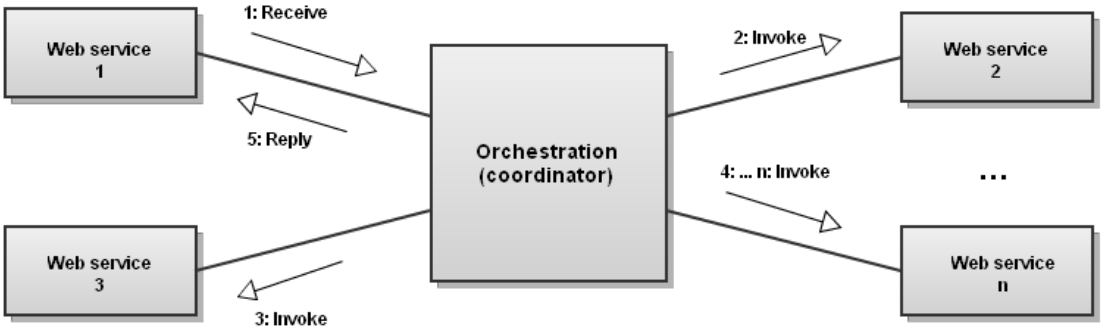

Figura 2.3: Composição de serviços Web com Orquestração (JURIC, s.d.)

A Coreografia, em contraste, não é baseada em um coordenador. Cada serviço Web envolvido na coreografia sabe exatamente quando executar suas operações e com quem interagir. Coreografia é, portanto, um esforço colaborativo focado na troca de mensagens entre os participantes de processos de negócios públicos. A figura 2.4 ilustra o processo. Nela, temos serviços Web cientes do processo de negócio, das operações a executar, das mensagens a serem trocadas e da ordem com que tudo isso vai ocorrer.

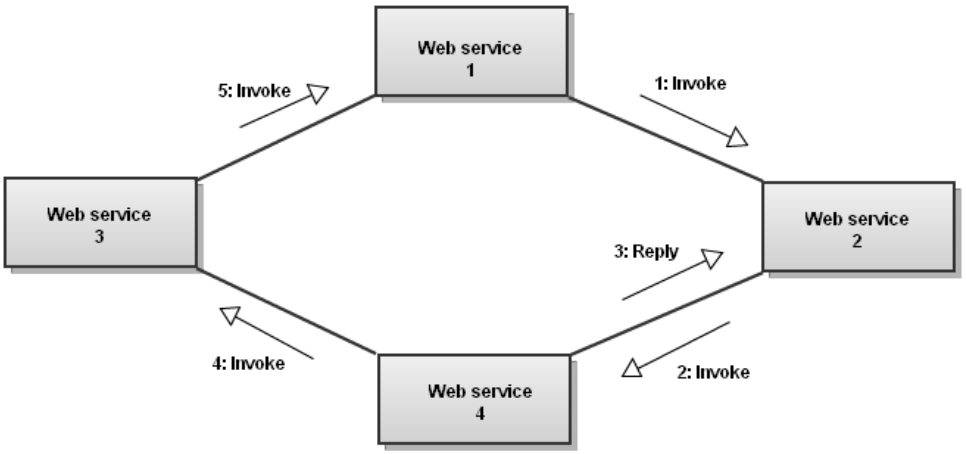

Figura 2.4: Composição de serviços Web com Coreografia (JURIC, s.d.)

Da perspectiva de composição de serviços Web para execução de processos de negócio, orquestração é um paradigma mais flexível e tem as seguintes vantagens sobre coreografia (JURIC, s.d.):

- A coordenação de processos componentes é centralmente gerenciada por um coordenador conhecido;
- Serviços Web podem ser incorporados sem estarem cientes de que estão participando de processo maior;
- Cenários alternativos podem ser estipulados em caso de ocorrência de falhas;

A priori, apenas serviços Web compostos que seguem o paradigma de orquestração são suportados pelos *CWSMarts*.

2.11BPEL

Recentemente, BPEL emergiu como padrão para definir e gerenciar atividades de processos de negócios e protocolos de interação de negócios compreendendo Web services colaborativos. É uma linguagem baseada em XML para especificação formal de processos de negócio e protocolos de interação (PAPAZOGLOU, 2008). O desenvolvimento da linguagem BPEL foi guiado pelo requerimento de suportar modelos de composição de serviços que provêm integração flexível, composição recursiva, separação da composição em relação às referencias, conversação *stateful* (presença da informação de estado), gerenciamento de ciclo de vida e propriedades de recuperação.

BPEL suporta dois diferentes modos de descrição de processos de negócio que seguem orquestração e coreografia (JURIC, s.d.):

- Processos Executáveis (*Executable process*) que permitem a especificação de todos os detalhes do processo de negócio. Eles seguem o paradigma de orquestração e podem ser executados por um "motor de orquestração" (*orquestration engine)*. É para esse tipo de processo que os *CWSMarts* foram projetados.
- Protocolos abstratos de negócios (*Abstracty business protocols*) que permitem a especificação somente da troca mensagens públicas entre as partes. Eles não incluem detalhes internos do fluxo do processo e não são executáveis. Eles seguem o paradigma de coreografia.

Um processo BPEL especifica a ordem exata na qual os serviços Web participantes devem ser invocados, operação que pode ser feita tanto sequencialmente quanto em paralelo. Com BPEL, é possível especificar comportamentos condicionais. Por exemplo, a invocação de um serviço pode depender do valor retornado por uma invocação anterior. É possível também construir *loops*, declarar variáveis, definir tratadores de exceções e assim por diante. Combinando todas essas construções é possível definir processos de negócio complexos de uma maneira algorítmica. Na verdade, devido a processos de negócios serem em essência grafos de atividades, pode ser útil representá-los usando diagramas de atividade UML.

Em um cenário típico, o processo BPEL recebe uma requisição de um cliente. Para satisfazê-la, o processo invoca os serviços Web envolvidos e então retorna ao cliente a resposta. Na figura 2.3 tem-se uma visão esquemática de um processo BPEL para administração de viagens de funcionários de uma empresa. O cliente invoca o processo de negócio especificando o nome do funcionário, o destino e a data de partida e de chegada. O processo BPEL checa primeiramente a classe de viagem para o funcionário através do serviço Web *EmployeeTravelStatus*. Então o processo verifica o preço da passagem aérea junto a duas companhias aéreas: *American Airlines* e *Delta Airlines*, assumindo que ambas as companhias disponibilizem um serviço Web para tal consulta. Essa verificação é feita em paralelo e de forma assíncrona, conforme pode ser verificada pelo uso de *Call-backs* no *portType* do processo. Por fim, o processo seleciona a opção de menor preço e retorna o plano de viagem para o cliente invocando uma operação (*call-back*) no cliente, ou seja, o processo BPEL é definido assincronamente, o que ocorre frequentemente quando há o uso de serviços assíncronos.

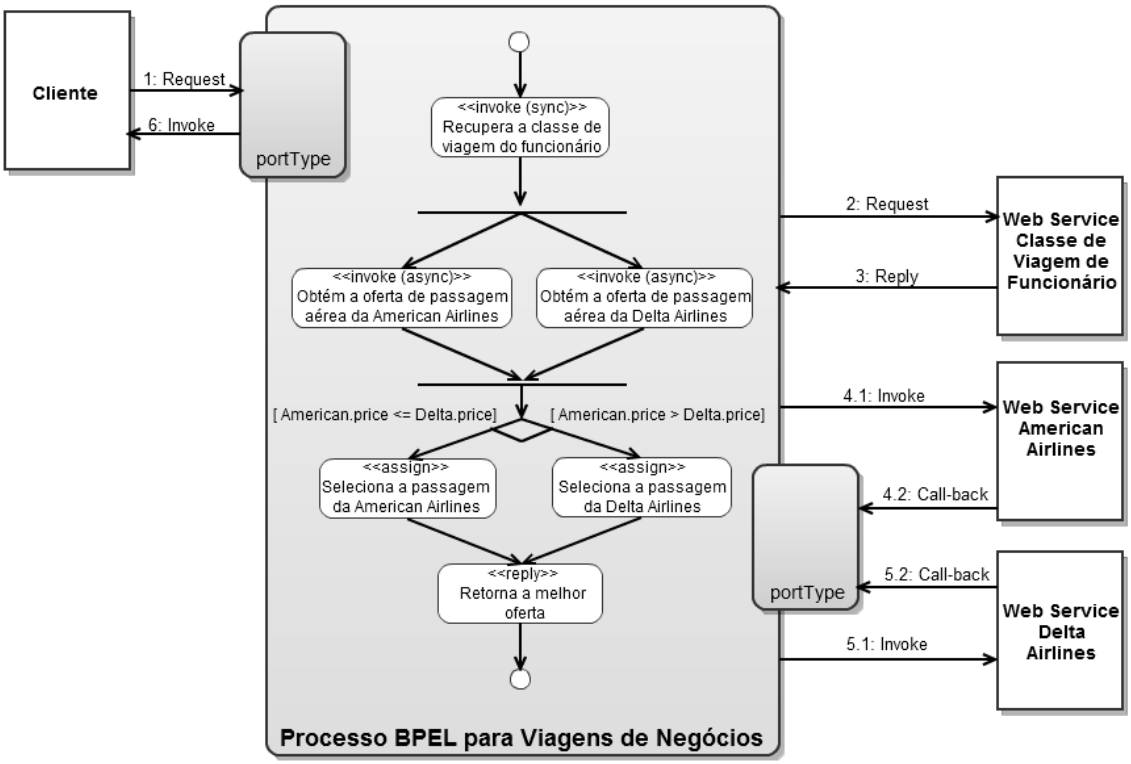

Figura 2.5: Exemplo de processo BPEL para viagens (JURIC, s.d.)

Ao definir um processo em BPEL o que se faz é essencialmente definir um novo serviço Web que é uma composição de serviços Web já existentes. Os passos para elaboração de um processo BPEL envolvem definir a descrição do novo serviço usando WSDL, definir os parceiros (*partner links*) que são as partes envolvidas na interação com o processo BPEL (os serviços Web invocados e o cliente que invoca o processo), declarar variáveis e por último escrever a definição da lógica que o processo implementa. A descrição WSDL e o código do processo BPEL para o exemplo do processo de negócio para viagens estão no anexo A e anexo B, respectivamente.

Por ser atualmente um padrão na composição de Web services, essa será provavelmente a linguagem na qual a maioria dos novos CWS que tentarão ingressar no *CWSMart* estarão especificados. Conforme será discutido posteriormente, os módulos *matcher* e *description* trabalharão diretamente com estas especificações.

2.12 Data Mart

Data mart (DM) é uma versão especializada de um *Data Warehouse* (DW). A diferença chave entre eles é que a criação do *data mart* ocorre em razão de uma necessidade especifica, predefinida para um certo grupo e configuração de dados selecionados. Segundo Spenik e Sledge (2001, p. 648), "*um DM é um repositório de dados reunidos a partir de dados operacionais ou outras origens que é projetado para atender um departamento particular ou um grupo funcional*". Por tratar-se de um conjunto de dados menor, permite que a sua implementação seja compreendida em um espaço de tempo menor, reduzindo também o custo de investimento.

Entre as razões para o desenvolvimento de *CWSMarts* estão o fácil acesso a dados frequentemente utilizados como serviços Web compostos frequentemente requisitados,

serviços Web componentes mais utilizados, o que se revela importante na medida que alguns desses serviços podem acabar sendo sobrecarregados e se tornarem *bottlenecks* (gargalos). Ter acesso a estas informações implica diretamente em melhoria no tempo de resposta para o usuário final; manutenibilidade dos CWS na medida em que se identificam mudanças nos serviços Web componentes; ter acesso a informações estatísticas sobre o funcionamento dos CSW é importante também no processo de recomendação do serviço ao cliente.

2.12.1 Data Warehouse *versus* Data Mart

> Um *Data Warehouse* é uma agregação central dos dados. Um *Data Mart* pode ser derivado de um *Warehouse* ou o *Warehouse* pode ser derivado de *Data Marts* menores especializados. O *Data Mart* enfatiza facilidade de acesso e capacidade de utilização para um propósito particular de projeto. Em geral, um *Data Warehouse* tende a ser estratégico mas freqüentemente incompleto; um *Data Mart* tende a ser tático e dirigido para atender uma necessidade imediata (SPENIK e SLEDGE, 2001, p. 648).

A tabela 2.1 mostra uma comparação entre DM e DW.

Tabela 2.1: Diferenças-chave entre *Data Marts* e *Data Warehouse*

Data warehouse	***Data mart***
Utilização corporativa	Utilizado por departamento ou unidade funcional
Implementação complexa e demorada	Implementação rápida e fácil
Maior volume de dados	Volume de dados menor e mais especializado
Desenvolvido utilizando dados	Desenvolvido a partir das necessidades de dados dos usuários atualmente disponíveis

Fonte: Spenik e Sledge (2001)

2.12.2 Tabela de Fatos, Tabela de Dimensões, Cubos

No contexto de DM e DW, conforme Machado (2000), fato é uma coleção de itens de dados, composta de dados de medidas e de contexto. A tabela de fatos armazena medições quantitativas do negócio. A maioria das colunas em uma tabela de fatos é numérica e aditiva, pois tais dados podem ser usados pra executar cálculos necessários.

> A principal função de uma tabela de dimensão é reunir os atributos que serão utilizados para qualificar as consultas e cujos valores serão utilizados para agrupar e sumarizar as métricas (ou fatos). Ou seja, as tabelas dimensão contêm atributos textuais que funcionam como filtros para as consultas dos usuário (ITALIANO e ESTEVES, [s.d], p. 37).

A representação mais intuitiva de visualizar um modelo dimensional é através do desenho de um cubo. Petkovic (2001, p. 450), define o cubo como "[...] um subconjunto de dados do DW que pode ser organizado em estruturas multidimensionais.". De acordo com Parrini (2002), um cubo possui diversas células e cada fato é guardado em uma célula. O fato possui um conjunto de medidas que o representa. Para se referenciar a um fato específico, é necessário especificar qual a sua coordenada ao longo de cada dimensão.

Como será visto em mais detalhes na seção 4.4, no *data mart* do *CWSMart* a tabela fato contém dados relativos a cada execução de um serviço Web composto, enquanto

que as tabelas dimensões contém informações adicionais usadas para filtrar determinado conjunto de “fatos” execução.

2.12.3 OLAP

A técnica de *Data Marts* e *Data Warehouse* prevê uma forma completa e concisa para o armazenamento de dados históricos de caráter informativo. Porém, para que estes dados possam ser consultados de forma facilitada e interativa pelo usuário, uma nova categoria de softwares foi proposta para permitir aos analistas, gerentes e executivos estudar e entender os dados de maneira rápida, consistente, e que possibilita o acesso a todas as possíveis visões da informação. Esta nova categoria de softwares é chamada de ferramentas *On-Line Analytical Processing* (OLAP). Embora não seja o foco principal, o *data mart* e as operações que podem ser desenvolvidas sobre ele são uma importante ferramenta para análise e tomada de decisões para os administradores do *CWSMart*.

2.12.3.1 Operações Básicas

De acordo com Machado (2000) e Kimball (1998), existem diferentes formas de navegar entre as hierarquias a fim de selecionar quais perspectivas deseja-se analisar. Dentre as operações básicas podem ser citadas:

- ***Drill Across***: está relacionado ao fato de poder movimentar de um esquema para o outro, desde que ambos tenham algumas dimensões em conformidade. Ou seja, as mesmas dimensões estão compartilhadas. Como exemplo, é possível citar os dados de compra de matéria prima, sumarizados por mês, semana e dia. Executando um *Drill Across*, o usuário passaria do mês direto para o dia.
- ***Drill Down***: ocorre quando o usuário aumenta o nível de detalhe da informação, diminuindo o grau de granularidade. O usuário pode navegar do mais alto nível até o mais detalhado. Como exemplo, se pode citar a visualização do total de rotatividade de em um departamento de recursos humanos. Realizando o *Drill Down*, poderíamos visualizar a rotatividade de um determinado cargo.
- ***Roll Up***: é o contrário do *Drill Down*. Ocorre quando o usuário aumenta o grau de granularidade, diminuindo o nível de detalhamento da informação. O usuário pode navegar do nível de detalhe até o nível mais alto de sumarização de dados. Como exemplo, se pode citar o total de vendas de um determinado produto, quando utilizado o *Roll Up*, passaria-se a visualizar a venda de todos os produtos.
- ***Drill Through***: este conceito está relacionado ao fato de poder armazenar um nível de detalhe menor do que aquele mencionado na tabela fato, mas permitido pela sua granularidade. Por exemplo, em um nível de granularidade de pessoal, por dia e departamento, deseja-se buscar uma informação que está presente no cartão-ponto. A operação *Drill Through* poderia efetuar esta busca no próprio sistema operacional caso houvesse compatibilidade entre os dois ambientes.
- ***Slice and Dice***: é uma das principais características de uma ferramenta OLAP. Este conceito surgiu da necessidade de recuperar uma “fatia” de um microcubo. Ele serve para modificar a posição de uma informação, alterar as linhas por colunas de maneira a facilitar a compreensão dos usuários e girar o cubo sempre que houver necessidade. Como exemplo em um cubo que representa a produção

de roupas e calçados, utilizando o *Slice and dice*, poder-se-ia visualizar somente a produção de calçados.

Para o *CWSMart*, estas operações são válidas para dados provenientes dos *logs* de funcionamento dos CWS, conforme será explicado posteriormente, essa tarefa de coletar informações a partir da execução dos CWS será tarefa do módulo *monitor*. Dessa forma se permite a manipulação das dimensões relacionadas com o funcionamento (execução) dos CWS no *CWSMart*.

2.13 Framework

Os *frameworks* estão cada vez mais presentes na área de desenvolvimento de software, pois são a maneira pela qual os sistemas orientados a objetos conseguem a maior reutilização (GAMMA et. al, 2000). O reuso de *frameworks* no desenvolvimento de aplicações, aumenta a produtividade como um todo e qualidade do software desenvolvido.

Na literatura podem ser encontradas muitas definições para *frameworks* orientados a objetos, a seguir são apresentadas algumas delas.

Segundo Silva (2000), a abordagem de *frameworks* orientados a objetos utiliza o paradigma de orientação a objetos para produzir uma descrição de um domínio para ser reutilizada. Sendo assim um *framework* é uma estrutura de classes inter-relacionadas, que correspondem a uma implementação incompleta para um conjunto de aplicações de um domínio, sendo que esta estrutura de classes deve ser adaptada para a geração de aplicações específicas.

Para Pree (1995), um *framework* é uma coleção de classes abstratas e concretas que representam um subsistema, estas classes (abstratas e concretas) podem ser estendidas ou adaptadas para construir um novo subsistema.

Segundo Gamma et al. (2000), um *frameworks* predefine os parâmetros de projeto, de maneira que o projetista/implementador da aplicação possa se concentrar nos aspectos específicos da sua aplicação.

No contexto deste trabalho e com base nos conceitos apresentados, o termo *framework* é aqui aplicado no sentido de que esse define uma estrutura padrão para um *CWSMart*, bem como encapsula um conjunto de classes e funções básicas para a instanciação deste e para a o gerenciamento das informações nele armazenado.

3 TRABALHOS RELACIONADOS

Frameworks são muito utilizados no processo de desenvolvimento de software. A possibilidade de reuso de grandes componentes de software trás inúmeras vantagens relacionadas ao aumento de produtividade e na qualidade do produto final. Um dos primeiros *frameworks* que se tem conhecimento é o *MVC* desenvolvido no ambiente do *Smalltalk*-80 (PRE, 1995). Este *framework* é muito utilizado hoje em dia na comunidade de *software*. Através do *MVC* é possível dividir uma aplicação em três camadas distintas, desta forma separando a lógica da aplicação chamada de *Model* (modelo), da *interface* do usuário conhecida como *View* (visão) e do fluxo da aplicação, o *Controller* (controlador). Cada camada destas corresponde a uma classe abstrata do *framework MVC*, e elas cooperam através de um protocolo de interação bem definido. Outra classe de *framework* muito utilizado hoje em dia são os chamados GUI *framework*s, como exemplo podem ser citados GTK+, QT, Java Swing, Adobe Flex, Motif, IBM Presentation Manager, Microsoft .NET/WinForms, entre muitos outros.

No campo de Web services e Web services compostos, Chan (2009) propõe um *framework* que provê estratégias de reações flexíveis quando um evento inesperado ocorre na execução do *Web service* composto. A proposta prevê a monitoração, detecção e recuperação de falhas ocorridas em tempo de execução, para que se garanta a correta execução do CWS. Em essência, o *framework* proposto, chamado *Web Service Composition Recovery Framework (WSCRF)*, consiste de três módulos centrais, a saber, *monitoring*, *diagnostics* e *recovery*. O módulo de monitoração captura entradas, dados e mensagens de controle a partir do *workflow* que determina o CWS. Os padrões de interação assim como os padrões de controle são extraídos para formar uma base com o propósito de análise. O módulo de diagnóstico detecta e identifica as causas de falhas na composição. O módulo de recuperação tem por objetivo, em caso de detecção de um evento inesperado, recuperar o processo composto de modo que o objetivo original seja atingido.

Em outro trabalho relacionado, Chollet e Lalanda (2009) apresentam um *framework* para composição, ou orquestração, de serviços Web em um nível abstrato com a capacidade de geração de código apropriado para a aplicação. O *framework* provê uma solução extensível para expressar separadamente fluxos de controle e propriedades não funcionais. Ele é baseado nas noções de serviços abstratos, meta-modelagem e programação generativa (*generative programming*).

Chukmol (2008) apresenta um *framework* para descoberta de Web services levando em conta o reuso dos resultados das buscas através da técnica de *cache*, a qualidade do serviço através de teste qualitativo, a evolução dos Web services através da técnica de *version track* e um esquema *novel* para descoberta de *Web services* usando *annotating information* do usuário. O *framework* é composto de diferentes módulos. O módulo *Collection engine* opera em *background* e é responsável por coletar diferentes Web

services a partir de diferentes provedores. O módulo *Internal storage* é responsável por guardar estes Web services em uma única e acessível fonte central. O módulo *Testing module* é encarregado de analisar cada Web service salvo no repositório e preparar automaticamente cenários de teste e dados de acordo com a especificação. O *Indexing module* é responsável por indexar de diferentes formas o conteúdo do repositório central. Cada índice (*index*) é usado por um *matcher* no *Matching module*. Cada *matcher* do *Matching Module* implementa um algoritmo de *matching* para tratar um tipo de *query*. Os *matchers* são componentes centrais da proposta na medida em que eles calculam similaridade entre a entrada do usuário (*query*) contra o conjunto de *Web services* indexados pelo módulo anterior. Há ainda outros módulos e operações, a saber, *Combination Module*, *User Test*, *Annotation Based Discovery*, *Caching*, *Query customization* previstas no trabalho.

Maamar et al. (2010) propõe uma arquitetura de camadas para *CWSMarts*. Esses são centros especializados onde *Web services* estão já combinados e empacotados em *Web services* compostos prontos para uso. Esses CWS podem ser invocados a qualquer momento sem passar pelos passos regulares de descoberta, seleção, composição, invocação e monitoramento. *CWSMarts* baseiam-se em dois conceitos maiores, o de *community* de *Web services* e o de *data marts*. O objetivo deste trabalho é desenvolver um *framework* para o *CWSMart* capaz de armazenar Serviços Web Compostos para áreas especificas e oferecer funcionalidades específicas para consultá-los, reorganizá-los e, também, recomendá-los. No capítulo seguinte será discutido os aspectos da arquitetura proposta para *CWSMarts* bem como a arquitetura do *framework*.

4 FRAMEWORK PARA MARTS DE SERVIÇOS WEB COMPOSTOS

Neste capítulo será especificado em detalhes a estrutura do *CWSMart*, a partir disso será detalhado a arquitetura do *framework* para o *Mart* de serviços Web compostos e, por fim, será descrito uma prova de conceito para o *framework* com o desenvolvimento de uma aplicação que faz uso da estrutura proposta.

4.1 Análise de Serviços Web Compostos

Maamar et al. (2010) definem três perspectivas a partir das quais serviços Web compostos (CWS) podem ser analisados: estrutura, agrupamento e competitividade. A perspectiva de estrutura (*structure*) foca na lógica de negócio do processo, nos serviços Web componentes e na linguagem utilizada para especificar esse CWS. A perspectiva de agrupamento (*clustering*), por sua vez, foca nos aspectos relacionados com a população dos *CWSMarts*, ou seja, como atrair CWS para eles, deixando-os cientes da existência de *CWSMarts*, no acesso à funções de similaridade de funcionalidades entre os CWS, no rastreamento e mineração das execuções dos CWS. A perspectiva de competitividade (*competitiveness*), por fim, foca na competitividade que pode ser estabelecida entre CWS residentes no *CWSMart*.

Um *CWSMart* pode ser construído a partir de uma arquitetura de quatro camadas (Figura 4.1). A primeira camada, chamada de "*component*" é composta pelos serviços Web componentes que os provedores desenvolvem, descrevem e publicam em registros dedicados como UDDI. Um exemplo de funcionalidade de um serviço Web poderia ser a reserva de um quarto de hotel. Essa camada alimenta a segunda, chamada de "*community*", essa camada é responsável por hospedar serviços Web componentes com uma mesma funcionalidade. Importante ressaltar que apesar da heterogeneidade dos serviços Web, suas funcionalidades são suficientemente bem definidas e homogêneas para permitir concorrência de mercado (BUI e GACHER, 2005).

A terceira camada é a camada de serviços Web compostos, chamada "*composite*", ela surge das necessidades complexas dos usuários dos dias atuais. Esse problema é resolvido agregando-se diversos serviços Web componentes a partir de diferentes comunidades. Identifica-se as comunidades que satisfazem as necessidades do usuário e em seguida seleciona-se, entre os membros de cada comunidade, os serviços Web componentes que farão parte do CWS.

A quarta camada é a camada de "*mart*", ela é a contraparte da camada comunidade com ênfase desta vez em serviços Web compostos no lugar de serviços Web componentes. Os *CWSMarts* são alimentados a partir da camada de composição subjacente. Uma vez associados à *CWSMarts* os serviços Web compostos podem ser

analisados de acordo com as funções providas por esta camada. O acesso a funções de similaridade entre CWS é um exemplo de função provida pela camada de *mart*.

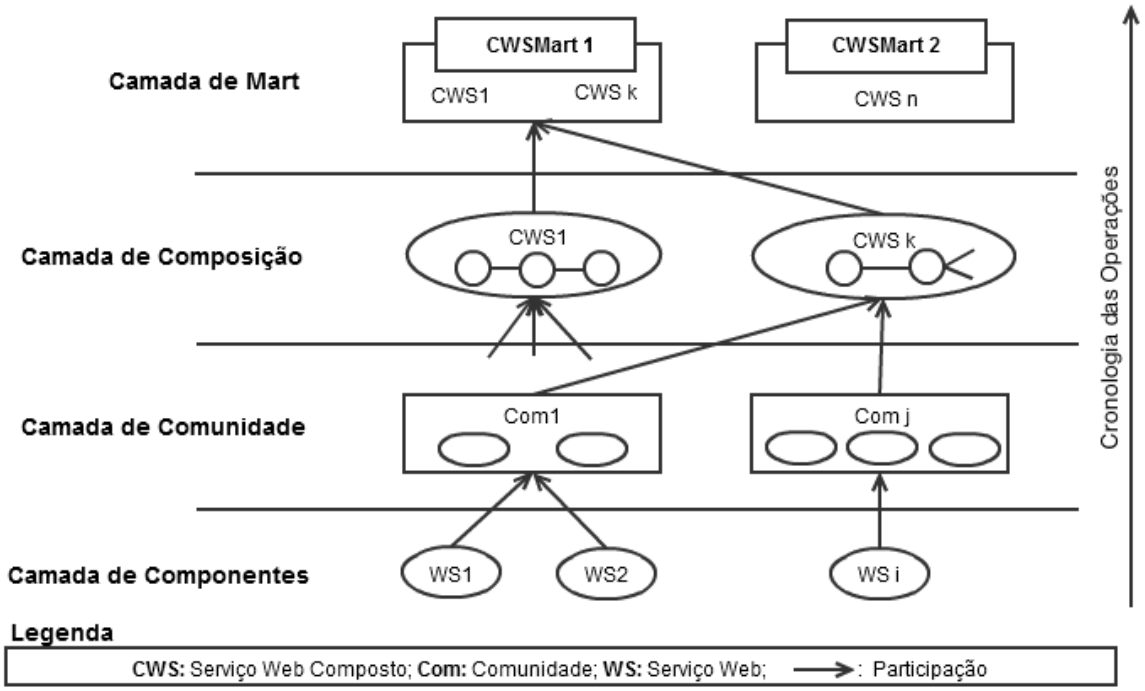

Figura 4.1: Arquitetura de 4 camadas para suportar *CWSMarts* (MAAMAR et al., 2010)

4.2 Estrutura e Especificação do *CWSMart*

Um *CWSMart* é um repositório de especificações de serviços Web compostos funcionalmente semelhantes entre si; aqui o conceito de especificação é diferente do de especificação, por exemplo, *a la* BPEL, que define um CWS. Ele consiste de um conjunto de módulos que dão suporte a funcionalidades providas por esta camada e em um *data mart* construído a partir das execuções dos CWS.

A figura 4.1 ilustra a estrutura interna de um *CWSMart*. Nela aparecem diferentes módulos, um repositório dedicado e um conjunto de plataformas computacionais.

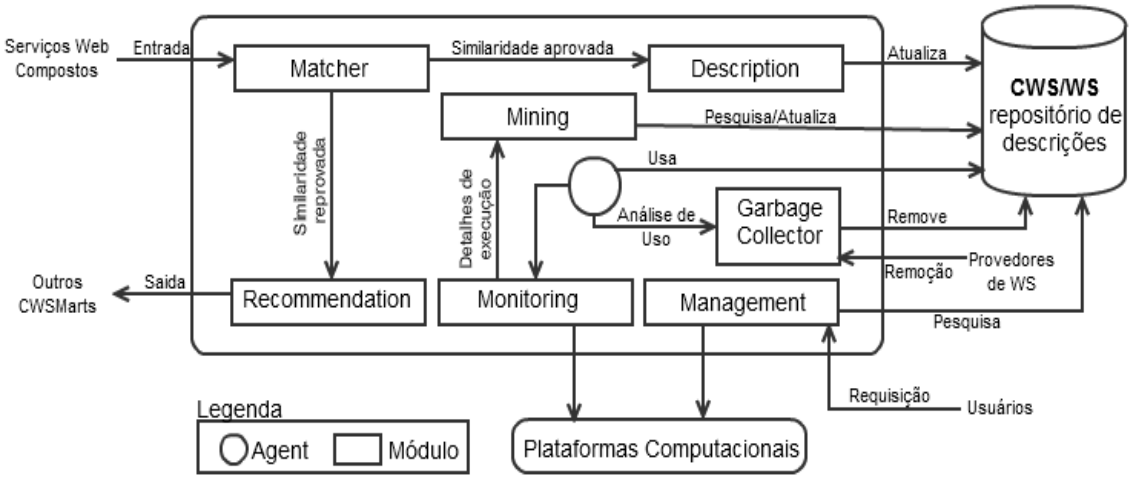

Figura 4.2: Estrutura Interna de um *CWSMart* (MAAMAR et al., 2010)

Antes de detalhar a responsabilidade e o objetivo de cada módulo, um refinamento foi identificado e proposto na estrutura interna do *CWSMart*: o módulo *mining* terá acesso a outra fonte de dados além do repositório que será um *data mart* construído especificamente para guardar dados de execução de CWS e permitir análises mais completas e objetivas.

O módulo *management* tem por objetivo tratar requisições dos usuários executando CWSs. Este módulo consulta o repositório de descrições do *mart* procurando serviços Web compostos capazes de satisfazer a requisição. Uma vez que o CWS apropriado é encontrado o módulo *management* implanta ("*deploy*") este serviço nas plataformas computacionais. Como visto na seção 2.9, devido a CWSs poderem ser projetados tanto proativamente quanto reativamente, existem dois possíveis cenários na implantação do CWS:

- No cenário proativo, o CWS é diretamente executado, pois todos os serviços Web componentes são conhecidos e as soluções para resolução de problemas, como falha em algum dos componentes, por exemplo, foram devidamente tratadas.
- No cenário reativo, o CWS precisa procurar os serviços componentes, selecionar os que satisfazem aos requerimentos não funcionais do usuário e, finalmente, os executar.

O módulo *matcher* ou *similarity* tem como sua função principal receber um serviço Web composto, comparar semanticamente sua funcionalidade com a funcionalidade do *mart* (o *mart* candidato a hospedar o CWS) (MAAMAR et at., 2007), checar as regras de inclusão, ou seja, avaliar se há lugar para o CWS baseado nas regras que controlam o conteúdo do *CWSMart*. Com base nisso, o módulo decide se o CWS candidato pode integrar este *mart*. Caso a integração seja aprovada o módulo *matcher* encaminha o CWS para o módulo *description* que tratará de inserir o CWS neste *mart* ou, em caso de a integração não ser aprovada, o CWS é encaminhado ao módulo *recommendation* que recomendará outros *marts* que possam aceitar este CWS.

O módulo recommendation é executado quando o módulo *matcher* reprova a integração, ou seja, há uma decisão de não hospedar o CWS candidato. Neste caso o módulo *recommendation* é responsável por recomendar outros *marts* que possam eventualmente aceitar o CWS em questão. Para que isso ocorra é necessário que este

módulo esteja ciente da existência de outros *CWSMarts*, isso pode ocorrer de duas maneiras: através de um repositório local, atualizado pelos próprios *CWSMarts* parceiros que informam da sua condição ou através de um repositório centralizado e comum a *CWSMarts* parceiros. A melhor estratégia tem ainda que ser estudada e especificada em trabalhos futuros. Há a possibilidade do *mart* em questão não saber de nenhum outro *mart*, e, nesse caso, a recomendação não existe, apenas há a comunicação ao dono do CWS para que este possa prosseguir com a busca de outro *CWSMart* por sua própria conta.

O módulo *description* é executado quando há a decisão de hospedar o *CWSMart* candidato. Sua principal função é analisar a especificação do CWS, que será recebida como entrada, possivelmente em BPEL, para então gerar outra especificação em termos dos atributos discutidos na seção 4.3 e, por fim, armazenar esta especificação no repositório.

O módulo *mining* alimentará o *data mart* a partir de informações coletados pelo módulo *monitoring* e fornecerá ferramentas para análise dos dados do *data mart* e do repositório de especificações dos CWS. Dessa forma será possível encontrar padrões recorrentes no *mart* ou entre os diversos CWS residentes no *mart*. Exemplos de padrões poderiam ser serviços Web componentes frequentemente utilizados, as restrições comuns de execução em relação a serviços Web componentes, os elementos de entrada comuns assim como suas origens (usuários ou outros serviços Web).

O módulo *monitoring* supervisiona os serviços Web compostos em tempo de execução seguindo sua implantação pelo módulo *management*. Os dados coletados são repassados para o módulo *mining* que alimenta o *data mart* com informações sobre a execução do CWS como tempos de resposta, exceções lançadas etc.

O módulo *agent* executa em intervalos de tempo a serem definidos pelo administrador do *CWSMart*. A cada execução este módulo executa uma análise de uso no *mart* consultando o módulo *mining* para descobrir quais serviços Web compostos e seus respectivos serviços Web componentes devem ser removidos do repositório. Essa informação é passada para o módulo *garbage collection* que então executa sua tarefa junto ao repositório.

O módulo *garbage collection* controla o processo de remoção de CWS e respectivos WS não usados do repositório. Essa função é importante para garantir o desempenho e a qualidade (QoS) do *CWSMart* em níveis aceitáveis. Se um serviço Web componente é usado por algum serviço Web composto que permanecerá no *mart*, então este serviço Web permanecerá no *mart*.

4.3 Proposta de Repositório

Através do repositório de descrições, um *CWSMart* hospeda vários serviços Web compostos que compartilham uma mesma funcionalidade independentemente de como eles são especificados, quanto eles cobram dos usuários, quantos serviços Web componentes eles possuem etc. Nesse sentido surge a questão de como armazenar e que atributos armazenar do CWS no *CWSMart*.

O problema de como armazenar é resolvido através de um banco de dados relacional. O problema de quais atributos armazenar é resolvido com base em uma análise na especificação inicial do CWS, a partir desta análise extrai-se a seguinte lista de argumentos que descreverá o CWS no *CWSMart*:

1. Linguagem: refere-se à linguagem usada para especificar o serviço Web composto como BPEL, XLANG, etc. O módulo *management* seleciona a plataforma computacional para a execução deste CWS com base nesta informação.
2. Serviços Web componentes *core*: identifica os serviços Web componentes que formam o *backbone* do serviço Web composto e assim não podem ser substituído ou ignorados em tempo de execução. Uma falha qualquer em um destes componentes causaria uma falha no serviço Web composto.
3. Serviços Web componentes opcionais: identifica-se os serviços Web componentes que o serviço Web composto pode "pular" em tempo de execução se eles não estiverem disponíveis por alguma razão. Esta classe de serviços Web provê suporte para a classe de serviços Web *core* quando é necessário, por exemplo, algum tipo de customização, como idioma preferido do usuário, por exemplo.
4. Requerimentos: refere-se aos requisitos funcionais e não funcionais exigidos para a execução deste CWS. O módulo *management* selecionará e implantará um CWS com base na requisição do usuário e nas informações de requerimentos.
5. Tipo de CWS: indica se o CWS foi projetado proativamente ou reativamente.

A figura 4.3 mostra um modelo de dados relacional que implementa as tabelas e atributos necessários para a persistência das informações do repositório. A tabela central chamada de CWS guarda informações sobre os CWSs armazenados na base. A tabela COMPONENT_WS guarda informações sobre os serviços Web componentes e está relacionada com a tabela CWS através da tabela associativa CWS_COMPONENT, possibilitando dessa forma que um serviço Web composto utilize um número arbitrário de serviços componentes. Cada serviço Web participante da composição desempenha um papel na composição (*core*, opcional) o que é representado na tabela COMPONENT_TYPE. A tabela REQUIREMENT armazena requerimentos funcionais e não-funcionais, definidas na tabela REQUIREMENT_TYPE, e está relacionada a tabela CWS através da tabela associativa CWS_REQUIREMENT. Adicionalmente, o tipo do serviço composto está definido na tabela CWS_TYPE. O "Apêndice A" especifica um dicionário de dados para o modelo da figura 4.3.

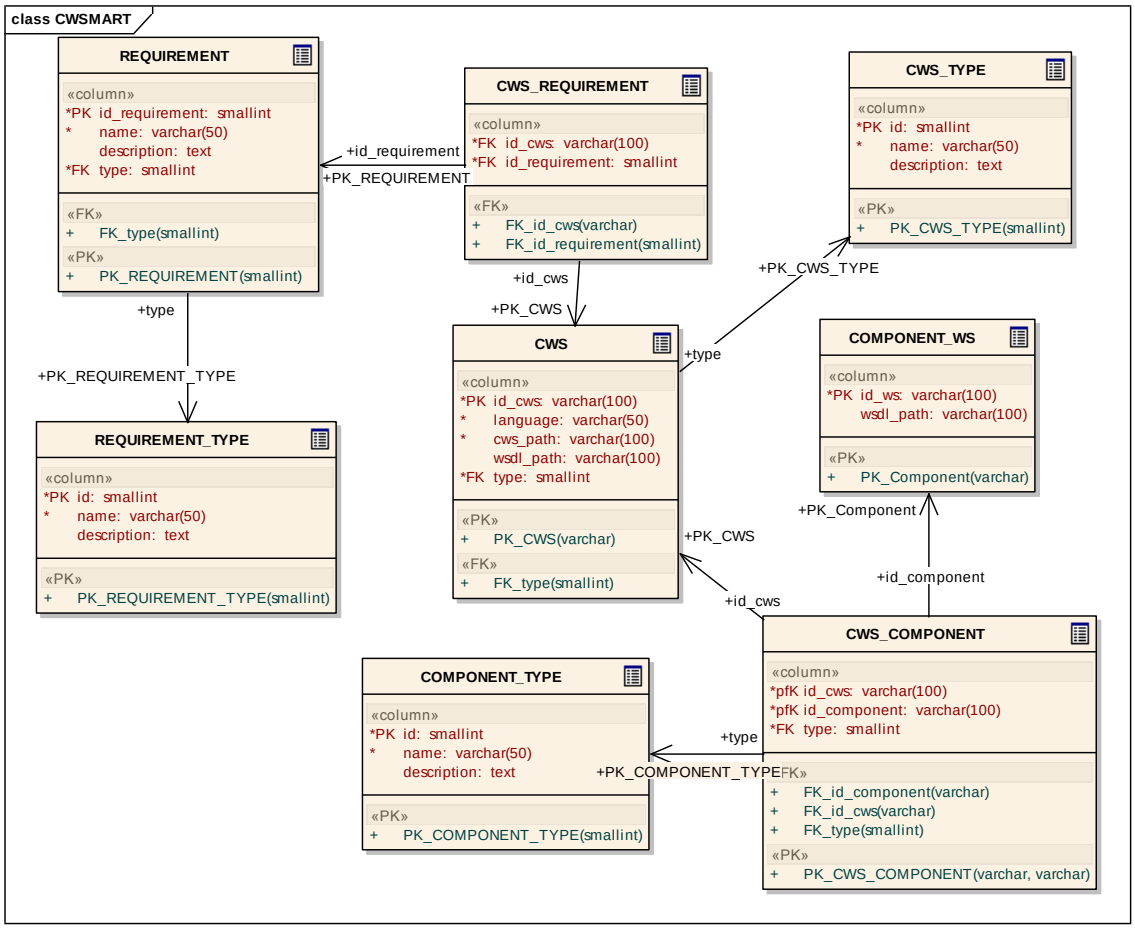

Figura 4.3: Repositório para o *CWSMart*

4.4 Proposta de Data Mart

Para dar suporte a um gerenciamento dos *CWSMarts* baseado em análises de situações e tendências anteriores e, dessa forma, ajudar no desenvolvimento de estratégias para decisões futuras, foi projetado um *data mart*.

A figura 4.4 contém um modelo composto por uma tabela fato (execução de um CWS) e cinco tabelas dimensão que compõe o DM. A tabela D_TIME armazena os dados referentes a informação de tempo, do momento em que a execução ocorreu. A tabela D_REQUEST_SOURCE armazena dados referentes a quem fez a requisição de execução do CWS. A tabela D_PROVIDER armazena os dados referentes ao provedor do serviço Web composto. A tabela D_PARAMETER armazena os dados passados por parâmetro ao CWS e por fim, a tabela D_STRUCTURE armazena os dados referentes a estrutura do serviço executado. No “Apêndice B” encontra-se um dicionário de dados do modelo ilustrado na figura 4.4.

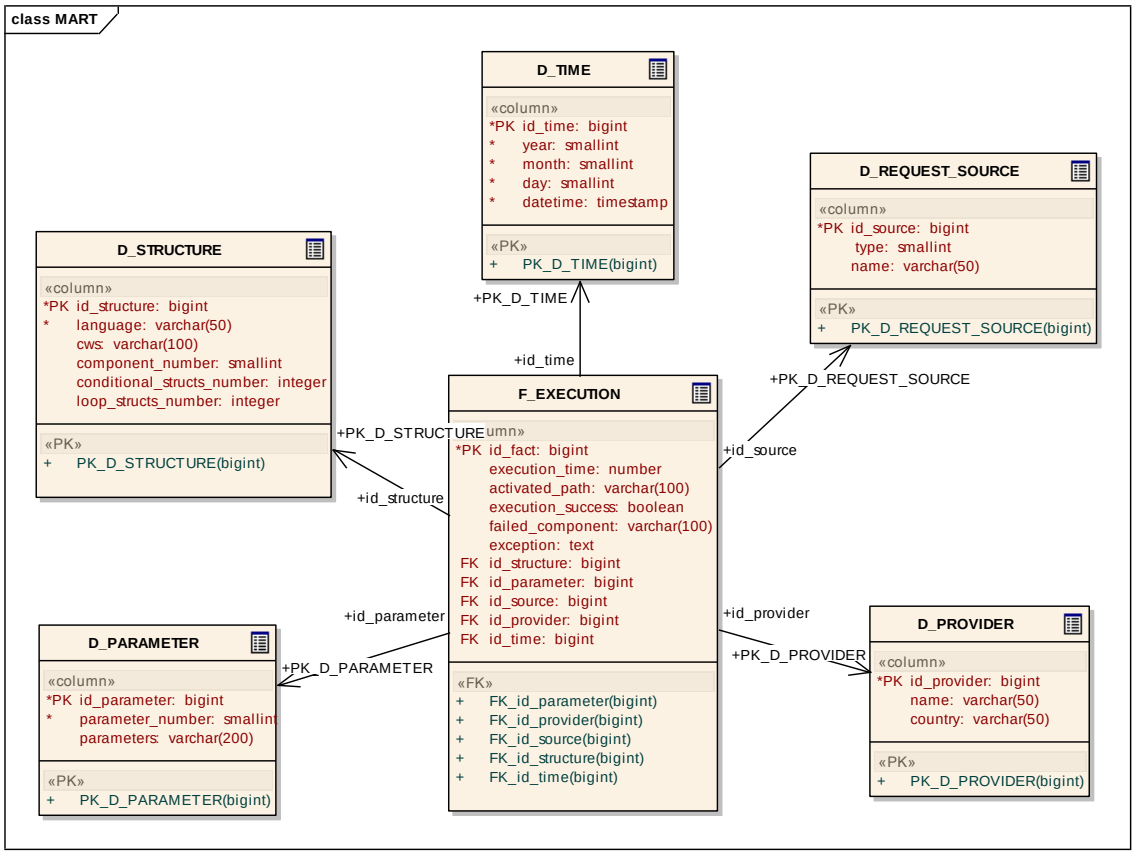

Figura 4.4: Proposta de *data mart* para o *CWSMart*

O *data mart* é alimentado por dados obtidos pelo módulo *monitoring* repassados ao módulo *mining* que atualiza e também implementa as ferramentas necessárias para análises baseadas no *data mart*.

O modelo adotado para o *data mart* foi o *Star Schema*. É o modelo mais difundido, e consiste basicamente em um conjunto de tabelas de dimensões relacionadas com uma única tabela de fatos. A tabela de fatos fica localizada no centro do modelo, e suas tabelas de dimensões ficam arranjadas ao redor desta unidade central, originando um formato semelhante ao de uma estrela (MACHADO, 2000).

4.5 Arquitetura do Framework

Um *framework* é um conjunto de classes cooperativas que implementam os mecanismos que são essenciais para um domínio de problemas específicos. Uma funcionalidade nova no domínio do problema pode ser criada estendendo as classes do *framework*. Ao contrário de um padrão de projeto, um *framework* não é uma regra geral de projeto. Normalmente, um *framework* usa múltiplos padrões (HORSTMANN, 2007).

Um "*framework* de aplicação" consiste em um conjunto de classes que implementa serviços comuns a um certo tipo de aplicações. Para criar aplicações reais, cria-se subclasses a partir de algumas classes do *framework* e implementa-se funcionalidades adicionais que são especificas da aplicação que se está construindo. Em um *framework* de aplicação, as classes do *framework*, e não as classes específicas da aplicação, controlam o fluxo de execução. Esse fenômeno normalmente é chamado de "inversão de controle".

A partir da análise dos serviços Web compostos e da estrutura interna do *CWSMart* foi desenvolvido um diagrama de classes que serviu de base para o desenvolvimento do *framework* proposto neste trabalho.

Os módulos especificados na seção 4.2 foram projetados e implementados no *framework* de modo que eles possam ser estendidos. Esse "suporte" a extensão, no caso do *framework* desenvolvido, atende a dois objetivos:

- Para comportar novas funcionalidades, como por exemplo, suporte a novas linguagens de composição de serviços Web e, para customização da aplicação para áreas especificas. Esse é o objetivo mais comum ao se desenvolver um *framework*, ou seja, criar uma solução reutilizável de software, de modo a reduzir o tempo e o esforço necessário para desenvolver uma aplicação completa para uma família de aplicações dentro de um domínio.
- Permitir que funcionalidades específicas dos módulos possam ser implementadas em trabalhos futuros. A vantagem nesse caso é diminuir o tempo e o esforço necessários tanto para o projeto quanto para a implementação de módulos específicos, e não na geração de uma aplicação completa como no objetivo anterior.

A figura 4.5 ilustra uma visão simplificada das classes do *framework*. A classe principal de cada módulo é abstrata, de modo que a aplicação que usa o *framework* precisa estender essas classes implementando algumas das funcionalidades e algoritmos. O *framework* agrega uma instancia de cada um dos módulos, que por sua vez, são instanciados através da classe abstrata chamada *ExtensionFactory* que segue o padrão de projeto *Abstract Factory* (GAMMA et. al, 2000). A ideia central de usar esse padrão é permitir o desenvolvimento do *framework* independentemente das instancias concretas dos módulos; que são conhecidas apenas em tempo de execução. Desse modo, aplicação tem a liberdade de fornecer os algoritmos específicos para cada um dos módulos ou mesmo usar módulos já implementados que podem ser simplesmente "plugados" no *framework* através de sua instanciação na "fábrica" concreta, ou seja, na classe que implementa os métodos abstratos da classe *ExtensionFactory*. Na figura 4.5 esta classe está representada com o nome de *ExtensionFactoryImpl* no domínio da aplicação.

Para utilizar o *framework*, é necessário na sua instanciação passar como parâmetro o nome da classe que implementa a "fábrica" concreta de módulos. A partir disso, é possível utilizar todos os métodos públicos disponibilizados pelo *framework* e implementados em cada um dos módulos. Basicamente, a aplicação, ao chamar algum dos métodos públicos do *framework* pela primeira vez, provoca a instanciação dos módulos envolvidos na operação através da fábrica concreta. A partir daí, o *framework* controla o fluxo de execução, fazendo uso, quando necessário, dos métodos implementados pela aplicação.

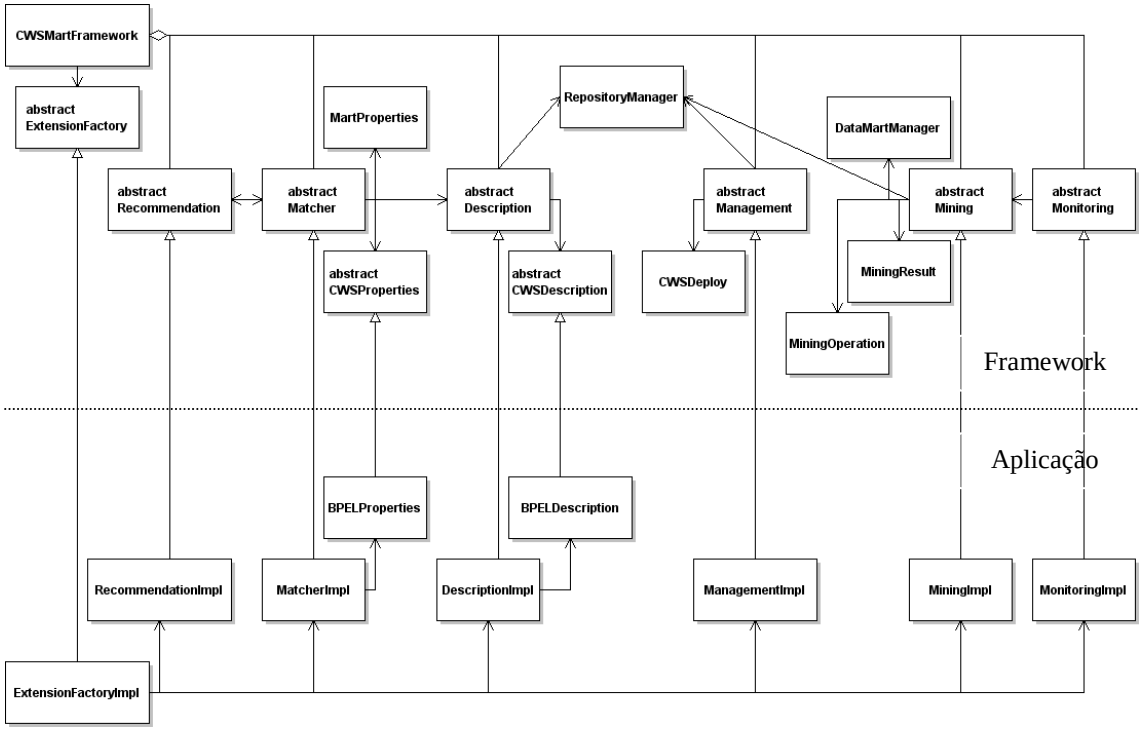

Figura 4.5: Modelo de Classes simplificado do *Framework* para *CWSMart*

De forma semelhante, o *framework* não conhece *a priori* em que linguagem de composição de serviços Web o CWS estará especificado, mas tem a habilidade de manipulá-los uma vez que implementa o padrão *Strategy* (GAMMA et. al, 2000). O padrão especifica uma forma de "configurar" uma classe com um dentre diferentes algoritmos. No caso, as classes a serem configuradas são as classes *CWSProperties* e *CWSDescription*, o algoritmo que varia é o algoritmo de *parsing* da descrição do CWS e o fator que determina qual algoritmo será utilizado é a linguagem no qual o CWS está especificado. Dessa forma, a aplicação pode trabalhar com a linguagem, ou as linguagens, que desejar, bastando para isso implementar o algoritmo de *parsing* para cada uma das linguagens suportadas.

Nas subseções seguintes será abordada em detalhes a implementação de cada um dos módulos e o como eles devem ser estendidos.

4.5.1 Módulo *matcher*

O módulo *matcher*, descrito na seção 4.2, tem a função principal de avaliar se o CWS candidato pode ingressar no *CWSMart*.

A figura 4.6 mostra o diagrama de classes associado com o módulo *matcher*. Nela podemos ver as classes do *framework* contidas no quadro de nome "*Framework*", e as classes de uma aplicação exemplo que utiliza o *framework* na parte inferior, fora do quadro. Podemos distinguir também:

- As classes de gerência do *framework*: a classe principal do *framework*, *CWSMartFramework*; as classes que implementam o padrão *Abstract Factory*, *Extension Factory* e *ExtensionFactoryImpl* e, a classe que utiliza o *framework TCC2ImplementationExample*.
- As classes do módulo *matcher* em si, no lado direito da figura.

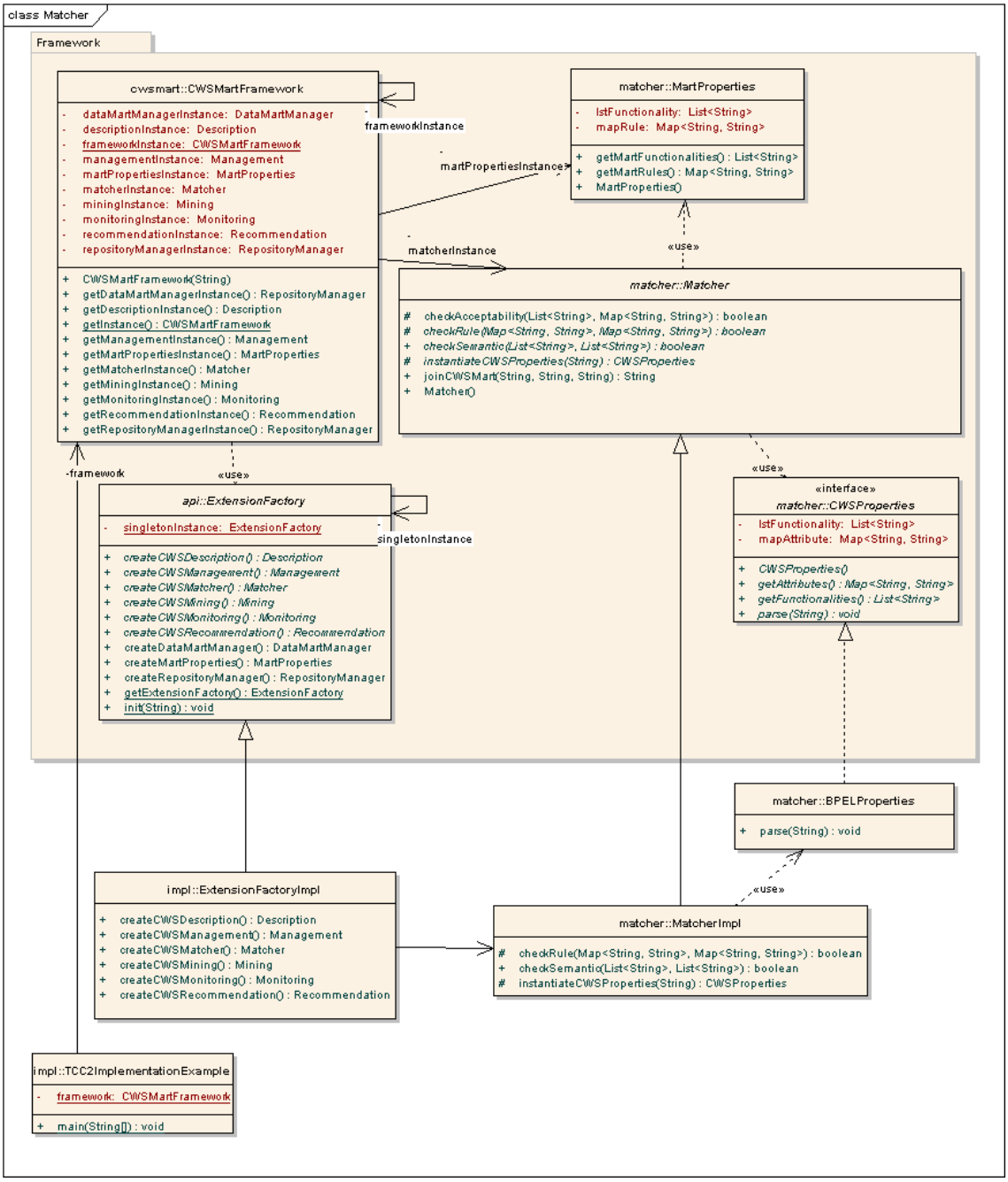

Figura 4.6: Diagrama de Classes: módulo *matcher*

O módulo *matcher* disponibiliza o método público *joinCWSMart* que verifica, através do método *checkAcceptability*, se o CWS candidato pode ingressar no *mart*. Caso a resposta seja positiva, o CWS é encaminhado para o módulo *description* para inserção no *mart* e, caso a resposta seja negativa, o CWS é entregue ao módulo *recommendation* para que se verifique a possibilidade de recomendação a outro *CWSMart* que eventualmente possa aceitar o CWS. O método *instantiate* instancia um algoritmo adequado para fazer o *parsing* da descrição do CWS.

O algoritmo implementado no método *checkAcceptability* está ilustrado na figura 4.7. Essencialmente, o CWS é analisado em relação a sua funcionalidade, se é compatível com a funcionalidade do *mart*, através do método *checkSemantic*, e é verificado também se as regras que controlam a "população" do *mart* são compatíveis

com os atributos do CWS, isso é feito através do método *checkRule*. O método público *checkSemantic* também é utilizado pelo módulo *recommendation* para verificar se o CWS em questão é funcionalmente compatível com os *CWSMarts* conhecidos.

Func checkAcceptability*(CWSFunctionality, CWSAttributes, MartFunctionality, MartRules)*
Input: *CWSFunctionality* – funcionalidade(s) do serviço Web composto
Input: *CWSAttributes* – atributos do service Web composto
Input: *MartFunctionality* – funcionalidade(s) do *CWSMart*
Input: *MartRules* – regras do *CWSMart*
Begin
if checkSemantic(*CWSFunctionality, MartFunctionality*) **and**
checkRule(*CWSAttributes, MartRules*) **then**
return true
- O serviço Web composto pode ingressar no *CWSMart*

end if
return false
- O serviço Web composto não pode ingressar no *CWSMart*

End

Figura 4.7: Algoritmo associado com o módulo *matcher* (MAAMAR et al., 2010)

Para criar uma instancia concreta do módulo *matcher* é necessário estender a classe *Matcher* do *framework*. Ao fazer isso, será necessário implementar três métodos definidos como abstratos no *framework*. Na figura 4.6 pode ser visto a classe *MatcherImpl* que estende a classe *Matcher* do *framework* e implementa os métodos *checkRule*, *checkSemantic* e *instantiateCWSProperties*. Também é necessário implementar o algoritmo de *parsing* da descrição do serviço Web composto. Na figura 4.6, a classe *BPELProperties* implementa o algoritmo de *parsing* para CWS descritos em BPEL.

4.5.2 Módulo *description*

O módulo *description* é executado quando o módulo *matcher* aprova a inclusão de um CWS no *mart*, conforme especificado na seção 4.2. O trabalho deste módulo consiste em especificar o CWS de acordo com um conjunto de atributos descritos na seção 4.3 e então armazenar essa descrição no banco de dados do repositório.

A figura 4.8 mostra o diagrama de classes associado com o módulo *description*. Nela podemos ver as classes envolvidas no dominio do *framework* e as classes envolvidas no dominio da aplicação. As classes de controle e gerencia do *framework* já foram explicadas na introdução da seção 4.5 e na subseção 4.5.1 e não serão analisadas aqui.

A classe principal, chamada *Description*, disponibiliza o método público chamado *addCWS*. É responsabilidade desse método obter os dados necessários a partir da especificação inicial do serviço Web composto, em BPEL por exemplo, e então salvar os dados obtidos no repositório. Para esta última tarefa o módulo d*escription* utiliza um componente de software interno ao *framework* chamado de *RepositoryManager*, responsável por comunicar-se com o banco de dados a fim de persistir dados ou recuperar dados.

Para implementar o módulo *description* é necessário estender a classe *CWSDescription* para implementar o algoritmo de *parsing* da descrição do CWS e estender a classe *Description* para implementar o método *instantiate* que nada mais faz

que instanciar a classe que implementa o algoritmo de *parsing* adequado para o CWS que se quer obter os atributos baseado na linguagem que ele está especificado.

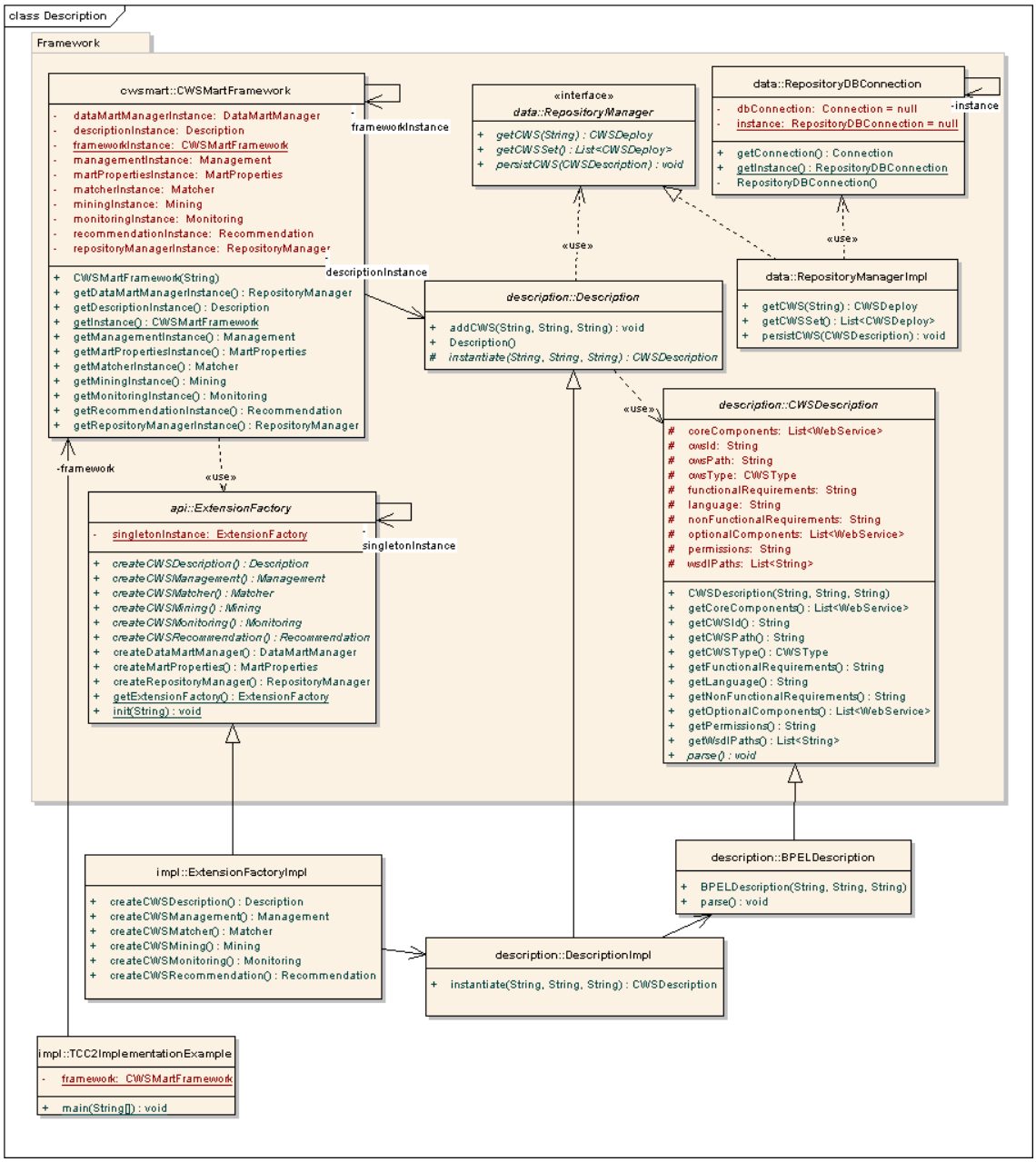

Figura 4.8: Diagrama de Classes: módulo *description*

4.5.3 Módulo *recommendation*

O módulo *recommendation* "entra em cena" quando o módulo *matcher* não aprova o ingresso do CWS no *mart*. É tarefa deste módulo recomendar um *CWSMart* funcionalmente compatível com o CWS, conforme especificado na seção 4.2.

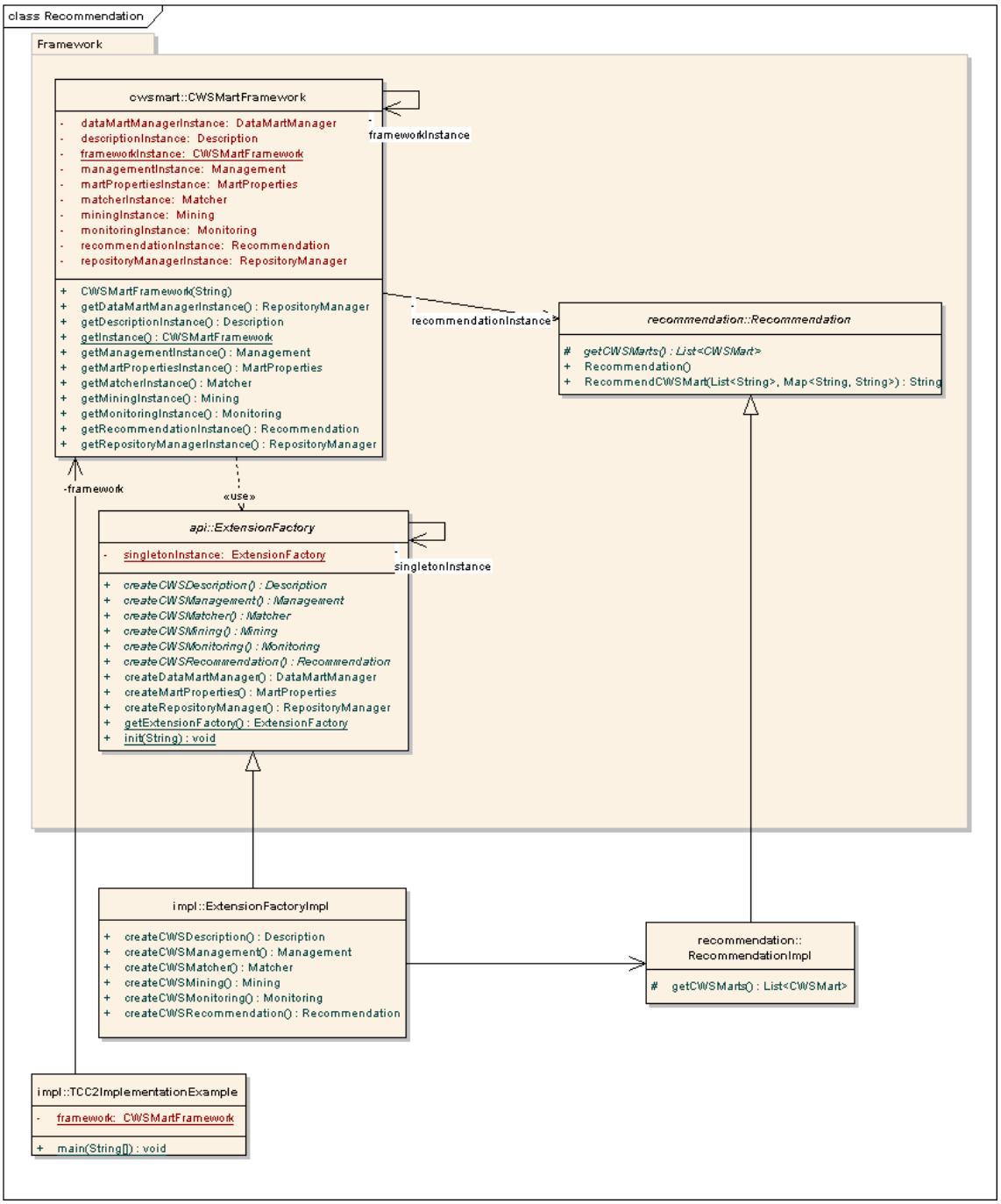

Figura 4.9: Diagrama de Classes: módulo *recommendation*

A figura 4.9 ilustra o diagrama de classes associado ao módulo *recommendation*. Além das já conhecidas classes de gerência do *framework*, temos a classe *Recommendation* que disponibiliza o método *recommendCWSMart*. O algoritmo associado a esse método pode ser visto na figura 4.10. Ele basicamente executa uma busca sequencial nos *CWSMarts* conhecidos até que se encontre um *CWSMart* compatível. Esse *mart* é então retornado para o usuário que direciona sua tentativa de inclusão ao *mart* indicado. Embora a recomendação auxilie a encontrar um *mart* funcionalmente compatível, ela não é garantia de sucesso de aceitação no *mart* indicado. Isso ocorre devido às checagens adicionais executadas no já visto módulo *matcher*.

Func recommend(*CWSFunctionality, SetOfCWSMarts*)
Input: *CWSFunctionality* – funcionalidade(s) do serviço Web composto
Input: *SetOfCWSMarts* – conjunto de todos os *CWSMarts*
Output: *idCWSMart* – identificador de um *CWSMart* que é do tipo string
Auxiliary: i – integer
Auxiliary: n – número de *CWSMarts* existentes, do tipo integer
Begin
for i = 0 *to n* **do**
if checkSemantic(*CWSFunctionality, MartFunctionality*$_i$) **then**
return *idCWSMart*$_i$
➢ O *CWSMart*$_i$ pode ser recomendado ao serviço Web composto
end if
endfor
return *null*
➢ O serviço Web composto não pode ser recomendo para nenhum *CWSMart*
End

Figura 4.10: Algoritmo associado com o módulo *recommendation* (MAAMAR et al., 2010)

Para criar uma instância concreta do módulo *recommendation* é necessário estender a classe *Recommendation* implementado o método abstrato *getCWSMarts* conforme ilustrado na figura 4.9 na classe *RecommendationImpl*. O *framework* não especifica uma forma padrão de tornar o módulo *recommendation* ciente da existência de outros *CWSMarts*. Conforme visto na seção 4.2, isso pode ser feito de duas formas: através de um repositório global ou através de um repositório local. Em trabalhos futuros pode-se estudar a melhor abordagem e, se for o caso, integrá-la ao *framework*. Atualmente cada aplicação deve implementar a forma mais conveniente de tornar essa informação disponível.

4.5.4 Módulo *management*

O módulo *management* é o responsável por tratar as requisições do usuário ativando a execução de CWSs, conforme foi visto na seção 4.2.

Na figura 4.11 temos as classes associadas a esse módulo. A classe *Management*, no dominio do *framework* no centro da figura, é a classe principal do módulo. Ela está associada ao componente *RepositoryManager*, responsável pelo acesso a base de dados do repositório, uma vez que o conteúdo do repositório é “comparado” em relação a requisição do usuário. Ela disponibiliza o método público *handleUserRequest* que é o ponto de entrada para a execução do módulo. Os métodos *build*, *deploy* e *satisfy* são métodos abstratos. A classe *CWSDeploy* é uma classe auxiliar, cujas instancias representam CWS contidos no *mart*.

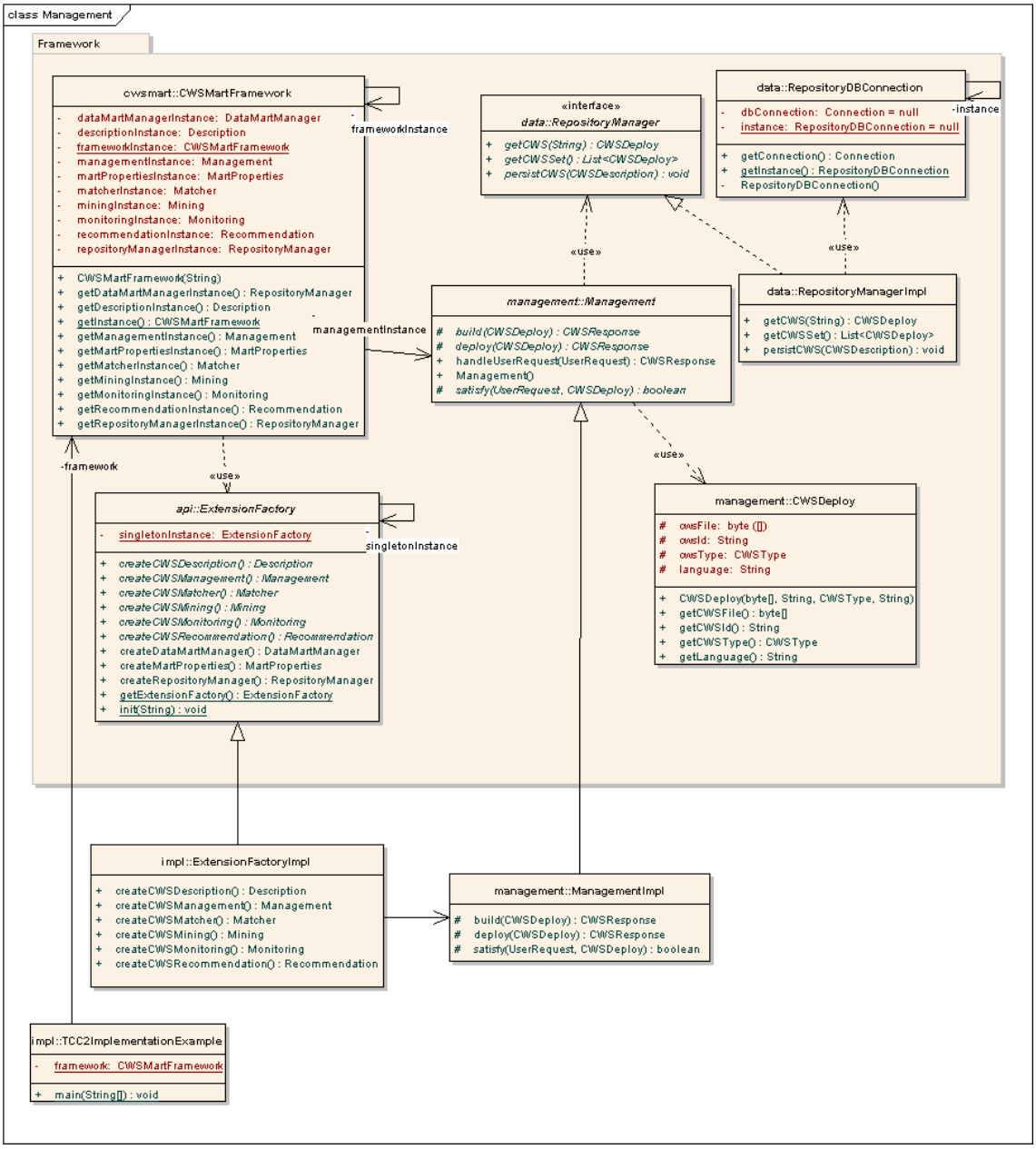

Figura 4.11: Diagrama de Classes: módulo *management*

A figura 4.12 ilustra o algoritmo associado ao módulo *management* e implementado no método *handleUserRequest*. A lista de CWS armazenados no repositório é percorrida e, uma vez encontrado um CWS que satisfaça a requisição do usuário, esse CWS é ou implantado diretamente se foi projetado proativamente ou construído e então implantado se foi projetado reativamente. Os métodos *satisfy* que verifica se o CWS satisfaz os requisitos funcionais e não funcionais da requisição do usuário; *deploy* que ativa a execução do CWS e *build* que procura, seleciona e executa os serviços componentes que fazem parte de CWS são métodos adicionais utilizados pelo método *handleUserRequest.* Eles são definidos como abstratos no *framework* e devem ser implementados pela aplicação que implementa este módulo.

Func handleUserRequest(*UserRequest*)
Input: *UserRequest* – requisição a satisfazer do usuário
Output: *CWSResponse* – resposta retornada ao usuário
Auxiliary: i – integer
Auxiliary: n – número de *CWSMarts* existentes, do tipo integer
Begin
for i = 0 *to n* **do**
if satisfy(*UserRequest, CWS*$_i$) **then**
return *idCWS*$_i$
- O serviço Web composto *CWS*$_i$ satisfaz a requisição do usuário
- O laço é interrompido

end if
endfor
if *idCWS*$_i \neq$ *null* **then**
if *type(idCWS*$_i$*) = proactive* **then**
deploy(*idCWS*$_i$)
- Executa o service Web composto

else
build(*idCWS*$_i$)
- Procura e seleciona serviços Web componentes
- Resolve conflitos entre serviços Web componentes
- Executa o serviço Web composto

end if
else return *null*
- Não há um serviço Web composto que pode satisfazer a requisição do usuário

End

Figura 4.12: Algoritmo associado com o módulo *management* (MAAMAR et al., 2010)

4.5.5 Módulo *mining*

O módulo *mining* fornece as ferramentas para busca e análise de padrões recorrentes entre as especificações dos CWS e entre as suas execuções. Exemplos de padrões são serviços Web componentes frequentemente usados, parâmetros frequentemente passados, tempos de execução dos CWSs do repositório assim como de serviços Web componentes integrantes da composição etc. Este módulo está descrito em mais detalhes na seção 4.2.

A figura 4.13 ilustra o diagrama de classes associado com este módulo. A classe principal, chamada de *Mining*, fornece dois métodos públicos. O método *addData* adiciona novos dados no *data mart*. Esses dados provêem do módulo *monitoring*. O método *mine* recupera informações do *data mart* de acordo com um conjunto de restrições passadas por parâmetro. Para criar uma ou mais restrições existe a classe *MiningOperation*. Uma instancia dessa classe contém uma lista de dimensões do *data mart* e as restrições desejadas sobre cada dimensão.

A figura ilustra ainda, além do *RepositoryManager*, um segundo componente de software interno ao *framework* e responsável por prover o acesso a base de dados do *data mart* chamado de *DataMartManager*.

Para ter acesso a esse módulo é necessário estender a classe *Mining* e implementar o método abstrato *mine*.

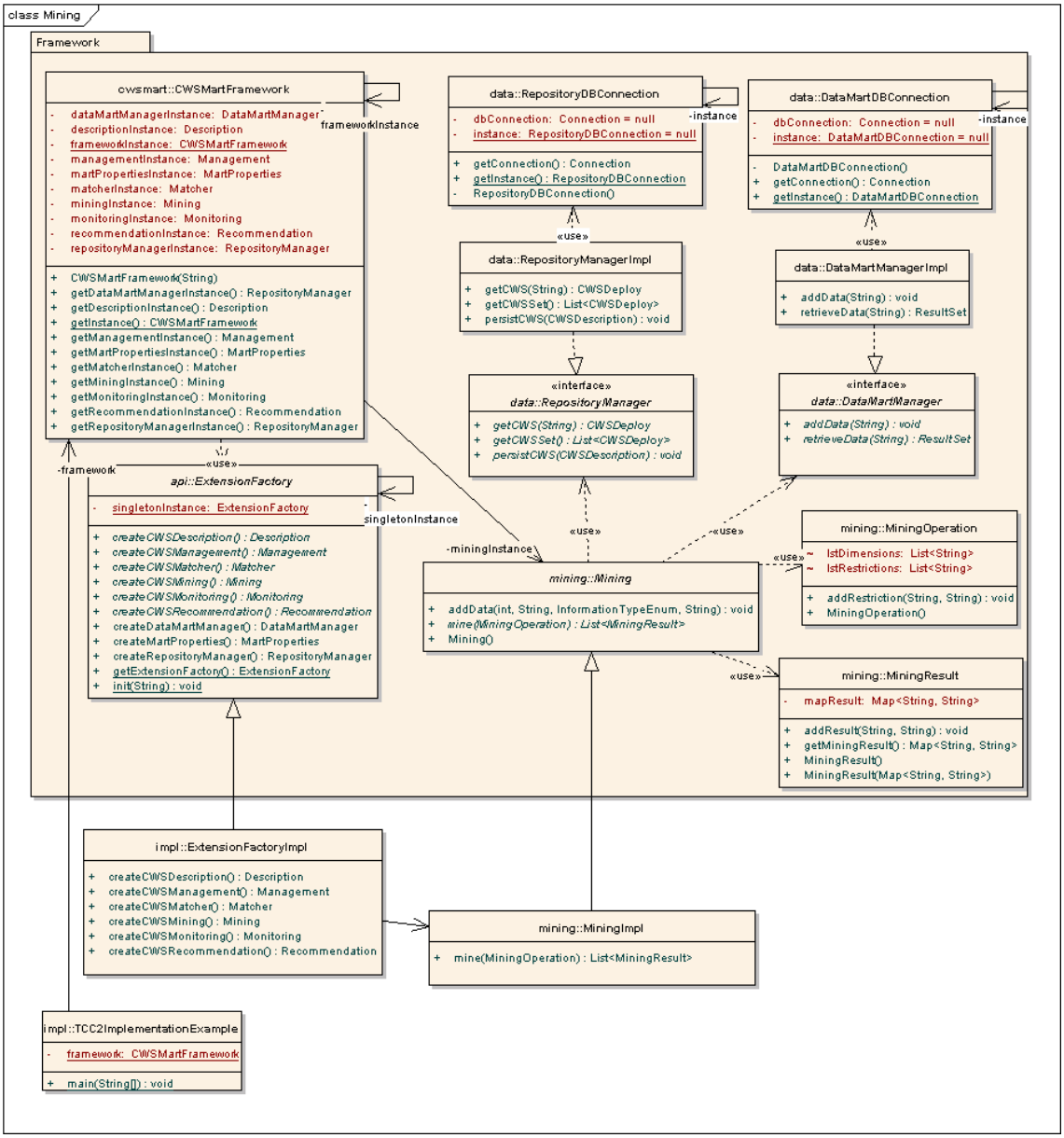

Figura 4.13: Diagrama de Classes: módulo *mining*

4.5.6 Módulo *monitoring*

O principal objetivo do módulo *monitoring* é coletar dados sobre a execução de CWSs no *CWSMart*. Informações como tempo de execução, se o serviço executou com sucesso ou se foram lançadas exceções, se algum serviço componente é frequentemente um gargalo na composição, entre outras, são importantes para o administrador do *CWSMart* tomar decisões.

Há pelo menos duas abordagens possíveis para permitir o desenvolvimento do módulo *monitoring*. A primeira é assumindo que os sistemas monitorados estão cientes desse fato e cooperam no processo informando os dados necessários. Por exemplo, ao iniciar a sua execução o CWS pode enviar uma mensagem para o módulo *monitoring* informando este evento. A segunda possibilidade é não ter CWSs cientes do

monitoramento. A obtenção das informações pode vir da análise dos logs de execução dos CWSs.

A figura 4.14 mostra o diagrama de classes associado com o módulo *monitoring*. A classe *Monitoring* declara o método abstrato *monitor*. Esse método deve ser implementado pela aplicação que estender este módulo, de modo que a melhor abordagem quanto a forma de monitoração é uma decisão da aplicação. Estudos futuros podem definir uma forma padrão de monitoramento nos *CWSMart* ou mesmo implementar as duas formas discutidas e disponibilizar apenas a escolha para a aplicação.

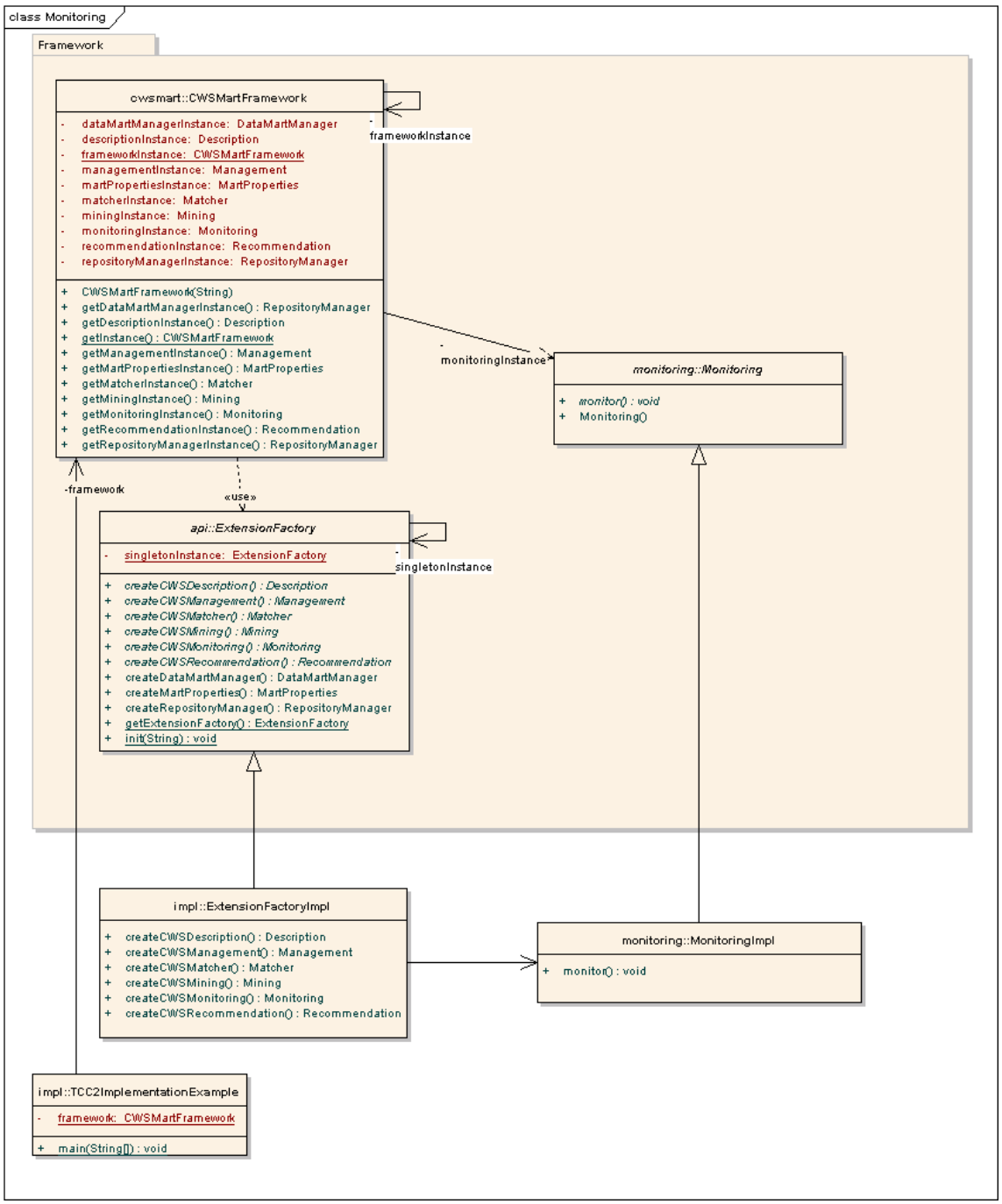

Figura 4.14: Diagrama de Classes: módulo *monitoring*

4.5.7 Documentação do Framework

Um *framework* é um componente de software reutilizável. Sendo assim, é fundamental que ele seja bem documentado a fim de que outros programadores possam entender o que e como fazer para usá-lo.

Cada método definido no *framework* desenvolvido, seja nas classes, classes abstratas ou interfaces, foi devidamente comentado com sua respectiva funcionalidade, parâmetros, tipo de retorno e exceções que podem ser lançadas. Ao final do desenvolvimento foi gerada uma documentação de referência rápida com auxilio da ferramenta *javadoc* da Sun (disponível em http://java.sun.com/j2se/javadoc/) capaz de gerar documentação de APIs em formato HTML a partir de comentários no código fonte.

O código fonte deste projeto está gerenciado com o auxílio do sistema de controle de versões *Subversion* (http://tortoisesvn.tigris.org/). Todo o projeto, incluindo código fonte e documentação, está disponível publicamente em http://code.google.com/p/cwsmart-framework/.

4.5.8 Aplicação Exemplo utilizando o Framework

Para ajudar a validar o *framework* desenvolvido, foi implementada uma aplicação simplificada utilizando a estrutura proposta. A seguir, serão destacados os principais detalhes envolvidos no desenvolvimento dessa aplicação exemplo.

Além da "fábrica" concreta de módulos, e de uma classe principal que instancia e utiliza o *framework*, foram criados três módulos, são eles: *matcher*, *description* e *recommendation* através da extensão das classes e implementação dos métodos necessários de acordo com o descrito nas seções anteriores.

A figura 4.15 ilustra um diagrama das classes implementadas nesta aplicação. A classe *TCC2ImplementationExample* instancia o *framework* passando como parâmetro a classe que implementa a fábrica concreta:

```
CWSMartFramework framework = new
    CWSMartFramework("br.ufrgs.inf.cwsmart.impl.ExtensionFactoryImpl");
```

A partir daí tem-se acesso aos módulos do *framework* e pode-se usar as funcionalidades providas por cada módulo. Por exemplo, para candidatar um novo serviço Web composto para ingresso no *CWSMart* usa-se o módulo *matcher*, chamando o método *joinCWSMart* passando como parâmetros a descrição do CWS, a linguagem da descrição, e as descrições wsdl relacionadas ao CWS:

```
framework.getMatcherInstance().joinCWSMart("Travel.bpel", "bpel",
"Travel.wsdl", "Airline.wsdl", "Employee.wsdl");
```

A linguagem de composição de serviços Web suportada na aplicação exemplo foi a BPEL. O *parsing* da descrição BPEL, realizado no módulo *matcher* para obter a funcionalidade do CWS e no módulo *description* para obter os atributos especificados na seção 4.3, foi realizado com o auxílio da biblioteca desenvolvida em *Java* chamada *Verbus*, versão 0.1.7a (disponível em http://www.it.uc3m.es/jaf/verbus/).

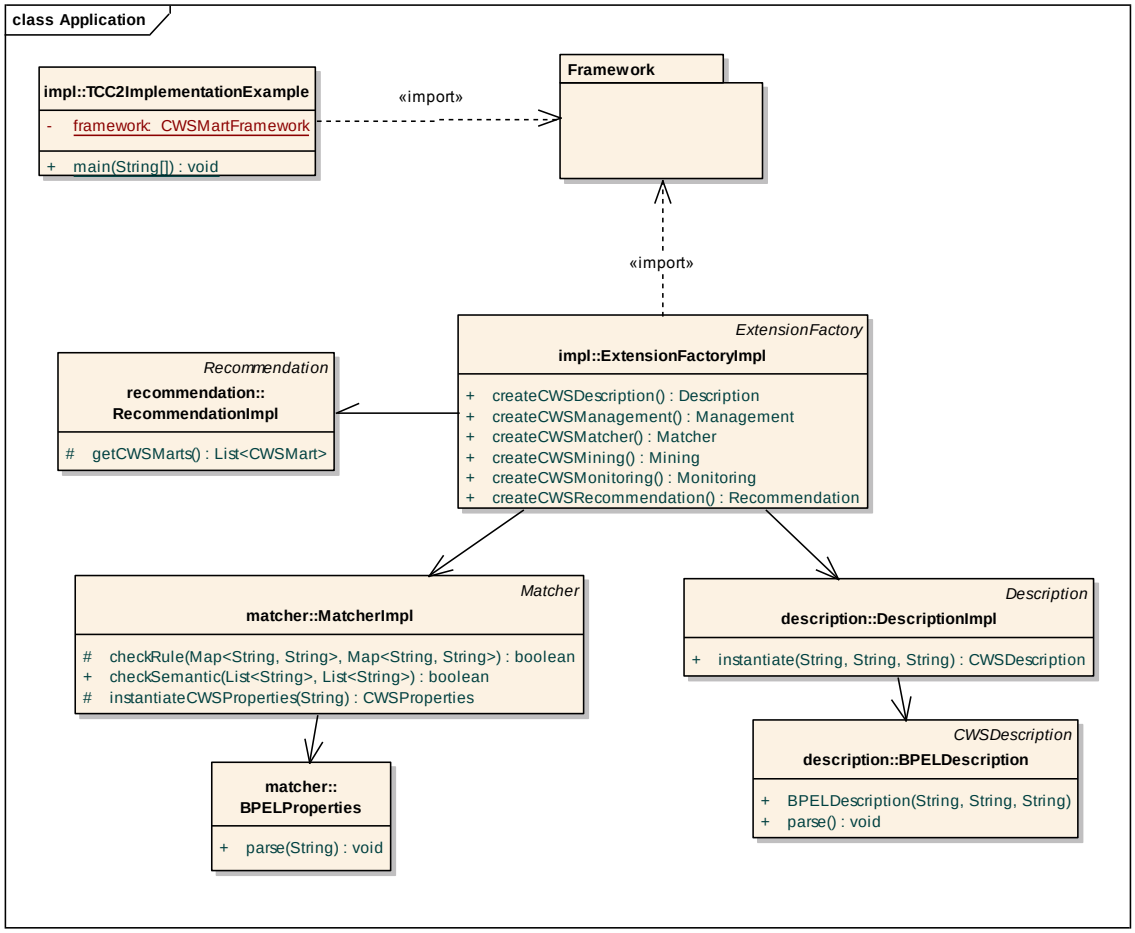

Figura 4.15: Diagrama de classes da aplicação exemplo

A funcionalidade do CWS submetido para o ingresso no *mart*, necessária no método *checkSemantic* do módulo matcher, foi obtida através do atributo *myRole* do elemento *PartnerLink* na seção *PartnerLinks* da descrição BPEL do serviço composto. Como, na descrição, pode haver vários elementos *PartnerLink*, o elemento escolhido foi o *PartnerLink* definido como o "parceiro" na primitiva BPEL *receive*, ou seja, buscou-se o atributo *myRole* da relação cliente-serviço Web composto sobre o ponto de vista do serviço composto:

```
//getting the bpel cws functionality
Parser parser = new Parser(cwsPath);
Activity activity = parser.getProcess().getActivity();
if(activity != null &&
   BPWSUtil.getActivityType(activity) == BPWSUtil.ACT_SEQUENCE) {
  Sequence sequence = (Sequence) activity;
  List<Activity> lstAct = (List<Activity>) sequence.getActivities();
  for (Activity act : lstAct) {
    if (BPWSUtil.getActivityType(act) == BPWSUtil.ACT_RECEIVE) {
      String cwsRole = ((Receive) act).getPartnerLink().getMyRole();
      this.lstFunctionality.add(cwsRole);
    }
  }}
```

O módulo *recommendation* utilizou um repositório global para tomar conhecimento sobre a existência de outros *CWSMart*. As informações contidas nesse repositório são

pares do tipo chave (identificador de um *CWSMart*) e valor (funcionalidade do *CWSMart*). Não se entrou no mérito de como esse repositório é atualizado, apenas se assumiu a sua existência e a corretude de seus dados.

Adicionalmente, foram definidos os dois arquivos de configuração utilizados pelo *framework*, a saber: *mart.config* e *mart.properties*. No primeiro é especificada a string e conexão da base de dados utilizada pelo módulo *description* para a persistência de informações no repositório. Nesse mesmo arquivo deve-se informar a string de conexão da base de dados do *data mart*, de modo que o módulo *mining* possa ter acesso a essa informação. No segundo arquivo é especificada a funcionalidade do *mart* e as regras que controlam a sua "população". As informações do segundo arquivo são utilizadas pelo módulo *matcher* nas decisões relacionadas a hospedagem ou não de CWS candidatos.

Com esta aplicação foi possível testar não só uma das operações previstas para o *CWSMart* que é a operação de inclusão de novos serviços Web compostos no *mart*, considerando todo o caminho percorrido pelo CWS seja quando ele é hospedado no repositório ou quando ele recebe uma recomendação para tentar o ingresso em outro *mart* mas também o funcionamento geral do *framework*, a independência dos módulos no domínio da aplicação, o modo como as classes do *framework* interagem com as classes da aplicação e o fluxo de execução comandado pelo *framework* e complementado por algoritmos customizados.

5 CONCLUSÕES

O trabalho realizado envolveu o estudo do estado da arte no nível de *CWSMarts* englobando questões do tipo como desenvolver serviços Web compostos prontos para uso, como comparar serviços Web compostos uns com os outros, como recomendar serviços Web compostos e que permanecem sem resposta de modo que soluções pontuais, caso a caso, são propostas. Este trabalho contribui com a área na medida em que propõe um *framework* para *CWSMarts*, possibilitando uma estrutura básica para a gerência do conteúdo de um *CWSMart* em termos de atração e eliminação de serviços Web compostos, da comparação de serviços Web compostos contidos no mesmo *CWSMart*, de recomendação de serviços Web compostos e ao possibilitar consultas e análises baseadas no conteúdo e no funcionamento do *CWSMart*.

Como a tecnologia de *CWSMarts* não existe, algumas dificuldades foram encontradas no sentido de que algumas coisas tiveram que ser definidas ou assumidas em função de não existir uma referência na literatura. Como exemplo, pode ser citada a forma como as informações referentes às execuções dos CWS deveriam ser persistidas. Nesse caso foi proposto um modelo de *data mart* capaz de guardar informações importantes sobre cada execução de um CWS, e com isso permitir tomadas de decisões em relação ao gerenciamento do *CWSMart* baseado em análises de comportamentos e tendências identificadas no passado. A vantagem desse fato é que se abre espaço para o planejamento de novas soluções o que sempre é um desafio e na maioria das vezes a parte mais interessante.

5.1 Resumo de Resultados

Projetar um *framework* é muito mais complicado que projetar uma única aplicação, porque é preciso conhecer muito sobre o domínio do problema para poder antecipar o que os desenvolvedores que utilizarão o *framework* esperam encontrar.

Baseando-se na aplicação exemplo, verificou-se que o *framework* respondeu bem as expectativas, no sentido de que a implementação de cada módulo pôde ser feito de maneira independente dos demais, o que satisfez um dos objetivos do *framework* que é possibilitar o reuso de software para trabalhos futuros focados especificamente em cada um dos módulos do *CWSMart.* Foi possível também estender o *framework* de modo a comportar serviços Web compostos descritos em BPEL, assim como a customizar parâmetros, tais como funcionalidade do *mart*, regras para aceitação de novos integrantes etc.; e algoritmos como de *parsing* e de *matching*.

O projeto, composto por diagramas UML, mostrou-se importante para a concretização do trabalho. Em um *framework*, onde existe uma grande quantidade de classes e interfaces, que se relacionam ainda com outras implementadas em aplicações

especificas definidas posteriormente, a construção de um bom projeto, baseado em padrões de projeto, quando possível, se mostra fundamental para um bom produto final.

5.2 Limitações do Trabalho

Os módulos *Agent* e *Garbagge Collector* não foram incluídos na implementação do *framework*. No entanto, considerando o modo como o *framework* foi construído, não deve haver problemas na inclusão desses dois módulos futuramente.

Uma das funcionalidades dos *CWSMarts* é recomendar outro *CWSMart* para o serviço Web composto que foi recusado no *mart* em questão. A questão de como tornar o módulo *recommendation*, responsável por essa funcionalidade, ciente da existência de outros *marts*, ou seja, fornecer os meios para que ele possa pesquisar em um repositório de *CWSMarts*, local ou global, não foi desenvolvida neste trabalho.

O protótipo desenvolvido usando o *framework* usou apenas uma parte do *framework*, ou seja, apenas três módulos foram estendidos e utilizados. O projeto do *framework*, avaliando sob a perspectiva de extensão, de qualquer forma consegue ser avaliado satisfatoriamente bem uma vez que o protótipo foi bem sucedido ao que se propôs, no entanto, o projeto interno dos demais módulos carece de testes, com o desenvolvimento de uma aplicação para podermos avaliar o seu funcionamento.

5.3 Perspectivas

Existe muito campo para trabalhos futuros na área de *CWSMarts* e no *framework* desenvolvido. Cada módulo implementado no *framework* pode ser estudado em mais profundidade. O *framework* fornece a base para que isso seja possível com o menor nível de esforço possível. O módulo *monitoring*, por exemplo, é foco de um estudo em nível de trabalho de graduação por outro aluno orientado pelo professor Leandro Wives em andamento no Instituto de Informática da UFRGS. Outro exemplo é o módulo *recommendation* que precisa ser estudado quanto a melhor forma de torná-lo ciente da existência de outros *CWSMarts*, se através de um repositório local, onde cada *CWSMart* comunica aos outros *CWSMarts* parceiros a respeito de atualizações importantes ou se através de um repositório global mantido por um administrador ou mesmo pelos próprios *marts*. Além disso, os módulos que não entraram no projeto do *framework* precisam ser incorporados ao *framework*, são eles: *Agent* e *Garbagge Collector*.

Por fim, segundo Horstmann (2007), uma boa regra para validar o projeto de um *framework* é usá-lo para construir pelo menos três aplicações diferentes. Sendo assim, o desenvolvimento de novas aplicações que utilizem o *framework* tende a gerar refinamentos de modo que ele atingirá um nível mais maduro após esses possíveis ajustes.

REFERÊNCIAS BIBLIOGRÁFICAS

BENATALLAH, B.; SHENG, Q. Z.; DUMAS, M. **The Self-Serv Environment for Web Services Composition**. IEEE Internet Computing, 7(1), Janeiro/Fevereiro 2003.

BUI, T.; GACHER, A. **Web Services for Negotiation and Bargaining in ElectronicMarkets: Design Requirements and Implementation Framework.** In *Proceedings of the 38th Hawaii International Conference on System Sciences (HICSS'2005)*, Big Island, Hawaii, USA, 2005.

CHAKRABORTY, D.; JOSHI, A. **Dynamic Service Composition: State-of-the-Art and Research Directions**. Technical report, TR-CS-01-19, Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, Maryland, USA, 2001.

CHAN, K. S. **Towards a framework for web service compositions recovery**. In *Proceedings of the Warm Up Workshop For ACM/IEEE ICSE 2010* (Cape Town, South Africa, April 01 - 03, 2009). N. Medvidovic and T. Tamai, Eds. WUP '09. ACM, New York, NY, 17-20. DOI= http://doi.acm.org/10.1145/1527033.1527040

CHOLLET, S.; LALANDA, P. **An Extensible Abstract Service Orchestration Framework**. In *Proceedings of the 2009 IEEE international Conference on Web Services - Volume 00* (July 06 - 10, 2009). ICWS. IEEE Computer Society, Washington, DC, 831-838. DOI= http://dx.doi.org/10.1109/ICWS.2009.14

CHUKMOL, U. **A framework for web service discovery: service's reuse, quality, evolution and user's data handling**. In *Proceedings of the 2nd SIGMOD PhD Workshop on innovative Database Research* (Vancouver, Canada, June 13 - 13, 2008). IDAR '08. ACM, New York, NY, 13-18. DOI= http://doi.acm.org/10.1145/1410308.1410313

CURBERA, F.; GOLAND, Y.; KLEIN J. et al. **Business Process Execution Language for Web Services (BPEL4WS) Version 1.1,** 2003.

Docentes de Sistemas Distribuídos, DEI, IST, UTL. **UDDI**. Universidade Técnica de Lisboa. Disponível em: < http://disciplinas.ist.utl.pt/leic-sod/2009-2010/labs/06-ws-registry/uddi/index.html >. Acesso em: Maio. 2010.

GAMMA, E. et al. **Padrões de Projeto**: Soluções Reutilizáveis de Software Orientado a Objetos. Porto Alegre: Bookman, 2000. 364p.

HARMON, P. **Analyzing Activities, Business Process Trends.** vol. 1, no. 4, pp. 1-12, Abril 2003.

History of GUI Frameworks and some thoughts. Disponível em: <http://www.cincomsmalltalk.com/userblogs/pollock/blogView?showComments=true&entry=3241191126>. Acesso em Novembro, 2009.

HORSTMANN, C. **Padrões e Projeto Orientados a Objetos.** 2nd ed. Porto Alegre: Bookman, 2007.

ITALIANO, I. C.; ESTEVES, L. A. **Modelagem de Data Warehouse e Data Mart**. SQL Magazine, Grajaú, ano 2, Ed. 14, p. 37-43, [s.d.]

JURIC, M.B. **A Hands-on Introduction to BPEL**. Disponível em: <http://www.oracle.com/technology/pub/articles/matjaz_bpel1.html>. Acesso em: Maio, 2010.

KAVANTZAS, N.; BURDETT, D.; RITZINGER, G. et al. **Web Services Choreography Description Language Version 1.0**. W3C Candidate Recommendation, 2005. http://www.w3.org/TR/ws-cdl-10/.

KIMBALL, R. **Data Warehouse Toolkit.** São Paulo: Makron Books, 1998.

KODALI, R. **What is service-oriented architecture?** Disponível em: <http://www.javaworld.com/javaworld/jw-06-2005/jw-0613-soa.html>. Acesso em: Agosto, 2009.

MAAMAR, Z.; BENSLIMANE, D. et al. **A Multi-Layer and Multi-Perspective Approach to Compose Web Services**. 21st International Conference on Advanced Information Networking and Applications (AINA 2007). Niagara Falls, Canada: IEEE Computer Society: 31-37 p. 2007.

MAAMAR, Z.; LAHKIM, M.; BENSLIMANE, D. et al. **Web Services Communities - Concepts & Operations**. In Proceedings of the 3rd International Conference on Web Information Systems and Technologies (WEBIST'2007), Barcelona, Espanha, 2007.

MAAMAR, Z.; M'BARECK, N. O. A.; TATA, S. **Towards An Approach for Enhancing Web Services Discovery**. In Proceedings of the 1st International Workshop on Information Systems and Web Services (ISWS'2007) held in conjunction with The 16 IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE'2007).

MAAMAR, Z.; WIVES, L. K.; SHENG, Q. Z. et al. **From Communities of Web Services to Marts of CompositeWeb Services.** In: Advanced Information Networking and Applications, AINA 2010. Australia, 2010. (submetido).

MACHADO, F. N. R. **Projeto de Data Warehouse: Uma Visão Multidimensional**. São Paulo: Érica, 2000. 248 p.

MEDJAHED, B.; BOUGUETTAYA, A. **A Dynamic Foundational Architecture for Semantic Web Services**. Distributed and Parallel Databases, Kluwer Academic Publishers, 17(2), Março 2005.

OASIS UDDI Specifications. Disponível em: < http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm>. Acesso em: Novembro, 2009.

PAPAZOGLOU, M.P. **Web Services: Principles and Technology**, Prentice Hall, 2008.

PARRINI, E. **Gestão do Conhecimento no Suporte à Decisão em Ambiente OLAP**. Rio de Janeiro: 2002. 157 p. Dissertação (Mestrado em Informática) – Instituto de

Matemática, Núcleo de Computação Eletrônica – NCE, Universidade Federal do Rio de Janeiro, Rio de Janeiro.

PETKOVIC, D. **SQL Server 2000 – Guia Prático**. São Paulo: Makron Books, 2001.

PRE, W. **Design Patterns for Object-Oriented Software Development**. New York-USA: Addison-Wesley Publishing Company, 1995. 268p.

SILVA, R. P. **Suporte ao desenvolvimento e Uso de frameworks e componentes**. Porto Alegre: 2000. 262p. Tese (Doutorado em Ciência da Computação) – Programa de Pós-Graduação em Computação, UFGRS, 2000. Disponível em: <http://www.inf.ufsc.br/~ricardo/download/tese.pdf>. Acesso em: 25 nov. 2009.

SPENIK, M.; SLEDGE, O. **Microsoft SQL Server 2000 DBA: Guia de sobrevivência.** Rio de Janeiro: Campus, 2001. 773 p.

THATTE, T. **XLANG: Web Services for Business Process Design**, 2001. http://www.gotdotnet.com/team/xml wsspecs/xlang-c/default.htm.

Wikipedia, The Free Encyclopedia. **WSDL: Web Services Description Language**. Disponível em: < http://en.wikipedia.org/wiki/Web_Services_Description_Language >. Acesso em: Novembro, 2009.

WIVES, L. K. **Dodona: Recomendação de Serviços Web**. Descrição do Projeto de Pesquisa, 2008.

World Wide Web Consortium. **W3C Note 15 March 2001**. **Web Services Description Language (WSDL) 1.1**. Disponível em: < http://www.w3.org/TR/wsdl >. Acesso em: Agosto. 2009.

World Wide Web Consortium. **W3C Recommendation (Second Edition) 27 April 2007. SOAP Version 1.2**. Disponível em: < http://www.w3.org/TR/soap12-part1 >. Acesso em: Agosto. 2009.

World Wide Web Consortium. **Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language**. Disponível em: < http://www.w3.org/TR/wsdl20>. Acesso em: Maio. 2010.

ZUROWSKA, K.; DETERS, R. **Load management in model-aware execution of composite web services**. In Proceedings of the 2009 ACM Symposium on Applied Computing (Honolulu, Hawaii). SAC '09. ACM, New York, NY, 2134-2139. DOI= http://doi.acm.org/10.1145/1529282.1529754

APÊNDICE A DICIONÁRIO DE DADOS DA BASE DE DADOS DO REPOSITÓRIO

As tabelas abaixo contêm informações dos campos, tipo de dados, tamanho e conteúdo para as tabelas do modelo de dados relacional do repositório de *CWSMarts* com o objetivo de formar um dicionário de dados para este modelo.

Tabela A.1: Dicionário de dados da tabela CWS

Nome da Coluna	***Tipo de dado***	***Comentário***
id_cws	varchar(100)	Chave primária. Identificador de um CWS
language	varchar(50)	Linguagem na qual o CWS está especificado
cws_path	varchar(100)	Caminho onde a especificação está armazenada
wsdl_path	varchar(100)	Caminho onde a descrição WSDL do CWS está armazenada.
type	smallint	Chave estrangeira. Indica o tipo do CWS

Tabela A.2: Dicionário de dados da tabela CWS_TYPE

Nome da Coluna	***Tipo de dado***	***Comentário***
id	smallint	Chave primária
name	varchar(50)	Nome de um tipo de CWS: *proativo, reativo*
description	text	Descrição de um tipo de CWS. Atributo opcional

Tabela A.3: Dicionário de dados da tabela COMPONENT_WS

Nome da Coluna	***Tipo de dado***	***Comentário***
id_ws	varchar(100)	Chave primária. Identificador de um WS
wsdl_path	varchar(100)	Caminho onde a descrição WSDL do WS está armazenado. Atributo opcional.

Tabela A.4: Dicionário de dados da tabela CWS_COMPONENT

Nome da Coluna	***Tipo de dado***	***Comentário***
id_cws	varchar(100)	Chave primária e chave estrangeira. Referencia o identificador de um CWS.
id_component	varchar(100)	Chave primária e chave estrangeira. Referencia o identificador de um WS.
type	smallint	Chave estrangeira. Indica o “papel” do WS componente na composição

Tabela A.5: Dicionário de dados da tabela COMPONENT_TYPE

Nome da Coluna	***Tipo de dado***	***Comentário***
id	smallint	Chave primária
name	varchar(50)	Nome do "papel" do WS componente na composição: *core, opcional*
description	text	Descrição do "papel" desempenhado. Atributo opcional

Tabela A.6: Dicionário de dados da tabela REQUIREMENT

Nome da Coluna	***Tipo de dado***	***Comentário***
id_requirement	smallint	Chave primária
name	varchar(50)	Nome de um requerimento
description	text	Descrição de um requerimento. Atributo opcional.
type	smallint	Chave estrangeira. Tipo do requerimento

Tabela A.7: Dicionário de dados da tabela CWS_REQUIREMENT

Nome da Coluna	***Tipo de dado***	***Comentário***
id_cws	varchar(100)	Chave primária e chave estrangeira. Referencia o identificador de um CWS.
id_requirement	smallint	Chave primária e chave estrangeira. Referencia o identificador de um requerimento

Tabela A.8: Dicionário de dados da tabela REQUIREMENT_TYPE

Nome da Coluna	***Tipo de dado***	***Comentário***
id	smallint	Chave primária
name	varchar(50)	Nome de um tipo de requerimento: funcional, não-funcional
description	text	Descrição de um tipo de requerimento. Atributo opcional.

APÊNDICE B DICIONÁRIO DE DADOS DA BASE DO DATA MART

As tabelas abaixo apresentam um dicionário de dados contendo informações de tipo de dados, tamanho e conteúdo dos campos das tabelas do modelo de dados do *data mart* para *CWSMarts*.

Tabela B.1: Dicionário de dados da tabela F_EXECUTION

Nome da Coluna	***Tipo de dado***	***Comentário***
id_fact	bigint	Chave primária
execution_time	number	Tempo de execução do CWS
activated_path	varchar(100)	Caminho da especificação ativado na execução
execution_sucess	boolean	Identifica se a execução ocorreu com sucesso
failed_component	varchar(100)	Identificador do componente que falhou
exception	text	Detalhes da exceção lançada

Tabela B.2: Dicionário de dados da tabela D_PARAMETER

Nome da Coluna	***Tipo de dado***	***Comentário***
id_parameter	bigint	Chave primária
parameter_number	smallint	Número de parâmetros passado
parameters	varchar(200)	Valor dos parâmetros passados

Tabela B.3: Dicionário de dados da tabela D_STRUCTURE

Nome da Coluna	***Tipo de dado***	***Comentário***
id_structure	bigint	Chave primária
language	varchar(50)	Linguagem na qual o CWS está especificado
cws	varchar(100)	Identificador do CWS executado
component_number	smallint	Número de WS componentes
conditional_structs_number	integer	Número de estruturas condicionais na especificação do CWS
loop_structs_number	integer	Número de estruturas de “laços” na especificação do CWS

Tabela B.4: Dicionário de dados da tabela D_TIME

Nome da Coluna	***Tipo de dado***	***Comentário***
id_time	bigint	Chave primária
year	smallint	Ano em que a execução do CWS ocorreu
month	smallint	Mês em que a execução do CWS ocorreu
day	smallint	Dia em que a execução do CWS ocorreu
datetime	timestamp	Data e hora da execução do CWS

Tabela B.5: Dicionário de dados da tabela D_REQUEST_SOURCE

Nome da Coluna	***Tipo de dado***	***Comentário***
id_source	bigint	Chave primária
type	smallint	Identificador do tipo de solicitador da execução do CWS
name	varchar(50)	Nome do tipo de solicitador: *user*, *web service*

Tabela B.6: Dicionário de dados da tabela D_PROVIDER

Nome da Coluna	***Tipo de dado***	***Comentário***
id_provider	bigint	Chave primária
name	varchar(50)	Nome do provedor CWS
country	varchar(50)	País do provedor do CWS

ANEXO A DESCRIÇÃO WSDL DO PROCESSO BPEL PARA VIAGENS

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:emp="http://packtpub.com/service/employee/"
        xmlns:aln="http://packtpub.com/service/airline/"
        xmlns:tns="http://packtpub.com/bpel/travel/"
        targetNamespace="http://packtpub.com/bpel/travel/"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/" >

<import namespace="http://packtpub.com/service/employee/" location="./Employee.wsdl"/>
<import namespace="http://packtpub.com/service/airline/" location="./Airline.wsdl"/>

<message name="TravelRequestMessage">
  <part name="employee" type="emp:EmployeeType" />
  <part name="flightData" type="aln:FlightRequestType" />
</message>

<portType name="TravelApprovalPT">
  <operation name="TravelApproval">
    <input message="tns:TravelRequestMessage" />
  </operation>
</portType>

<portType name="ClientCallbackPT">
  <operation name="ClientCallback">
    <input message="aln:TravelResponseMessage" />
  </operation>
</portType>

<!-- Partner link type -->
<plnk:partnerLinkType name="travelLT">
  <plnk:role name="travelService">
    <plnk:portType name="tns:TravelApprovalPT" />
  </plnk:role>
  <plnk:role name="travelServiceCustomer">
    <plnk:portType name="tns:ClientCallbackPT" />
  </plnk:role>
</plnk:partnerLinkType>
</definitions>
```

ANEXO B PROCESSO BPEL PARA VIAGENS

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Asynchrnous BPEL process -->
<process name="BusinessTravelProcess"
        targetNamespace="http://packtpub.com/bpel/travel/"
        xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
        xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
        xmlns:trv="http://packtpub.com/bpel/travel/"
        xmlns:emp="http://packtpub.com/service/employee/"
        xmlns:aln="http://packtpub.com/service/airline/" >

  <partnerLinks>
    <partnerLink name="client" partnerLinkType="trv:travelLT" myRole="travelService"
                 partnerRole="travelServiceCustomer"/>
    <partnerLink name="employeeTravelStatus"  partnerLinkType="emp:employeeLT"
                 partnerRole="employeeTravelStatusService"/>
    <partnerLink name="AmericanAirlines"   partnerLinkType="aln:flightLT"
                 myRole="airlineCustomer"  partnerRole="airlineService"/>
    <partnerLink name="DeltaAirlines"  partnerLinkType="aln:flightLT"
                 myRole="airlineCustomer"                partnerRole="airlineService"/>
  </partnerLinks>

  <variables>
    <!-- input for this process -->
    <variable name="TravelRequest" messageType="trv:TravelRequestMessage"/>
    <!-- input for the Employee Travel Status web service -->
    <variable name="EmployeeTravelStatusRequest"
              messageType="emp:EmployeeTravelStatusRequestMessage"/>
    <!-- output from the Employee Travel Status web service -->
    <variable name="EmployeeTravelStatusResponse"
              messageType="emp:EmployeeTravelStatusResponseMessage"/>
    <!-- input for American and Delta web services -->
    <variable name="FlightDetails" messageType="aln:FlightTicketRequestMessage"/>
    <!-- output from American Airlines -->
    <variable name="FlightResponseAA" messageType="aln:TravelResponseMessage"/>
    <!-- output from Delta Airlines -->
    <variable name="FlightResponseDA" messageType="aln:TravelResponseMessage"/>
    <!-- output from BPEL process -->
    <variable name="TravelResponse" messageType="aln:TravelResponseMessage"/>
  </variables>

  <sequence>
    <!-- Receive the initial request for business travel from client -->
    <receive partnerLink="client" portType="trv:TravelApprovalPT"
             operation="TravelApproval" variable="TravelRequest" createInstance="yes" />

    <!-- Prepare the input for the Employee Travel Status Web Service -->
```

```
<assign>
  <copy>
    <from variable="TravelRequest" part="employee"/>
    <to variable="EmployeeTravelStatusRequest" part="employee"/>
  </copy>
</assign>

<!-- Synchronously invoke the Employee Travel Status Web Service -->
<invoke partnerLink="employeeTravelStatus" portType="emp:EmployeeTravelStatusPT"
            operation="EmployeeTravelStatus"
            putVariable="EmployeeTravelStatusRequest"
            outputVariable="EmployeeTravelStatusResponse" />

<!-- Prepare the input for AA and DA -->
<assign>
  <copy>
    <from variable="TravelRequest" part="flightData"/>
    <to variable="FlightDetails" part="flightData"/>
  </copy>
  <copy>
    <from variable="EmployeeTravelStatusResponse" part="travelClass"/>
    <to variable="FlightDetails" part="travelClass"/>
  </copy>
</assign>

<!-- Make a concurrent invocation to AA in DA -->
<flow>
  <sequence>
    <!-- Async invoke of the AA web service and wait for the callback -->
    <invoke partnerLink="AmericanAirlines" portType="aln:FlightAvailabilityPT"
              operation="FlightAvailability" inputVariable="FlightDetails" />
    <receive partnerLink="AmericanAirlines" portType="aln:FlightCallbackPT"
              operation="FlightTicketCallback" variable="FlightResponseAA" />
  </sequence>

  <sequence>
    <!-- Async invoke of the DA web service and wait for the callback -->
    <invoke partnerLink="DeltaAirlines" portType="aln:FlightAvailabilityPT"
              operation="FlightAvailability" inputVariable="FlightDetails" />
    <receive partnerLink="DeltaAirlines" portType="aln:FlightCallbackPT"
              operation="FlightTicketCallback" variable="FlightResponseDA" />
  </sequence>
</flow>

<!-- Select the best offer and construct the TravelResponse -->
<switch>
  <case
            condition="bpws:getVariableData('FlightResponseAA','confirmationData','/confir
            mationData/aln:Price')   <=
            bpws:getVariableData('FlightResponseDA','confirmationData','/confirmationData
            /aln:Price')">
     <!-- Select American Airlines -->
     <assign>
       <copy>
         <from variable="FlightResponseAA" />
         <to variable="TravelResponse" />
       </copy>
     </assign>
  </case>
```

```
        <otherwise>
          <!-- Select Delta Airlines -->
          <assign><copy>
             <from variable="FlightResponseDA" />
             <to variable="TravelResponse" />
           </copy></assign>
        </otherwise>
      </switch>

      <!-- Make a callback to the client -->
      <invoke partnerLink="client" portType="trv:ClientCallbackPT" operation="ClientCallback"
              inputVariable="TravelResponse" />
    </sequence>
  </process>
```