Desenvolvimento e avaliação de um registro de serviços de ferramentas de teste

Rodrigo Pinto Gondim

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 26 de maio de 2010

Assinatura: _____

Desenvolvimento e avaliação de um registro de serviços de ferramentas de teste

Rodrigo Pinto Gondim

Orientador: ***Prof. Dr. Paulo Cesar Masiero***

Monografia apresentada ao Instituto de Ciências Matemáticas e de Computação — ICMC/USP, como parte dos requisitos para obtenção do título de Mestre em Ciências de Computação e Matemática Computacional.

USP - São Carlos
Maio/2010

A minha irmã e a minha família.

Agracimentos

Agradeço muito a Deus por todos os momentos de esperança e por proteger e guiar minha vida até aqui.

A toda minha família pelo amor, apoio e auxílio na vinda para São Carlos. Em especial minha Vó, tia Dauzira, tio Edson e minha irmã, sem vocês o mestrado não seria possível.

Ao meu orientador Paulo Masiero, pela oportunidade, profissionalismo, incentivo, orientação e paciencia, sem o qual não teria realizado este trabalho.

Muito obrigado aos meus amigos do LabES e agregados: Van, Jorge Piu, Otávio, Rodolfo, Vânia, Marcelo, Endo, Abe, Paulo Ceará (Gambi), Sandro KLB, Mel, Erika, Nerso, Messias, Ana Carla, Marcão, Fabiano, Katia, Marllos, Sorin, David, Silvana, Maria, Lucas Bueno, Rafael (Frotinha), Adriano (Dj Alemão), Vinicius, Andrézinho, Bruno Cafeo, Draylson, Nardi, Bruno Feres, José Tim e as Profs. Ellen Francine, Rosana Braga e Simone Senger. Obrigado a todos!

Aos professores da UFMS e ICMC que contribuíram para minha formação, em especial, Marcelo Turine, Debora Paiva, Ronaldo Ferreira e Marcelo Siqueira que me incentivaram a fazer o mestrado.

Muito obrigado aos amigos da ABU pelo carinho e amizade: Lorena, Marcelo e Adriano.

Aos amigos de Campo Grande e da república Tereré em São Carlos pelos momentos de alegria e por tornar a estadia em São Carlos mais agradavel, Diogo, Mario, Kenji, Alex, Alessandro, Patrick, Letrícia, Márcio, Jucimara, Vanessa, Kish, Lucas, Ronaldo, Sanderson, Cesinha, Ivo, André, Pri, Debora e Eduardo. Obrigado de coração!

À Fapesp e CNPq pelo apoio financeiro.

Resumo

Um importante mecanismo da arquitetura orientada a serviços é o componente de registro de serviços. Ele permite a interação entre provedores e clientes, oferecendo um meio de acesso aos serviços desenvolvidos e publicados no registro. Nesta dissertação é proposto o desenvolvimento de um registro de serviços para dar apoio à publicação, busca e classificação de serviços Web, em particular, àqueles relacionados a ferramentas de teste de software. Uma limitação comum dos serviços de registro refere-se às buscas realizadas, pois são basicamente sintáticas e podem trazer resultados pouco relacionados aos interesses do usuário. Para resolver este problema uma ontologia de teste foi adaptada e incorporada ao registro com o objetivo de oferecer facilidades de busca e agregar informação semântica nos serviços registrados. Uma arquitetura genérica baseada em serviços para o domínio de engenharia de software é apresentada e instanciada para o domínio de teste de software com o objetivo de auxiliar no entendimento e implementação do registro de serviços proposto. Também são apresentados exemplos de ferramentas de teste publicadas no registro e um exemplo de busca e interação com o serviço de teste JaBUTiWS, previamente publicado no registro, que tem por objetivo apoiar o teste estrutural de componentes e serviços.

Abstract

An important mechanism of Service Oriented Architecture is the service registry (or service broker). It allows interaction among providers and consumers, offering a point to access the services developed and published in the registry. In this dissertation we propose the development of a service broker to support the publication, search and categorization of Web services, particularly those related to software testing tools. A common limitation of service brokers refers to searching facilities since they are primarily syntactic and thus can bring results that are not well related with the user's interest. To tackle this problem a test ontology was adapted and incorporated into the broker with the aim of improving the likelihood of finding the correct service in searches and also to add semantic information to the registered services. A generic service oriented architecture for the software engineering domain is presented and instantiated to the software testing domain with the purpose of facilitating the understanding and implementation of the proposed service registry. We also present some examples of software testing tools published in the registry and an example of search and interaction with the JaBUTiWS testing service, previously published in the registry, which aims to support structural testing of components and services.

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO

1

Introdução

1.1 Contexto e Motivação

Cada vez mais, grandes empresas e amplos sistemas distribuídos necessitam de flexibilidade. Ao mesmo tempo, os sistemas e processos computacionais estão tornando-se mais complexos. Nota-se que as grandes corporações possuem vários sistemas instalados, aplicações e arquiteturas, todas elas de diversas épocas, utilizando tecnologias e arquiteturas diferentes. Um dos problemas do desenvolvimento de sistemas refere-se a integrá-los de forma heterogênea a fim de manter a flexibilidade global em relação a mudanças do ambiente (Dan *et al.*, 2006; Josuttis, 2007). Nesse sentido, surge a necessidade de uma abordagem capaz de lidar com a heterogeneidade dos sistemas e também que ofereça uma natureza descentralizada.

Nesse contexto, a arquitetura orientada a serviços (*Service Oriented Architecture* – SOA) busca acelerar o processo de desenvolvimento focando principalmente no reúso e na interoperabilidade entre os serviços. Nessa abordagem, todas as funcionalidades são implementadas como serviços, os quais possuem interfaces bem definidas e podem ser solicitados remotamente por outras aplicações. Os serviços são semelhantes aos componentes e desempenham funções específicas para seus clientes, com a diferença de que não são integrados fisicamente à aplicação do cliente e sua evolução está sob o controle do fornecedor do serviço (Canfora e Penta, 2006). O uso de serviços como unidades básicas de construção, além de permitir o desenvolvimento

rápido e de baixo custo, quando implementado corretamente provê uma maneira fácil para a integração de aplicações.

O fato de os serviços não serem fisicamente integrados na aplicação introduz alguns desafios, principalmente relacionado ao teste de software, mais precisamente, o teste de componentes e serviços Web. Isso ocorre porque os serviços, semelhantemente aos componentes, também precisam ser testados antes de serem utilizados em uma aplicação. A atividade de teste de componentes e serviços possui algumas limitações. Em geral o código fonte não está disponível e nem informações internas que permitam a geração de casos de teste com base em critérios baseados na implementação.

Além disso, Canfora e Penta (2006) afirmam que a definição de métodos e estratégias de teste de serviços ainda é uma área de pesquisa muito recente e que seu sucesso é a base para maior difusão das arquiteturas orientadas a serviços com suas características dinâmicas e adaptáveis. Em especial, há dificuldades para testar componentes e serviços com base em técnicas estruturais que, sabidamente, são complementares ao teste funcional. Isso evidencia a necessidade de, além de testar a funcionalidade dos serviços antes de usá-los, desenvolver abordagens que garantam a qualidade dos serviços ao longo do ciclo de vida da aplicação e dos serviços.

Os fatores apresentados anteriormente motivaram a proposta de trabalho de doutorado de Eler (2008), que tem por objetivo desenvolver um serviço de teste estrutural para testar programas, componentes e serviços Web escritos em Java e AspectJ. O serviço de teste desenvolvido está disponibilizado na Web para apoiar o desenvolvedor na realização dos testes dos componentes e serviços. Pretende-se apoiar o desenvolvedor, o integrador e o certificador. O serviço chama-se JaBUTiWS (Eler *et al.*, 2009) e tem como base a ferramenta JaBUTi, que é uma ferramenta de teste estrutural para programas orientados a objetos e aspectos escritos em Java e AspectJ. Além disso, faz parte deste trabalho permitir que o JaBUTiWS seja registrado em um registro que contenha serviços de apoio ao teste de software, para que os desenvolvedores de serviços possam localizá-lo e utilizá-lo dinamicamente. Este trabalho de mestrado está inserido no contexto do projeto de doutorado descrito acima e tem por objetivo auxiliar no processo de registro e validação do serviço de teste desenvolvido.

Outro fator importante a considerar nas abordagens orientadas a serviços refere-se a questões relacionadas a como gerenciar todos os serviços existentes. Cedo ou tarde surgem necessidades como, por exemplo, como um cliente pode encontrar um dado serviço e onde pode buscá-lo, ou ainda, onde encontrar informações adicionais de um serviço para permitir o seu uso correto. Essas questões levam ao conceito de um *registro de serviços* [1] ou *broker*. O registro tem o papel de facilitar o processo de descoberta de serviços. Ele é responsável por permitir o registro, a

[1] O registro de serviços na abordagem SOA é, do ponto de vista da arquitetura, um módulo ou componente, mas é também um serviço, por disponibilizar operações codificadas em WSDL, e também um software, ferramenta ou

publicação e a descoberta de serviços. Dessa forma, à medida que as aplicações SOA tornam-se pervasivas – crescente em número – o registro torna-se um ponto-chave para SOA.

Na perspectiva apresentada, um problema que ocorre refere-se às buscas realizadas nos registros de serviços, pois elas são basicamente sintáticas e, dessa forma, os clientes dos serviços não têm como saber se os serviços encontrados atendem de fato suas necessidades. Além disso, existem poucas opções de serviços de registros públicos e de uso aberto pela comunidade.

Por último, um aspecto relevante em relação ao domínio teste de software é que há uma quantidade cada vez maior de conceitos e informações de teste acumulados nos últimos anos. Nesse cenário, o conceito de ontologia assume um papel importante, pois sua ideia é justamente a de promover o reúso do conhecimento acumulado em uma determinada área, por meio de um vocabulário bem estabelecido e comum (Gruber, 1995). Tais fatores têm motivado alguns pesquisadores a propor ontologias de teste de software (Bai *et al.*, 2008; Barbosa *et al.*, 2006b). Uma abordagem genérica chama-se modelo de teste U2TP (U2TP, 2007) e permite a modelagem de diversos artefatos de teste de software. Destaca-se a ontologia de Barbosa *et al.* (2006b), denominada *OntoTest*, que foi desenvolvida para explorar os diferentes aspectos envolvidos na atividade de teste, tais como técnicas, critérios, etc. O propósito é definir um vocabulário comum para a área de ferramentas para apoio à atividade de teste, de modo a facilitar a construção de ferramentas de apoio e promover interoperabilidade entre elas.

1.2 Objetivos

Partindo dessas motivações, no presente trabalho é abordado o problema de incluir o serviço JaBUTiWS em registros de serviços Web relacionados a testes de software. Foi feita uma busca por um site de registros de serviço desse tipo e nenhum serviço de registro específico para ferramentas de teste foi encontrado. Dessa forma, a proposta do projeto de mestrado tem por objetivo especificar, projetar e implementar um Registro de Serviços específico para serviços de teste de software.

O fato de que o serviço de registro é específico para serviços de teste possibilita identificar e utilizar ontologias de teste para apoiar a classificação, o processo de descoberta e incluir informação semântica nos serviços catalogados. Para o registro de serviços desenvolvido pretende-se utilizar uma combinação das ontologias de Barbosa *et al.* (2006b) e Bai *et al.* (2008) para apoiar a classificação, armazenamento e recuperação dos serviços. A ideia é permitir por meio da ontologia, por exemplo, identificar qual estratégia ou técnica (estrutural, funcional, etc.) o serviço

aplicação que permite interação direta com o usuário para buscas. Assim, quando não houver risco de haver confusão, ele será chamado apenas de registro na maior parte deste texto.

de teste apoia, e assim aumentar a chance de que o serviço encontrado atenda às necessidades do cliente.

Além disso, pretende-se também validar o registro de serviços que está sendo desenvolvido. Para tanto será feita a inclusão de pelo menos três serviços de apoio ao teste no registro, incluindo o serviço JaBUTiWS. Adicionalmente, será validado a interação do registro com o serviço JaBUTiWS por meio da definição de um processo de teste simplificado, que inclui inclui *scripts* de teste para invocar as operações do serviço.

1.3 Organização

Neste capítulo foram apresentados o contexto no qual este trabalho está inserido, as motivações para a sua realização e os objetivos a serem alcançados. No Capítulo 2 é apresentada uma síntese dos fundamentos relacionados ao teste de software, abordando-se técnicas, critérios e ferramentas de teste de programas orientado a objetos e orientado a aspectos. Além disso, apresenta-se uma visão geral sobre processo de teste e aborda-se em detalhes ontologias na área de teste de software.

No Capítulo 3 é apresentada uma revisão dos principais conceitos das arquiteturas orientadas a serviços, com foco no componente de registro de serviços. É apresentada a arquitetura de um serviço Web, seus padrões de implementação e também é fornecido um exemplo de uso de um serviço Web. Em relação ao registro de serviços, é apresentado um estudo sobre suas principais características, funcionamento e seus padrões de implementação. Por último, é apresentada uma visão geral de alguns registros de domínio público.

No Capítulo 4 é apresentada uma arquitetura genérica baseada em serviços para o domínio de ferramentas de engenharia de software. Além disso, é realizada uma instanciação desta arquitetura para o domínio de ferramentas de teste e é apresentado o desenvolvimento de uma ontologia de teste para apoiar a descrição e recuperação de serviços desse domínio.

No Capítulo 5 são apresentados os detalhes de projeto e implementação do registro de serviços específico para serviços de teste de software. Também são discutidos as funcionalidades e os aspectos operacionais do registro desenvolvido no escopo deste trabalho.

No Capítulo 6 são apresentados o uso e validação do registro e um exemplo de interação com o serviço JaBUTiWS é detalhado. Finalmente, no Capítulo 7 são apresentadas as contribuições deste trabalho e os trabalhos futuros.

CAPÍTULO

2

Fundamentos do Teste de Software

2.1 Considerações Iniciais

Este capítulo tem por objetivo introduzir os conceitos básicos sobre teste de software, relevantes para esta proposta de trabalho. Ele está organizado da seguinte maneira: nas seções 2.2, 2.3 e 2.4 são apresentadas as definições básicas e a terminologia, bem como as principais técnicas de teste. Nas Seções 2.5 e 2.6 são discutidas algumas das principais questões relacionadas ao teste OO e OA. Na Seção 2.7 são apresentadas algumas iniciativas acadêmicas e industriais em relação ao desenvolvimento de ferramentas de teste. Relevante a esse trabalho, nas Seções 2.8 e 2.9 apresenta-se, respectivamente, algumas características referentes a um processo de teste e também um estudo de ontologias na área de teste de software. Por fim, na Seção 2.10 são apresentadas as considerações finais deste capítulo.

2.2 Terminologia e Conceitos Básicos

A utilização cada vez maior de sistemas baseados em computação em praticamente todas as áreas da atividade humana tem provocado uma demanda cada vez maior por qualidade e produtividade, tanto do produto quanto do processo de desenvolvimento de software, com o objetivo de atingir os padrões de qualidade especificados. Nesse sentido, a atividade de teste de software

constitui um dos principais elementos para fornecer evidências da confiabilidade do software (Barbosa *et al.*, 2006a; Maldonado, 1991).

É importante ressaltar que o objetivo da atividade de teste não é mostrar que um programa está correto. Ao invés disso, seu objetivo é identificar a presença de defeitos, caso eles existam. Dessa forma, para que os defeitos sejam descobertos antes de o software ser liberado para utilização, existe uma série de atividades, coletivamente chamadas de Validação, Verificação e Teste (VV&T), com a finalidade de garantir que tanto o modo pelo qual o software está sendo construído quanto o produto em si estejam em conformidade com o especificado (Delamaro *et al.*, 2007).

Atividades de VV&T não se restringem ao produto final. Ao contrário, podem e devem ser conduzidas durante todo o processo de desenvolvimento do software, desde a sua concepção e durante o desenvolvimento e a manutenção. Costuma-se dividir as atividades de VV&T em estáticas e dinâmicas. As estáticas são aquelas que não requerem a execução ou mesmo a existência de um programa executável para serem conduzidas. As dinâmicas são aquelas que se baseiam na execução de um programa ou de um modelo.

O teste de software é uma atividade dinâmica e sua intenção é a de executar o programa ou modelo utilizando entradas em particular e verificar se o comportamento está de acordo com o esperado. Caso a execução apresente resultados que não estão na especificação, diz-se que um erro ou defeito foi identificado. Além disso, os dados de tal execução podem servir como fonte de informação para a localização e a correção dos defeitos (Delamaro *et al.*, 2007).

Dentro desse contexto, para evitar eventuais ambiguidades, torna-se importante distinguir os termos defeito, engano, erro e falha. Segundo o padrão IEEE 610.12 (IEEE, 1990), um *defeito* é definido como sendo um passo, processo ou definição de dados incorretos; *engano* é a ação humana que produz um defeito; *erro* caracteriza-se por um estado inconsistente ou inesperado, originado por um defeito, durante a execução de um programa; e *falha* é a produção de uma saída incorreta em relação à especificação, ou seja, o resultado produzido pela execução é diferente do esperado.

Nota-se que os conceitos de engano e defeito são estáticos, pois não estão associados a um determinado programa ou modelo, bem como não dependem de uma execução particular. Além disso, é importante destacar que as definições apresentadas no parágrafo anterior não são seguidas o tempo todo entre os pesquisadores e engenheiros de software. Em particular, utiliza-se o termo erro de uma forma bastante flexível, muitas vezes significando defeito, erro ou até mesmo falha (Delamaro *et al.*, 2007).

Por fim, faz-se necessário definir que um caso de teste é um par formado por um dado de teste (elemento do domínio de possíveis entradas do programa) mais o resultado esperado para a

execução do programa com aquele dado de teste. Ao conjunto de todos os casos de teste usados durante uma determinada atividade de teste denomina-se de conjunto de teste ou conjunto de casos de teste.

2.3 Fases de Teste

De maneira geral, pode-se distinguir as seguintes fases de teste: teste de unidade, teste de integração, teste de sistema e teste de regressão (Delamaro *et al.*, 2007).

Teste de unidade: tem como foco as menores unidades de um programa, que podem ser funções, procedimentos, métodos ou classes. Nesse contexto, espera-se que sejam identificados erros relacionados a estruturas de dados incorretas ou simples erros de programação. O teste de unidade pode ser aplicado pelo próprio desenvolvedor à medida que ocorre a implementação, sem a necessidade de dispor-se do sistema finalizado.

Teste de integração: é realizado após o teste das unidades em separado e tem como ênfase descobrir defeitos nas interfaces das unidades ou módulos, durante a integração da estrutura do sistema. À medida que as diversas partes do software são colocadas para executar juntas, deve-se verificar se a interação entre elas funciona de maneira adequada e não contém falhas.

Teste de sistema: inicia depois que se tem o sistema completo, com todas as partes integradas. O objetivo é verificar se as funcionalidades especificadas nos documentos de requisitos estão todas corretamente implementadas. São explorados aspectos de correção, coerência e também requisitos não funcionais, como segurança e desempenho.

Teste de regressão: é realizado durante a manutenção do software. A cada modificação efetuada no sistema, corre-se o risco de que novos defeitos sejam introduzidos. Por esse motivo, deve-se realizar testes periódicos que comprovem que as modificações realizadas funcionam como o esperado e os requisitos anteriores continuam válidos.

Qualquer que seja a estratégia escolhida, existem quatro etapas bem definidas para realização da atividade de teste: planejamento, projeto dos casos de teste, execução e análise dos resultados. Além disso, outra importante questão refere-se à qualidade dos casos de teste desenvolvidos. Segundo Zhu *et al.* (1997), um critério de teste define quais propriedades ou requisitos de um programa devem ser exercitados para se avaliar a qualidade dos casos de teste.

2.4 Técnicas de Teste

Um ponto crítico da atividade de teste refere-se ao projeto dos casos de teste. Para tanto, é importante procurar formas de se utilizar apenas um subconjunto reduzido do domínio de valores de entrada, mas que ainda tenha alta probabilidade de revelar a presença de defeitos. Isso ocorre porque, em geral, o domínio de entrada pode ser infinito ou muito grande, tornando o tempo para realizar a atividade de teste impraticável. Essa limitação fez com que fossem definidas técnicas de teste e critérios estabelecidos por elas, de modo que a atividade de teste possa ser conduzida de forma sistemática (Delamaro *et al.*, 2007; DeMillo, 1980).

Em geral, há quatro principais técnicas de teste: funcional, estrutural, baseada em erros e baseada em estados. O que distingue cada uma delas é a fonte utilizada para definir requisitos de teste. Na técnica funcional, os requisitos de teste são estabelecidos a partir da especificação do software. Na técnica estrutural, os requisitos são derivados a partir da implementação do software. Na técnica baseada em erros, os requisitos de teste são obtidos a partir do conhecimento sobre erros típicos cometidos durante o processo de desenvolvimento de software. Na técnica baseada em estados, os requisitos de teste são derivados a partir da especificação representada por um modelo de estados, tal como uma máquina de estado finito ou um statechart (Barbosa *et al.*, 2006a).

2.4.1 Técnica Funcional

O teste funcional é uma técnica em que considera o programa testado como uma caixa preta, em que os detalhes de implementação não são considerados. Dessa forma, o software é avaliado sob o ponto de vista do usuário. São fornecidas entradas e avaliadas as saídas geradas para verificar se elas estão em conformidade com a especificação.

Por não levar em conta os detalhes de implementação, os critérios da técnica funcional podem ser aplicados em todas as fases da atividade de teste e em produtos desenvolvidos em qualquer paradigma de programação. Entretanto, é importante ressaltar que os critérios dependem da existência de uma boa especificação de requisitos, pois se baseiam nos requisitos do software em teste. A seguir são apresentados os critérios mais conhecidos da técnica funcional.

Particionamento em Classes de Equivalência

Como o domínio de entrada de um programa pode ser muito grande, o teste exaustivo torna-se infactível. Por isso, o particionamento de equivalência divide o domínio de entrada do programa em classes de equivalência que, de acordo com a especificação do programa, são tratadas da

mesma maneira. O objetivo é diminuir a quantidade de dados de entrada para tornar a atividade de teste viável. Assim, uma vez definidas as classes de equivalência, pode-se assumir que qualquer elemento da classe pode ser considerado um representante dela. Caso seja notado que algumas classes se sobrepõem, elas devem ser reduzidas para se separem e se tornarem distintas (Delamaro *et al.*, 2007).

Após identificar as classes de equivalência, deve-se determinar os casos de teste escolhendo-se um elemento de cada classe. De acordo com a descrição do critério, os passos a seguir podem ser realizados: i) identificar as classes de equivalência, observando as entradas e as saídas do programa com o objetivo de particionar o domínio de entrada; e ii) gerar casos de teste selecionando um elemento de cada classe, para ter o menor número de casos de teste possível.

Como uma avaliação do critério vale ressaltar que sua força está na capacidade de redução no tamanho do domínio de entrada e na criação de dados de teste baseados unicamente na especificação. Sua aplicação é indicada para programas em que as variáveis de entrada podem ser identificadas mais facilmente e assumem valores específicos. No entanto, um problema que pode ocorrer é que a especificação pode sugerir um grupo de dados que seria processado de forma idêntica, mas na prática isso não ocorrer. Por último, como desvantagem, tem-se que a técnica não fornece diretrizes para determinação dos dados de teste que permitam cobrir as classes de equivalência de maneira mais eficiente (Delamaro *et al.*, 2007).

Análise do Valor Limite

De acordo com Myers *et al.* (2004), a experiência mostra que casos de teste que exploram condições limites têm uma maior probabilidade de encontrar defeitos. Tais condições correspondem a valores que estão exatamente sobre ou imediatamente acima ou abaixo dos limitantes das classes de equivalência (Delamaro *et al.*, 2007).

Assim, este critério atua como um complemento ao particionamento de equivalência. Porém, ao invés de selecionar qualquer elemento de uma classe, os casos de teste são escolhidos nas fronteiras das classes, de modo que o limitante de cada classe de equivalência seja explorado. Segundo Myers *et al.* (2004), além da escolha seletiva dos dados de teste, o outro ponto que distingue esse critério do Particionamento de Equivalência é a observação do domínio de saída, uma vez que se exigem casos de teste que produzam resultados nos limites das classes de saída.

Grafo Causa-Efeito

O critério Grafo Causa-Efeito busca explorar algo não abordado pelos outros critérios, que são as combinações dos dados de entrada. Isso é feito por meio da definição de casos de teste que

exploram ambiguidades e incompletude nas especificações. Os requisitos de teste são estabelecidos baseado nas condições de entrada. Primeiramente, são identificadas as condições de entrada (causas) e as possíveis ações (efeitos) do programa. Em seguida constrói-se um grafo relacionando as causas e efeitos levantados. Esse grafo é convertido em uma tabela de decisão a partir da qual são derivados os casos de teste.

A vantagem deste critério é que ele exercita combinações de dados de teste que, possivelmente, não seriam consideradas. Entretanto, uma dificuldade caracteriza-se pela complexidade em se desenvolver o grafo caso o número de causas e efeitos seja muito grande. Vale ressaltar que a eficiência desse critério depende da qualidade da especificação. Uma especificação muito detalhada pode levar a um grande número de causas e efeitos e, por outro lado, uma especificação muito abstrata pode não gerar dados de teste significativos.

2.4.2 Técnica Estrutural

O teste estrutural (ou caixa branca) baseia-se na estrutura interna do programa para derivar seus requisitos de teste. Essa técnica requer a execução de partes ou componentes do programa sob o teste, sendo testados caminhos lógicos como condições e laços, bem como pares de definição e uso de variáveis. Dessa forma, os detalhes de implementação são de fundamental importância para geração dos casos de teste. Os critérios da técnica estrutural baseiam-se na complexidade, fluxo de controle e fluxo de dados do programa associado (Barbosa *et al.*, 2006a).

A maioria dos critérios utiliza uma representação de programa chamada de "Grafo de Fluxo de Controle" (GFC). Um programa P pode ser decomposto em um conjunto de blocos disjuntos de comandos. Em geral, a representação de um programa P por um GFC ($G = (N, E)$ em que N representa o conjunto de nós e E o conjunto de arestas) consiste em criar uma correspondência entre nós e blocos, além de indicar possíveis fluxos de controle entre blocos por meio das arestas. Assume-se que o GFC é um grafo orientado, com um único nó de entrada e de saída, no qual cada nó representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro. A partir do GFC podem ser escolhidos os elementos que serão executados, caracterizando assim o teste estrutural (Barbosa *et al.*, 2006a; Delamaro *et al.*, 2007).

As ocorrências de uma variável em um programa podem ser classificadas como uma "definição" ou um "uso". Uma ocorrência de uma variável é uma definição se ela está: (1) à esquerda em um comando de atribuição; (2) em um comando de entrada; ou (3) em parâmetros de saída nas chamadas de procedimentos. A ocorrência de uma variável é um uso quando a referência a ela não a definir. Podem ocorrer dois tipos de uso: c-uso e p-uso. O primeiro tipo refere-se a uma

computação realizada e está associado aos nós do GFC; o segundo tipo afeta o fluxo de controle de um programa e está associado às arestas do GFC (Maldonado, 1991).

Critérios Baseados na Complexidade

Conforme o nome sugere, tais critérios utilizam informações sobre a complexidade do programa para derivar os requisitos de teste. O principal exemplo dessa classe é o critério de McCabe, que utiliza a complexidade ciclomática do GFC para extrair os requisitos de teste. A complexidade ciclomática proporciona uma medida quantitativa da lógica de um programa. Ela oferece um limite máximo para o número de casos de teste que devem ser derivados a fim de garantir que todas as instruções sejam executadas pelo menos uma vez. A métrica de McCabe permite uma noção quantitativa da dificuldade na condução dos testes e uma indicação de confiabilidade final (Pressman, 2005).

Critérios Baseados no Fluxo de Controle

Nos critérios baseados no fluxo de controle são consideradas somente características de execução do programa, como comandos ou desvios, para obter os requisitos de teste. Os critérios mais conhecidos dessa classe são:

Todos-Nós: requer que sejam executados ao menos uma vez os nós do GFC, ou seja, que cada comando do programa seja executado pelo menos uma vez.

Todos-Arestas: requer que sejam executadas ao menos uma vez cada aresta do GFC, ou seja, cada desvio de fluxo de controle do programa.

Todos-Caminhos: requer que todos os caminhos possíveis do GFC sejam executados.

Critérios Baseados no Fluxo de Dados

Os critérios de fluxo de dados utilizam uma análise de fluxo para obter os requisitos de teste. Em tais critérios são requeridos o teste de definições e uso de variáveis, ou seja, as interações que envolvem definições de variáveis e subsequentes referencias às definições. Uma motivação para o surgimento dessa classe de critérios surgiu do fato de que, em geral, o teste baseado nos critérios Todos-Nós e Todos-Arestas possui resultados pouco eficazes para revelar a presença de defeitos até mesmo em programas simples. De forma semelhante, o critério Todos-Caminhos é aplicado de maneira bastante restrita, pois na maioria das vezes o número de caminhos de um programa é infinito ou extremamente grande devido às estruturas de repetição (Barbosa *et al.*, 2006a; Delamaro *et al.*, 2007).

Dentre os critérios mais conhecidos de fluxo de dados destacam-se os Potenciais-Usos (Maldonado, 1991) e os propostos por Rapps e Weyuker (1985). A seguir os critérios propostos por Rapps e Weyuker (1985) são apresentados:

Todas-Definições: requer que cada definição de variável seja exercitada pelo menos uma vez, tanto por p-uso ou c-uso.

Todos-Usos: requer que todas as associações entre uma definição de variável e seus usos sejam exercitadas através de um caminho livre de definição.

Todos-Du-Caminhos: requer que toda associação entre uma defnição-uso de variável seja exercitada por todos os caminhos livres de definição e livres de laço que cubram essa associação.

2.4.3 Técnica Baseada em Erros

O teste baseado em erros utiliza informações sobre os erros mais frequentes no processo de desenvolvimento de software para derivar os requisitos de teste. A técnica enfatiza os erros que o programador pode cometer durante o desenvolvimento e nas abordagens que podem ser usadas para detectar a sua ocorrência. Os exemplos típicos de critérios para essa técnica são Semeadura de Erros e Análise de Mutantes.

No critério Semeadura de Erros, é introduzida uma quantidade conhecida de erros no programa. Então executam-se os testes e do total de defeitos encontrados, verificam-se quais são naturais (defeitos inerentes do programa) e quais são artificiais (defeitos semeados). Por último, utilizando probabilidade, estima-se número de defeitos naturais ainda existentes no programa (Budd, 1981).

O critério Análise de Mutantes é utilizado para avaliar quanto um conjunto de casos de teste T é adequado para um programa P. Basicamente o critério faz uso de operadores de mutação para gerar um conjunto de programas ligeiramente modificados, conhecidos por mutantes, obtidos a partir do programa original P. O objetivo é determinar um conjunto de casos de teste que consiga revelar, por meio da execução de P, as diferenças de comportamento existentes entre P e seus mutantes (DeMillo *et al.*, 1987).

2.4.4 Técnica Baseada em Estados

O teste baseado em estados faz uso de um modelo, uma Máquina de Estado Finito (MEF), para modelar o programa a ser testado e obter os requisitos de teste. Com base nesse modelo,

utilizam-se critérios de geração de sequências de teste a fim de comprovar se o comportamento da MEF gerada é realmente obtido (Vincenzi, 2004).

Em geral, para modelos descritos por meio de MEFs, os dados de entrada de num caso de teste podem ser descritos como uma série de ações que levam a uma sequência de estados; enquanto a saída esperada é a saída pretendida para cada transição somado com um estado final. Considerando que os estados são nós e transições são arestas, uma MEF pode ser vista como um grafo. Assim, alguns critérios podem ser definidos (Offutt *et al.*, 2003):

Todos-estados: requer que todos os estados de uma MEF sejam exercitados.

Todos-transições: requer que todas as transições de uma MEF sejam exercitadas.

Todos-combinações: requer que todas as combinações de transições de uma MEF sejam executadas. Dependendo da MEF, esse critério pode tornar-se infactível.

2.5 Teste de Software OO

O paradigma de programação orientada a objetos surgiu com o objetivo de suprir principalmente as deficiências do paradigma procedimental. A ideia é agrupar em uma entidade, denominada classe, os dados (atributos) e funções (métodos) que manipulam estes dados. Este paradigma possui diversas vantagens em relação aos anteriores, dentre as quais se podem destacar o encapsulamento, que é a capacidade de um objeto manter isolados seus dados impedindo outros objetos de acessá-los, promovendo o chamado ocultamento de informação, além de encorajar a modularidade do programa e a herança, que permite novas classes serem definidas em função de classes já existentes. Tais recursos promovem o reúso e facilidade no entendimento do código, além de tornar a atividade de manutenção mais fácil.

Entretanto, ressalta-se que, apesar dessas vantagens, o uso de orientação a objetos por si só não é capaz de garantir o correto funcionamento de um programa, visto que ela não impede os desenvolvedores de cometerem enganos. Ao contrário, observa-se que as características de OO introduzem novas fontes de falhas, o que tornam as atividades de VV&T de fundamental importância no contexto OO (Delamaro *et al.*, 2007).

De acordo com Binder (1999), o paradigma OO possui construções poderosas que apresentam riscos de defeitos e problemas de teste. Isso é resultado do encapsulamento de métodos e atributos em uma classe e da possibilidade de, em poucas linhas de código, atribuir um comportamento ao sistema que só será definido em tempo de execução, por meio do acoplamento dinâmico. Vincenzi (2004) destaca que defeitos na programação de interfaces das funções são

comuns em programas procedimentais. Os programas OO possuem, em geral, muitos métodos e, consequentemente, muitas interfaces, aumentando a ocorrência desses tipos de defeitos.

Outro recurso que pode causar problemas para o teste é o polimorfismo. Embora ele possa ser utilizado para produzir código elegante e extensível, alguns problemas podem se originar de sua utilização. Por exemplo, suponha que um método `x()` em uma superclasse necessite ser testado. Em seguida, o método `x()` é sobrescrito em alguma subclasse. O funcionamento adequado do método `x()` não pode ser garantido, pois as pré-condições e pós-condições na classe derivada podem não ser as mesmas da superclasse (Binder, 1999; Vincenzi, 2004).

Fases do Teste OO

Como descrito anteriormente na Seção 2.3, a atividade de teste pode ser considerada incremental e realizada basicamente nas seguintes etapas: teste de unidade, teste de integração e teste de sistema. Algumas adaptações podem ser identificadas para o contexto de orientação a objeto, como será apresentado mais adiante. Binder (1999) destaca que, pelo fato de classes encapsularem, em geral, um grande conjunto de métodos que cooperam entre si, deve ser dada mais ênfase ao teste de integração.

É importante definir, em relação ao teste OO, o que se considera como a menor unidade de teste, pois pode ser tanto o método quanto a classe. Seguindo a abordagem de Vincenzi (2004) considera-se como a menor unidade a ser testada o método. Dessa forma, o teste de unidade no paradigma orientado a objetos também é conhecido por teste intramétodo. Além disso, como métodos de uma mesma classe podem interagir entre si, pode-se pensar no teste intermétodo como sinônimo para o teste de integração (Harrold e Rothermel, 1994).

McGregor e Korson (1994) consideram a menor unidade de teste a classe, eles propõem as seguintes variações em relação às estratégias do teste de POO:

- **Teste de classe**: enfatiza o teste de métodos de uma mesma classe, é classificada como um primeiro nível do teste de integração.
- **Teste de *cluster* (*grupo de classes*)**: consiste no teste de interação entre instâncias de classes de um mesmo *cluster*, classifica-se como um segundo nível do teste de integração.
- **Teste de sistema**: focaliza o teste do sistema completo, os casos de teste podem ser derivados de casos de uso ou de outros requisitos.

Harrold e Rothermel (1994) definem ainda outros dois tipos de teste para POO: teste intraclasse e teste interclasse. No primeiro caso, o objetivo é testar interações entre métodos públicos

por meio de chamadas em diferentes sequências. Busca-se ativar sequências inválidas de invocação de métodos inválidas que levem a estados inconsistentes. No teste interclasse a ideia é a mesma, distinguindo apenas que as sequências de invocações de métodos não precisam estar na mesma classe (Vincenzi, 2004).

Vincenzi (2004), mesmo considerando o método como menor unidade de teste, adaptou a abordagem de Harrold e Rothermel (1994) para o contexto do teste estrutural de unidade intramétodo. A Tabela 2.1 a seguir apresenta uma síntese dos tipos de teste que podem ser aplicados em cada uma das fases para programas OO, tanto considerando o método ou a classe como menor unidade.

Tabela 2.1: Tipos de teste aplicados nas fases do teste de programas OO (Adaptado do trabalho de Vincenzi (2004))

Fase	*Menor Unidade: Método*	*Menor Unidade: Classe*
Unidade	Intramétodo	Intramétodo, Intermétodo e Intraclasse
Integração	Intermétodo, Intraclasse e Interclasse	Interclasse
Sistema	Toda a Aplicação	Toda a Aplicação

Por último, vale ressaltar que diversos trabalhos podem ser encontrados na literatura em relação a estratégias e técnicas de teste desenvolvido para orientação a objetos. Em relação ao teste baseado em especificação, podem-se destacar os trabalhos de Binder (1999) e o de Chaim *et al.* (2003), já em relação ao teste baseado em programa destacam-se as propostas de Harrold e Rothermel (1994), Sinha e Harrold (1999), Vincenzi (2004), dentre outras.

2.6 Teste de Software OA

Sabe-se que a introdução da POO trouxe uma revolução às técnicas tradicionais de programação. Ainda assim, alguns problemas não foram completamente resolvidos por suas construções. Por exemplo, ela não possibilita uma clara separação de alguns interesses, usualmente não funcionais, também chamados de interesses transversais. Como uma proposta de resolução desse problema, Kiczales *et al.* (1997) propuseram a programação orientada a aspectos (POA), que oferece mecanismos para a construção de programas cujos interesses transversais ficam isolados e separados dos interesses funcionais, em vez de espalhados pelo sistema (Lemos, 2005).

A POA tem sido reconhecida como uma técnica que permite a construção de sistemas com melhor arquitetura, facilitando a manutenção dos diferentes interesses e a legibilidade do código. Entretanto, a sua simples utilização não evita que erros sejam introduzidos ao longo do desenvolvimento do software, sendo necessária a utilização de técnicas de teste para aumentar a qualidade e confiabilidade dos programas. Conforme ressaltado por Alexander (2003), o

entendimento e teste adequado de programas orientados a aspectos é um dos principais meios para tornar essa técnica menos custosa na prática. Portanto, a atividade de teste também assume um papel importante na programação orientada a aspectos (Lemos, 2005).

Antes de apresentar características do teste OA faz-se necessário definir primeiro alguns conceitos fundamentais de POA. Vale ressaltar que as definições apresentadas a seguir estão relacionadas com a linguagem AspectJ (Kiczales *et al.*, 2001), que é uma extensão de Java criada para permitir a POA de maneira genérica.

- *Aspectos:* interesses dos sistemas de software (conhecidos por interesses transversais) que não se encaixam bem em módulos de programação individuais. Esses tipos de interesse tendem a se espalhar através dos módulos do programa e, por isso, são chamados de transversais (*crosscutting*) (Kiczales *et al.*, 1997).

- *Pontos de Junção* (join points): são pontos concretos da execução de um programa que permitem aos aspectos adicionar comportamento em outros módulos do programa.

- *Conjuntos de junção* (pointcut): são utilizados para identificar diversos pontos de junção em um sistema. Após a identificação dos pontos, regras de combinação podem ser definidas como, por exemplo, realizar certa ação antes ou depois da execução dos pontos de junção (Lemos, 2005).

- *Adendo* (advice): são construções similares aos métodos e descrevem o comportamento em um dado ponto de junção; os adendos podem executar antes, depois ou no lugar dos pontos de junção.

Fases do Teste OA

Pode-se considerar que em programas OA, considerando-os como extensões de programas OO, as menores unidades a serem testadas são os métodos e adendos [1]. O aspecto ao qual o adendo pertence, somado com um ponto de junção que fará com que o adendo seja executado podem ser vistos como um *driver* do adendo, pois sem eles não é possível executar o adendo. Além disso, como um aspecto engloba basicamente atributos, métodos, adendos e conjuntos de junção, assim, considerando um único aspecto, pode-se pensar em teste de integração. Os adendos e métodos de um aspecto podem interagir para desempenhar funções específicas, o que caracteriza uma integração que deve ser testada.

Considerando os argumentos do parágrafo anterior, a atividade de teste de programas OA pode ser dividida nas seguintes fases (Lemos *et al.*, 2004):

[1] Parte desta seção foi baseada no trabalho de Masiero *et al.* (2006)

1. **Teste de Unidade:** testa cada método e adendo isoladamente. É dividido entre o teste intramétodo e o teste intraadendo.

2. **Teste de Módulo:** testa uma coleção de unidades dependentes – unidades que interagem por meio de chamadas ou interações com adendos. Essa fase pode ser dividida nos seguintes tipos de teste (considerando classes e aspectos como entidades diferentes):

   - Intermétodo: consiste em testar cada método público juntamente com outros métodos da mesma classe chamados direta ou indiretamente (chamadas indiretas são aquelas que ocorrem fora do escopo do próprio método, dentro de um método chamado em qualquer profundidade).
   - Adendo-método: consiste em testar cada adendo juntamente com outros métodos chamados por ele direta ou indiretamente.
   - Método-adendo: consiste em testar cada método público juntamente com os adendos que o afetam direta ou indiretamente (considerando que um adendo pode afetar outro adendo). Nesse tipo de teste não é considerada a integração dos métodos afetados com os outros métodos chamados por eles, nem com método chamados pelos adendos.
   - Adendo-adendo: consiste em testar cada adendo juntamente com outros adendos que o afetam direta ou indiretamente.
   - Intermétodo-adendo: consiste em testar cada método público juntamente com os adendos que o afetam direta e indiretamente, e com métodos chamados direta ou indiretamente. Esse tipo de teste inclui os quatro primeiros tipos de teste descritos acima.
   - Intraclasse: consiste em testar as interações entre os métodos públicos de uma classe quando chamados em diferentes sequências, considerando ou não a interação com os aspectos.
   - Interclasse: consiste em testar as interações entre classes diferentes, considerando ou não a interação dos aspectos.

3. **Teste de Sistema:** testa a integração de todos os módulos que formam um subsistema ou um sistema completo. Para essa fase geralmente é utilizado o teste funcional.

Por fim, vale ressaltar alguns trabalhos de verificação, validação e teste de software no contexto de POA que merecem destaque. Em especial no contexto de teste estrutural, dentre os quais se podem citar o trabalho de Zhao (2002, 2003), que apresenta uma abordagem baseada

no fluxo de controle de dados para o teste de unidade de programas orientados a aspectos e cria uma ferramenta que implementa a abordagem; e o trabalho de Lemos (2005), que propõe uma abordagem para o teste estrutural de unidade para programas OA. Entretanto, em oposição ao trabalho de Zhao (2003), os métodos e adendos são tratados isoladamente no teste de unidade. Em relação a outras propostas para o teste OA, sem relações com técnicas de teste específicas, podem-se citar os trabalhos de Alexander *et al.* (2004); Xie *et al.* (2006), dentre outros.

2.7 Ferramentas de Teste

A aplicação prática de um critério de teste está fortemente condicionada à sua automatização. O desenvolvimento de ferramentas de teste é de fundamental importância, uma vez que a atividade de teste é muito propensa a erros, além de improdutiva, se aplicada manualmente. As ferramentas de teste facilitam a condução de estudos experimentais que buscam avaliar e comparar os diversos critérios de teste (Delamaro *et al.*, 2007).

Isso motiva o desenvolvimento de ferramentas automáticas para auxiliar na condução de testes efetivos e na análise dos resultados obtidos. Esta seção tem por objetivo apresentar uma descrição de algumas das principais ferramentas para o apoio ao teste de software, com foco no teste estrutural.

Emma (Emma, 2006) é uma ferramenta de código aberto para teste de unidade de programas Java. Ela implementa os critérios Todos-Nós e Todas-Arestas e disponibiliza relatórios da cobertura alcançada. Uma vantagem de seu uso é que a ela não requer o código fonte do programa a ser testado, instrumentando diretamente o *bytecode* dos programas. Além disso, a ferramenta pode ser integrada com o Eclipse por meio do *plugin* EclEmma (EclEmma Plugin, 2006).

CodeCover (CodeCover Testing, 2007) é uma ferramenta de código aberto para o teste estrutural de programas escritos em Java ou em COBOL. Seu desenvolvimento foi iniciado em 2007 pela Universidade de Stuttgart na Alemanha. Ela oferece suporte aos critérios Todos-Nós e Todas-Arestas, além de gerar relatórios das coberturas alcançadas. Possui independência de plataforma, podendo ser executada pela linha de comando ou por meio de uma extensão (*plugin*) para o Eclipse.

C++ Test (Parasoft Corporation, 2005) é uma ferramenta de teste de unidade para programas C/C++ que executa os seguintes tipos de teste: teste funcional, teste estrutural e teste de regressão. Esta ferramenta automatiza o teste funcional por intermédio da geração automática dos casos de teste e dos resultados esperados, os quais são comparados com os resultados reais.

Além disso, o testador pode incluir seus próprios casos de teste e resultados esperados e obter um relatório dos resultados distintos do especificado (Domingues, 2002).

JTest (Parasoft Corporation, 2009) é uma ferramenta de teste de classes para programas Java e executa os seguintes tipos de teste: análise estática, teste funcional, teste estrutural e teste de regressão. No teste funcional, ela gera um conjunto essencial de casos de teste, projetado com o objetivo de alcançar a maior cobertura possível. Além disso, se desejar, o testador pode melhorar esse conjunto de casos de teste. Além de detectar erros, a ferramenta pode preveni-los, assegurando que eles não serão adicionados no código quando for modificado de forma automatizada (Domingues, 2002).

JUnit (Beck e Gamma, 2005) é um *framework* de teste que viabiliza a documentação e a execução automática de casos de teste. O *framework* disponibiliza relatórios sobre quais casos de teste não se comportaram de acordo com a especificação. Após a execução de um caso de teste, a saída obtida é compara com a saída esperada e as divergências são reportadas. O JUnit não fornece informação a respeito de cobertura obtida pelos casos de teste e não apoia a aplicação de um critério de teste.

Coverlipse (Coverlipse Plugin, 2005) é uma extensão da ferramenta JUnit em formato de *plugin* para o Eclipse. Ela permite visualizar a cobertura alcançada dos casos de teste do JUnit. Também oferece suporte aos critérios de teste Todos-Nós, Todas-Arestas e Todos-Usos.

POKE-TOOL (*Potencial Uses Criteria Tool for Program Testing*) é uma ferramenta que apoia a aplicação de critérios estruturais baseados em fluxo de controle e fluxo de dados. Inicialmente desenvolvida para o teste de programas escritos em C, e posteriormente estendida para o teste de programas em Cobol e Fortran (Chaim, 1991).

JaBUTi (*Java Bytecode Understanding and TestIng*) ferramenta de teste desenvolvida pelo Grupo de Pesquisa em Engenharia de Software do ICMC, em colaboração com outros grupos de pesquisa. Ela é um ambiente para o entendimento e teste de programas e componentes Java, possui como diferencial a não exigência do código fonte para realização dos testes. A ferramenta trabalha diretamente com *bytecode* Java e apoia a aplicação dos critérios Todos-Nós, Todas-Arestas, Todos-Usos e Todos-Potenciais-Usos no teste intramétodo (Vincenzi, 2004).

JaBUTi/AJ (*Java Bytecode Understanding and TestIng for AspectJ*) é uma extensão da ferramenta JaBUTi e apoia o teste de unidade de programas orientados a aspectos e também o teste estrutural de integração par-a-par escritos em Java e AspectJ. Por se tratar de uma extensão, todas as funcionalidades existentes na JaBUTi também estão presentes na JaBUTi/AJ (Lemos, 2005).

2.8 Processo de Teste de Software

Um processo, de modo geral, pode ser definido como um conjunto de atividades estruturadas e destinadas a resultar em um artefato ou serviço de valor para uma organização, cliente ou mercado e implica em uma ordenação específica de atividades com início, fim, insumos e produtos claramente identificados (Villela, 2004).

Um processo de software, por sua vez, pode ser definido como um conjunto de políticas, estruturas organizacionais, tecnologias, procedimentos e artefatos que são necessários para conceber, desenvolver, entregar e manter um produto de software (Fuggetta, 2000). Além disso, um processo de software independe da tecnologia utilizada no desenvolvimento e define o modo pelo qual o desenvolvimento de software é organizado, gerenciado, medido e melhorado (Montangero *et al.*, 1999). Segundo Lindvall e Rus (2000), o processo deve ajudar a guiar as pessoas em relação ao que fazer sobre a divisão e coordenação dos trabalhos e garantir uma comunicação efetiva.

Considerando um contexto ainda mais específico, no caso teste de software, Harrold (2000) aponta as principais direções para a área de teste que, segundo a autora, devem ser inevitavelmente exploradas nos próximos anos, conforme ilustra a Figura 2.1. Pode-se observar que dentre as principais direções para a área de teste de software destacam-se, entre outras, a importância de estudos para o desenvolvimento de processos de teste efetivos.

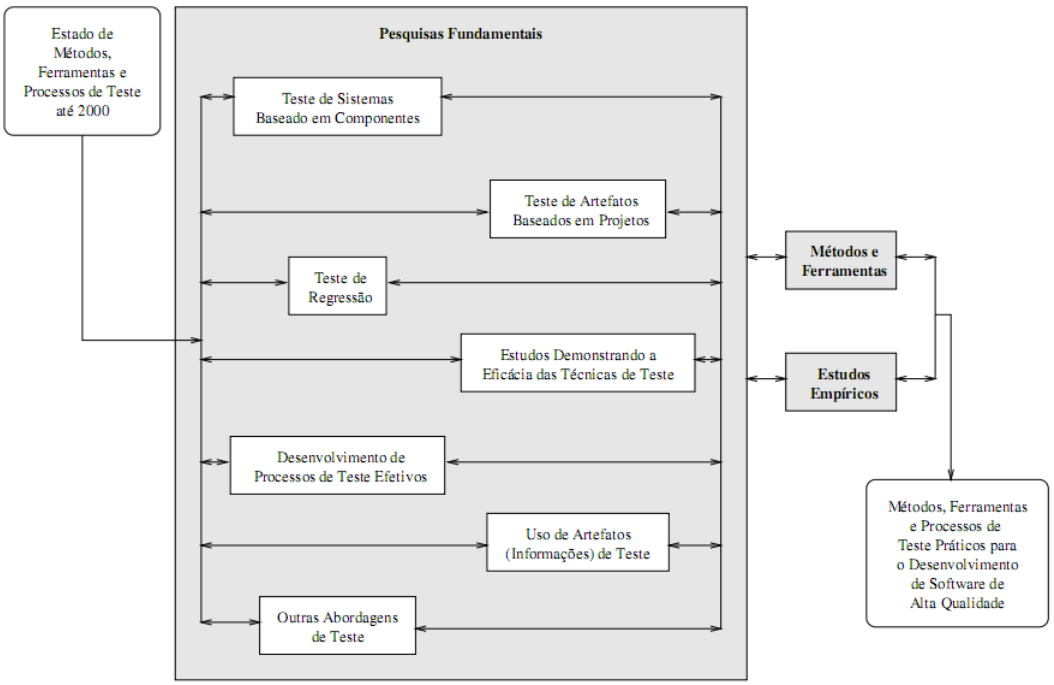

Figura 2.1: Direções de pesquisa na área de teste software (Harrold, 2000)

Na perspectiva apresentada, fazendo-se uma analogia com o processo de software, que consiste basicamente de uma sequência de passos para desenvolver e manter um software, pode-se definir *processo de teste* como uma sequência de etapas de teste necessárias para o desenvolvimento e manutenção da atividade de teste (Barbosa *et al.*, 2008). De acordo com McGregor e Korson (1994), um processo de teste possui dois objetivos principais: (1) elevar a confiança de que o software satisfaz, sob certas condições, seus objetivos; e (2) descobrir defeitos no software.

Ainda segundo McGregor e Korson (1994), um processo de teste deve ser interligado com o processo de software, tendo-se em mente o seguinte pensamento: "faça a análise, teste; projete um pouco, teste; codifique um pouco, teste". Isso se justifica, pois em sistemas de grande porte os erros são mais prováveis de ocorrer, tornando-se necessários removê-los o quanto antes possível, com o intuito de diminuir os custos de retrabalho e mantendo ainda a qualidade desejada.

Por último, Barbosa *et al.* (2008) ainda destacam que um processo de teste define-se tomando como base o paradigma de desenvolvimento e o domínio de aplicação em questão. Além disso, modelos de ciclo de vida de teste devem ser usados como referência para a definição de um processo de teste, estabelecendo, assim, suas etapas de teste principais.

2.9 Ontologias de Teste de Software

Uma ontologia é uma especificação explícita de um conceito comum (Gruber, 1995). Ela pode ser vista como uma visão simplificada e abstrata de um dado domínio de conhecimento. Ontologias têm sido aplicadas na descrição de diversos domínios do conhecimento como medicina, direito e também em computação. Elas têm se tornado populares, em grande parte, pelo fato de terem como objetivo promover um entendimento comum e compartilhado sobre um dado domínio.

De acordo com Duarte e Falbo (2000), as ontologias possuem os seguintes propósitos:

- refletir e melhorar a compreensão sobre o domínio, para as pessoas envolvidas no processo de desenvolvimento de uma ontologia;
- auxiliar a obtenção de consenso no entendimento de uma área de conhecimento, considerando que, em geral, para uma área de conhecimento, diferentes especialistas têm entendimento diferenciado sobre os conceitos envolvidos, o que leva a problemas na comunicação. Ao se construir uma ontologia, essas diferenças são explicitadas e busca-se um consenso sobre seu significado e sua importância;

- uma vez que haja uma ontologia sobre uma determinada área de conhecimento, uma pessoa que deseje aprender mais sobre essa área não precisa consultar sempre um especialista: ela pode estudar a ontologia e aprender sobre o domínio em questão por conta própria.

Ontologias podem ser usadas como artefatos de engenharia, constituídos por um vocabulário específico, com um conjunto de suposições referentes ao significado dos termos do vocabulário. Geralmente, esse conjunto de suposições tem a forma de uma teoria lógica de primeira ordem, no qual os termos do vocabulário aparecem como nomes de predicados unários ou binários, chamados de conceitos e de relações, respectivamente (Barbosa *et al.*, 2006b). Como são especificações formais e consensuais que oferecem um entendimento compartilhado de um domínio, a ser divulgado por pessoas e sistemas, elas trazem dois benefícios (Fensel, 2003):

- a definição de semânticas formais para a informação, permitindo o processamento da informação pelo computador;
- a definição de semânticas do mundo real, que torna possível relacionar conteúdo processável pelo computador com significado para seres humanos baseado em terminologias consensuais.

Gruber (1995) propõe um conjunto de características que ontologias devem ter para facilitar o compartilhamento e interoperabilidade de aplicações que as utilizam:

- **Clareza**: uma ontologia deve definir efetivamente o significado dos seus termos, de maneira objetiva;
- **Coerência**: uma ontologia deve ser coerente, ou seja, deve expressar inferências que são consistentes com suas definições;
- **Extensibilidade**: uma ontologia deve ser projetada a fim de antecipar o uso de vocabulário compartilhado;
- **Mínimo compromisso ontológico**: uma ontologia deve fazer poucas afirmações sobre o mundo a ser modelado, permitindo aos interessados especializar e instanciar a ontologia conforme necessário.
- **Mínima influência de codificação**: a conceitualização de ser especificada sem depender de uma codificação particular. A dependência deve ser minimizada para poder ser implementada com diferentes sistemas e estilos de representação.

As ontologias oferecem um entendimento compartilhado sobre domínios específicos. Em especial na área de computação, esse conhecimento pode ser utilizado entre sistemas computacionais. Segundo Freitas *et al.* (2005), uma ontologia é um conjunto de definições de conceitos, propriedades e relações que descrevem certo domínio de conhecimento, permitindo às aplicações usar de modo preciso, claro e formal a semântica da informação descrita pela ontologia. Por esse fato, ontologias constituem uma ferramenta útil para apoiar a especificação e implementação de sistemas de software.

Em engenharia de software, especificamente, diversas pesquisas têm sido conduzidas para criação de ontologias, a maioria delas para subdomínios específicos. Salienta-se que no futuro tais ontologias poderão ser integradas em uma única que abrange todo o domínio de engenharia de software. Falbo *et al.* (1998), desenvolveram uma ontologia sobre processos de software. Ghezzi e Gall (2008) criaram uma ontologia com conceitos de análise de programas. No trabalho de Hyland-Wood *et al.* (2006), os autores desenvolveram uma ontologia sobre conceitos de engenharia de software e definem formas de como ela pode ser utilizada para permitir a navegação em sistemas de software, independentemente de linguagem de programação, facilitando assim o entendimento e a manutenção do software.

O domínio de teste de software envolve a integração de três tipos básicos de conhecimento: teórico, empírico e baseado no uso de ferramentas de apoio. Com o passar dos anos este domínio tem produzido uma quantidade significativa de conceitos. Tal diversidade de informações e conceitos torna imprescindível a elaboração de um vocabulário comum, para permitir a criação de um entendimento consensual sobre teste de software (Barbosa *et al.*, 2006b).

Em relação ao domínio de teste de software, Huo *et al.* (2003) propuseram uma ontologia para apoiar o teste de aplicações Web em um ambiente de software multi-agente. Bai *et al.* (2008) criaram uma ontologia para modelar conceitos relacionados a execução da atividade de teste, tais como: casos de teste, plano de teste e configuração. Barbosa *et al.* (2006b) desenvolveram uma ontologia de teste, denominada *OntoTest*, que apoia a aquisição, organização, reúso e compartilhamento do conhecimento de teste. Também há uma abordagem mais genérica chamada modelo de teste U2TP (U2TP, 2007) que permite modelar outros conceitos de teste, tais como arquitetura de teste, comportamento, dados e tempo.

No contexto deste trabalho, é importante enfatizar as ontologias de Barbosa *et al.* (2006b), Bai *et al.* (2008) e U2TP (2007), pois elas foram desenvolvidas para explorar os diferentes aspectos envolvidos na atividade de teste, tais como técnicas, critérios, ferramentas, recursos humanos e organizacionais. Além disso, elas dão apoio à ideia do trabalho proposto no qual será incorporado uma ontologia de teste no registro desenvolvido.

Bai *et al.* (2008) definem uma ontologia que especifica os conceitos, relacionamentos e semântica de teste a partir de duas perspectivas: (1) *projeto de teste*, que especifica os conceitos

como casos de teste, dados e comportamento de teste; e (2) *execução de teste*, que especifica os conceitos necessários para exercitar os casos de teste, por exemplo: plano de teste, configuração e resultados do teste.

Na Figura 2.2 são mostrados os conceitos e relacionamentos da ontologia, na qual cada sistema em teste (*SUT*) está associado aos seguintes conceitos: uma área de dados (*DataPool*) que define os dados de teste e o conjunto de casos de teste, um plano de teste (*TestPlan*) que define configuração e execução de um teste (*Test*) e um conjunto de resultado de teste (*TestResult*).

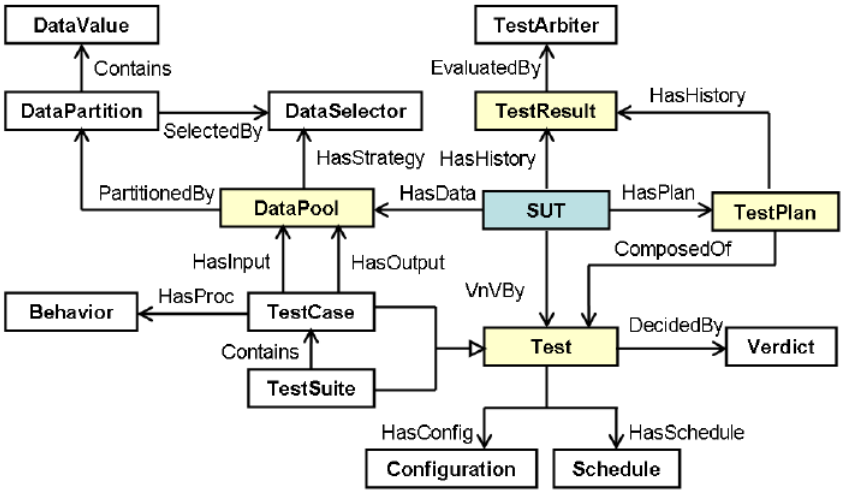

Figura 2.2: Ontologia de teste definida em Bai *et al.* (2008)

O U2TP é um perfil da UML 2.0 e define uma linguagem para especificar, projetar e visualizar artefatos de teste. É uma linguagem de modelagem de teste que pode ser aplicada em diversos domínios de aplicação, podendo ser usada sozinha ou integrada com a UML para manipular artefatos de teste. Basicamente, a linguagem possui uma estrutura organizada nos quatro grupos a seguir. Na Figura 2.3 são ilustrados alguns conceitos que estão presentes no modelo de teste U2TP.

- **Arquitetura de teste**: define conceitos relacionados à configuração e estrutura do teste.
- **Dados de teste**: define conceitos de valores do teste.
- **Comportamento**: define conceitos relacionados ao aspecto dinâmico do teste.
- **Tempo**: define conceitos relacionados à quantificação de tempo nos procedimentos de teste.

A *Ontotest* foi desenvolvida por pesquisadores do ICMC-USP com o objetivo de conceituar e formalizar os termos e relacionamentos existentes sobre teste de software. As definições

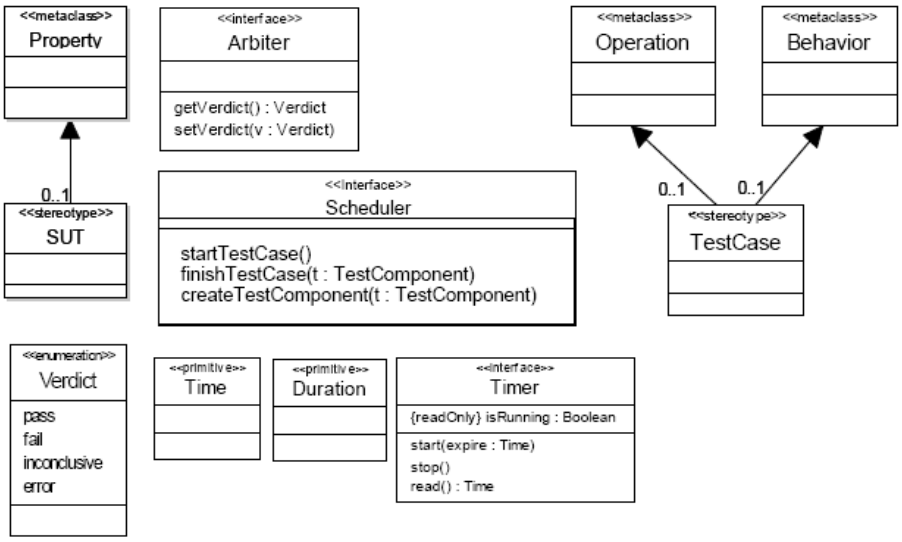

Figura 2.3: Alguns conceitos do modelo de teste definido em U2TP (2007)

a seguir são encontradas em Barbosa *et al.* (2006b) e Barbosa *et al.* (2008). A ontologia está organizada em camadas, considerando dois níveis de abstração. No nível mais alto, a ontologia é composta pelos principais conceitos e relações associadas à atividade de teste, compondo a ontologia principal (*Main Software Testing Ontology*), conforme ilustra a Figura 2.4.

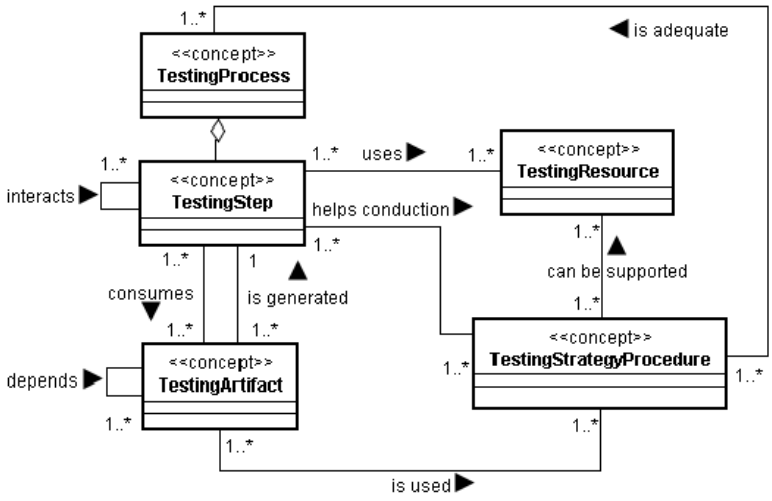

Figura 2.4: Estrutura da ontologia de teste *OntoTest* (Barbosa *et al.*, 2006b)

Nos sub-níveis, estes conceitos são refinados de maneira mais específica, compondo cinco sub-ontologias: (1) processo de teste (*TestingProcess*); (2) passo de teste (*TestingStep*); (3) artefato de teste (*TestingArtifact*); (4) estratégia e procedimento de teste (*TestingStrategyProcedure*); e (5) recurso de teste (*TestingResource*). As sub-ontologias são descritas a seguir.

1. **Processo de teste**: os processos de teste são baseados em uma tecnologia de desenvolvimento e um paradigma de desenvolvimento, além de um procedimento de teste. Modelos de ciclo de vida de teste são usados como referência na definição de um processo de teste, estabelecendo macropassos e a relação de dependência entre eles;

2. **Passo de teste**: quando um teste é realizado, um conjunto de passos essenciais deve ser considerado: os testadores devem ser capazes de planejar os testes, construir os casos de teste, executar os testes e analisar os resultados do teste. Além disso, o passo de teste pode ser visto como uma composição de atividades de teste;

3. **Artefato de teste**: um artefato de teste pode ser visto como um artefato de entrada ou de saída, dependendo de como for usado (consumido ou produzido por um passo de teste). Além disso, cada passo de teste pode envolver um número de diferentes artefatos, como documentos de teste, diagramas de teste, casos de teste, requisitos de teste, *drivers* e *stubs*, além de artefatos sob teste;

4. **Estratégia e procedimento de teste**: podem ser categorizados em métodos, guias e técnicas. Exemplos de técnicas são funcionais, estruturais, baseadas em erros e baseada em estados. Cada técnica primária estabelece um ou mais critérios de teste, que podem ser categorizados como critérios de seleção ou critérios de adequação. Diferentes estratégias podem ser estabelecidas, dependendo dos procedimentos de teste, passos de teste e fases consideradas;

5. **Recurso de teste**: de acordo com as sub-ontologias anteriores, um passo de teste pode ser visto como uma primitiva de transformação, no qual as entradas e saídas do passo de teste correspondem a um artefato de teste. Entretanto, outros elementos são necessários para a execução de um passo de teste, denominados recursos de teste. Esse recurso pode ser um recurso humano, como um testador ou um gerente de teste; um recurso de hardware ou de software, como uma ferramenta de teste ou um sistema de apoio. Recursos de hardware e software são caracterizados em termos de um ambiente de teste, que pode ser usado para automatizar os procedimentos de teste. As ferramentas de teste são um tipo especial de recurso de teste, que podem ser classificadas como primárias, organizacionais e de apoio, além de ferramentas de propósito geral.

Essa estrutura permite que a *OntoTest* seja flexível e possa ser integrada e reutilizada nas aplicações. Dependendo do contexto, ela pode ser utilizada completamente ou somente alguma de suas sub-ontologias pode ser considerada. É importante destacar estas partes, pois elas dão

apoio à proposta deste trabalho, no qual é utilizado parte da ontologia (sub-ontologia de estratégia e procedimento de teste) para apoiar a classificação e busca dos serviços armazenados no registro de serviços desenvolvido. Pode-se facilitar a busca de usuários interessados em serviços que apoiem o teste estrutural, por exemplo, provendo resultados de serviços classificados segundo a ontologia. As sub-ontologias são apresentadas nas Figuras 2.5, 2.6, 2.7, 2.8 e 2.9.

2.10 Considerações Finais

Neste capítulo foram apresentados os fundamentos de teste de software, abordando seus principais conceitos, técnicas e estratégias, tanto para o paradigma OO quanto OA. Foram apresentados exemplos de ferramentas com o objetivo de automatizar atividade de teste. Por último, foram apresentadas e discutidas ontologias de teste de software relevantes para este trabalho.

O próximo capítulo aborda a arquitetura orientada a serviços e detalhes específicos do componente de registro de serviços, escopo deste trabalho.

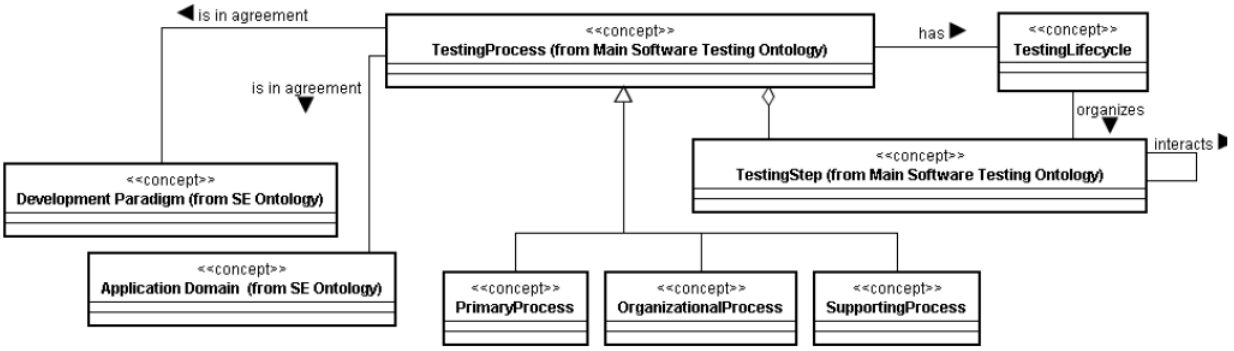

Figura 2.5: Sub-Ontologia de Processos de Teste

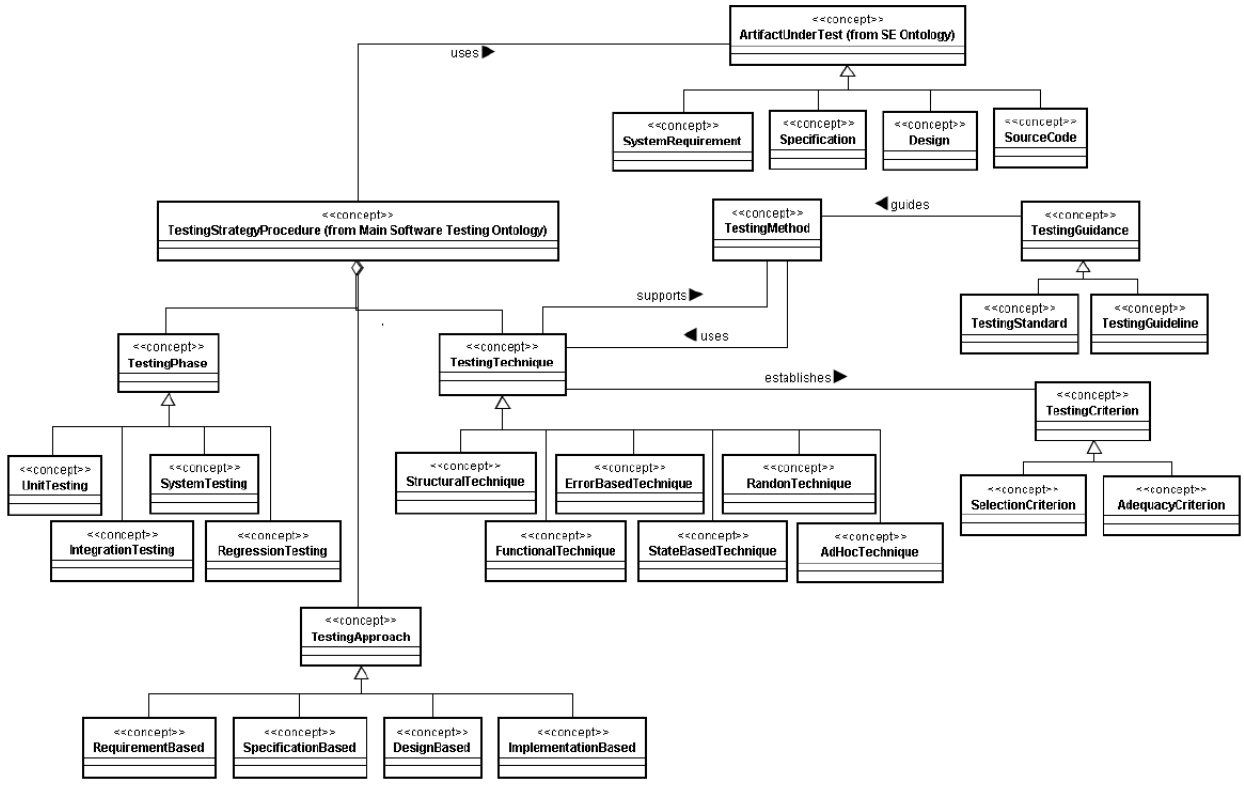

Figura 2.6: Sub-Ontologia de Estratégia e Procedimento de Teste

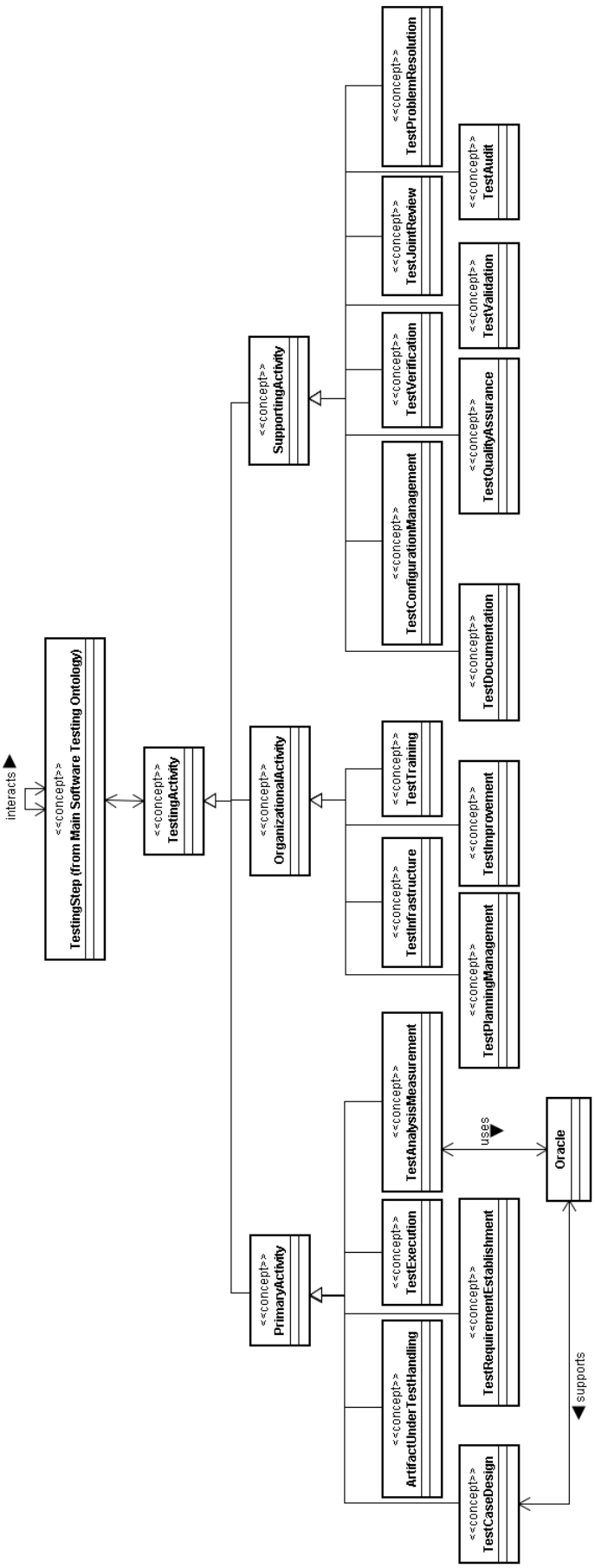

Figura 2.7: Sub-Ontologia de Passos de Teste

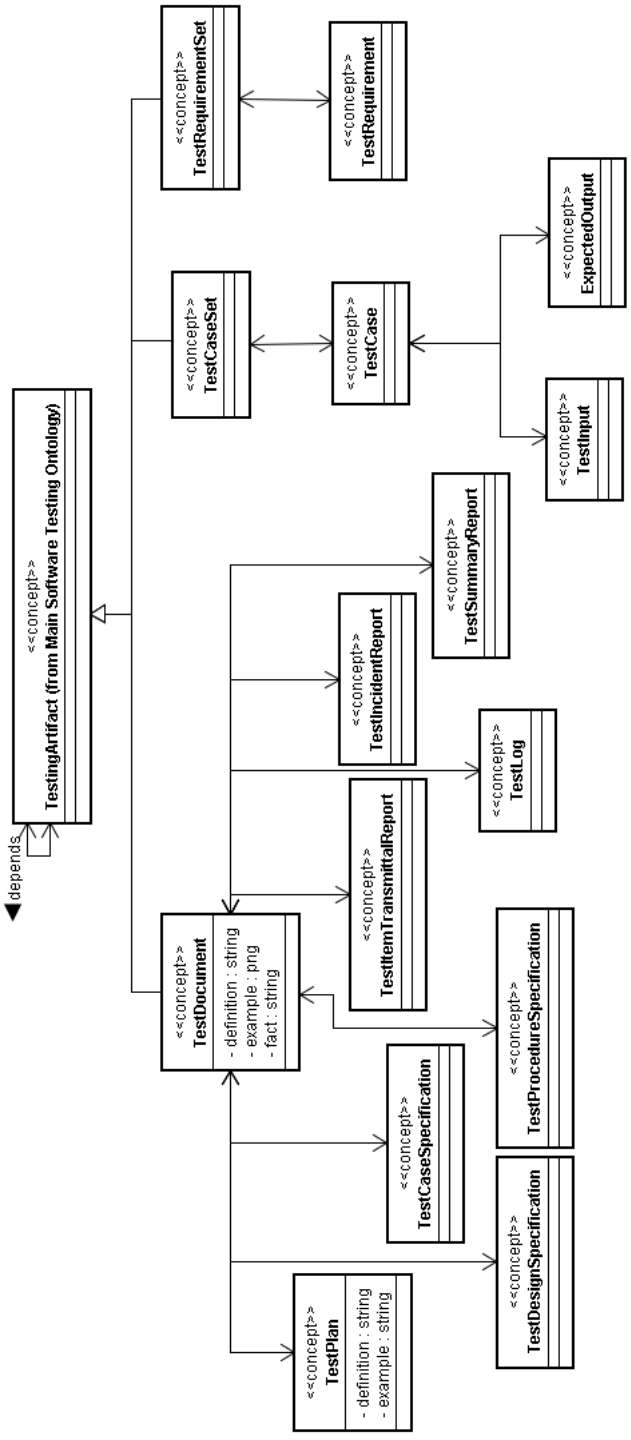

Figura 2.8: Sub-Ontologia de Artefatos de Teste

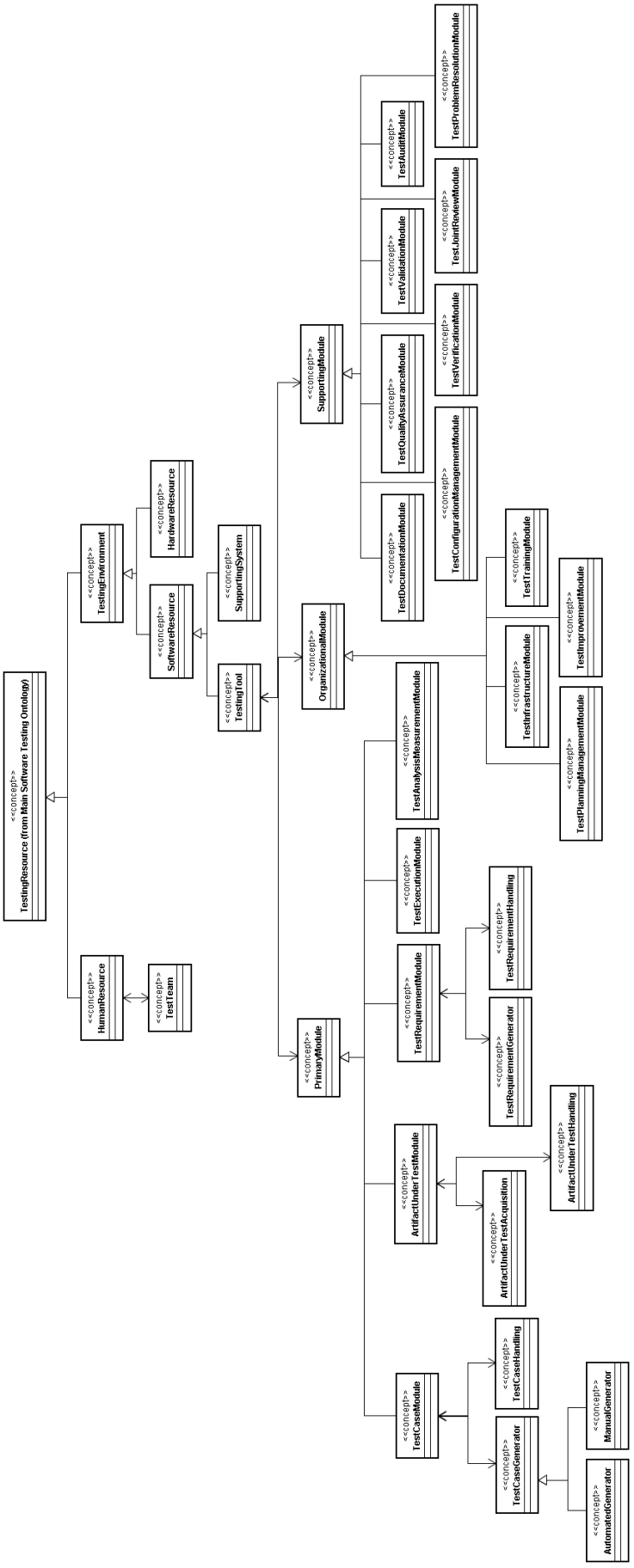

Figura 2.9: Sub-Ontologia de Recurso de Teste

CAPÍTULO

3

Arquitetura Orientada a Serviços

3.1 Considerações Iniciais

As arquiteturas orientadas a serviços oferecem muitas vantagens em termos de separação de interesses, reúso e flexibilidade no desenvolvimento e evolução das aplicações. Nesse estilo arquitetural obtém-se independência de tecnologia, pois é exigido que os serviços sejam definidos por uma linguagem de descrição padronizada e que tenham interfaces com operações de negócio bem definidas. Desse modo, empresas podem criar, implantar e integrar serviços, e compor novas funções e processos de negócio pela combinação de novos e antigos recursos de software.

A proposta deste capítulo é apresentar as principais definições e conceitos de SOA, dando ênfase para o componente registro de serviços que é a proposta deste trabalho. Também são discutidos os benefícios e os desafios desta abordagem de desenvolvimento. O capítulo esta organizado da seguinte forma: na Seção 3.2 são apresentados os componentes e as principais características da arquitetura orientada a serviços, também são apresentados os benefícios introduzidos pela abordagem SOA. Na Seção 3.3 é definido o conceito de serviço Web, são apresentados os principais padrões de implementação e é ilustrado um exemplo de uso. Na Seção 3.4 é realizado um estudo aprofundado sobre o funcionamento e as características do componente registro de serviços.

3.2 Arquitetura Geral

A arquitetura orientada a serviços (*Service Oriented Architecture* - SOA) é uma abordagem de desenvolvimento de sistemas computacionais que aproveita algumas características da orientação a objetos, do desenvolvimento baseado em componentes e dos sistemas distribuídos (Tsai *et al.*, 2005). Nesse contexto, um serviço pode ser visto como um componente, pois são unidades independentes que realizam funções específicas, são descritos e usados somente pela interface e podem ser compostos dinamicamente com outros serviços.

Segundo Ma (2005), a abordagem SOA pretende reduzir problemas relacionados à heterogeneidade, à interoperabilidade e até mesmo à mudança de requisitos. Para atingir esses objetivos a arquitetura tem as seguintes características: baixo acoplamento, apoio à interoperabilidade, descoberta dinâmica, independência de localização e independência de protocolo.

Na Figura 3.1 é ilustrada a SOA e a colaboração entre os componentes provedor (ou fornecedor), consumidor e o registro de serviços. Um provedor oferece um serviço e o publica em um ambiente de publicação/descoberta de serviços (registro de serviços). O consumidor usa o serviço de publicação/descoberta para achar um serviço adequado para interagir e executar alguma tarefa. O registro de serviços oferece funções para armazenar, classificar e localizar serviços registrados (Heckel e Mariani, 2005). Ele funciona de forma semelhante a uma lista telefônica, em que os serviços podem ser registrados e localizados por palavras-chave, tais como nome, funções oferecidas, empresas fornecedoras, etc. Por último, o consumidor também pode conectar-se ao serviço desejado de forma direta, no caso de já conhecer o serviço antecipadamente, ao invés de procurar o serviço no registro (Tsai *et al.*, 2005).

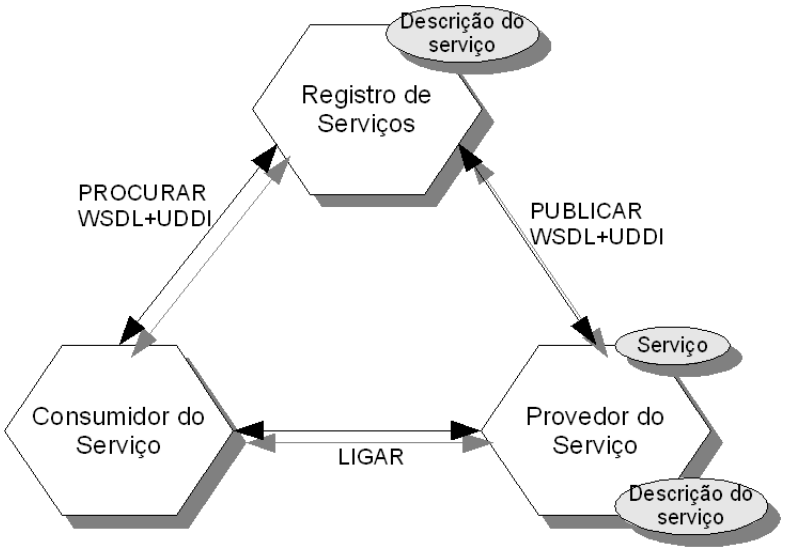

Figura 3.1: Colaboração na Arquitetura Orientada a Serviço (Papazoglou e Heuvel, 2007)

Ort (2005) destaca os seguintes aspectos como os principais benefícios de utilização da abordagem SOA:

Reusabilidade: uma das principais vantagens da aplicação dessa arquitetura, pois novos serviços podem utilizar serviços já existentes para auxiliar na execução de suas atividades.

Interoperabilidade: outro ponto forte da utilização de SOA, que ocorre porque clientes e fornecedores de serviços tem um padrão de comunicação bem definido.

Escalabilidade: devido ao baixo grau de acoplamento dos serviços, as aplicações que os utilizam tendem a fornecer mais escalabilidade, principalmente pela pouca dependência entre a aplicação e os serviços que estão sendo utilizados.

Flexibilidade: em aplicações muito acopladas, diferentes componentes da aplicação dividem as mesmas bibliotecas ou estados, o que dificulta alterações de requisitos do sistema. No entanto, devido ao baixo grau de acoplamento entre os serviços a abordagem oferece uma maior flexibildade.

Eficiência no custo: com todos os benefícios citados anteriormente e pelo fato da maioria das empresas já possuir a infraestrutura de rede necessária para implantar essa arquitetura, o custo é outro benefício importante desta abordagem.

Por fim, Papazoglou (2003) também destaca que os serviços podem ser oferecidos de duas formas: simples ou compostos. Os serviços simples, em geral, oferecem uma funcionalidade específica, enquanto os compostos constituem um processo que integra informações e funções de múltiplos serviços.

3.3 Serviços Web

Apesar da implementação das arquiteturas orientadas a serviços ser independente de tecnologia, a utilização de serviços Web (*Web services*) têm sido a tecnologia de implementação mais popular (Papazoglou e Heuvel, 2007). Segundo Kreger (2001), o que a Web foi para as interações entre aplicação-usuário (B2C), a tecnologia de serviços Web é para as interações entre aplicação-aplicação (B2B). Serviços Web permitem às companhias reduzirem custos de comércio eletrônio, disponibilizar soluções mais rápidas e criar novas oportunidades.

Segundo Josuttis (2007), serviços Web referem-se a uma coleção de padrões projetados para oferecerem interoperabilidade. Estes padrões descrevem tanto o protoloco utilizado para comunicação dos serviços quanto o formato das interfaces utilizadas para especificar os serviços e seus contratos.

3.3.1 Arquitetura de um Serviço Web

Os serviços Web se tornaram populares porque são construídos sobre infraestruturas tais como HTTP, SOAP, WSDL e XML, que são adequadas para a comunicação e integração de sistemas heterogêneos. A Figura 3.2 ilustra a arquitetura de um serviço Web; percebe-se que ela também pode ser vista como uma pilha de protocolos Web.

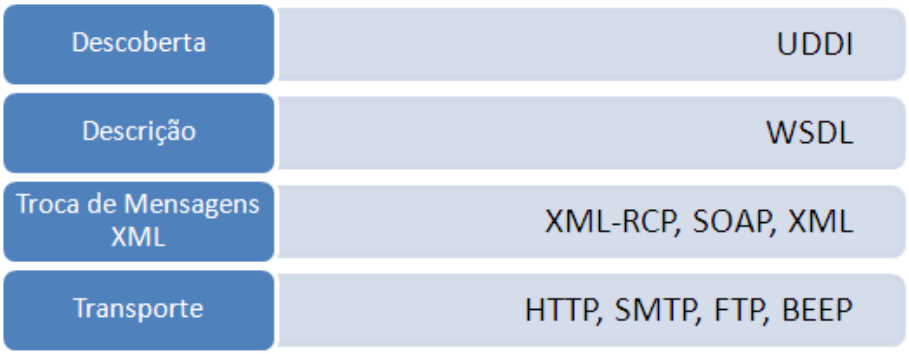

Figura 3.2: Arquitetura de uso de um serviço Web (Cerami, 2002)

Segundo Cerami (2002), essa pilha possui quatro camadas principais: (1) transporte, camada responsável por transportar as mensagens entre as aplicações; (2) troca de mensagem XML, camada responsável por codificar mensagens em um formato XML comum; (3) descrição, camada responsável por descrever a interface pública de um serviço Web específico; e (4) descoberta, camada responsável por centralizar os serviços em um registro comum e fornecer funcionalidades de publicação e busca.

3.3.2 Padrões de Implementação

Conforme discutido na seção anterior, a forma de se implementar serviços Web é por meio de padrões XML que, por sua vez, são independentes de sistema operacional e de linguagem de programação (Ma, 2005). Os principais padrões utilizados para implementar serviços Web são apresentados a seguir.

3.3.2.1 SOAP

Devido ao fato de toda a comunicação entre os serviços ser realizada por meio de mensagens, há necessidade de que a mensagem utilizada seja padronizada para possibilitar a comunicação dos serviços independentemente de sua origem ou implementação. A especificação SOAP foi definida para atingir estes requisitos.

Originalmente definido como *Simple Object Access Protocol* (SOAP, 2003), é o protocolo utilizado para a comunicação entre os serviços Web. Ele define a estrutura das mensagens trocadas entre os serviços e dá suporte a diversos mecanismos de transporte como HTTP e SMTP. Um pacote SOAP é composto das seguintes partes, como pode ser visto na Figura 3.3.

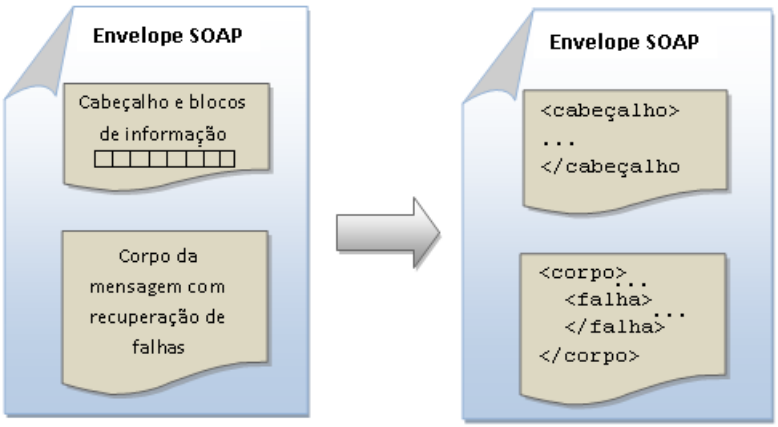

Figura 3.3: Estrutura básica de uma mensagem SOAP

Envelope: é o container responsável por armazenar todas as partes do pacote, definindo o início e o fim das mensagens.

Cabeçalho: é área dedicada a armazenar meta informações do pacote, tais como instruções de processamento ou roteamento.

Corpo: é o conteúdo atual da mensagem, tipicamente consiste de dados formatados em XML.

Falha: oferece às mensagens SOAP a habilidade de tratamento de exceções, pode conter uma resposta ou o código de erro quando uma exceção ocorrer.

Além disso, para facilitar o transporte de dados que não se acomodam facilmente em formato XML, também é possível incluir anexos às mensagens SOAP por meio da tecnologia de *SOAP attachments*. Este mecanismo comumente é utilizado no transporte de arquivos binários, como imagens por exemplo.

3.3.2.2 WSDL

A especificação do padrão WSDL teve início no ano de 2001 pela W3C e foi uma das primeiras iniciativas para apoiar o desenvolvimento dos serviços Web. Ele consiste basicamente de um arquivo de informação que atribui ao serviço uma identidade e permite sua invocação. Mais

precisamente, o padrão *Web Service Description Language* (W3C, 2001) é uma linguagem em XML capaz de descrever todas as informações pertinentes ao serviço. A descrição inclui detalhes como definição de tipos de dados, operações do serviço Web e o formato das mensagens de entrada e saída, entre outros. Dessa forma, o WSDL assume um papel chave no projeto de serviços Web.

Um documento WSDL consiste de duas partes, uma de definição abstrata e outra concreta das interfaces do serviço, conforme ilustra a Figura 3.4. A parte abstrata contém informações sobre os tipo de dados, as mensagem e as operações dos serviços (sua interface). Já a parte concreta inclui os detalhes de ligação com o serviço como o protocolo de comunicação utilizado, o endereço físico de acesso e também características de como as mensagens serão processadas.

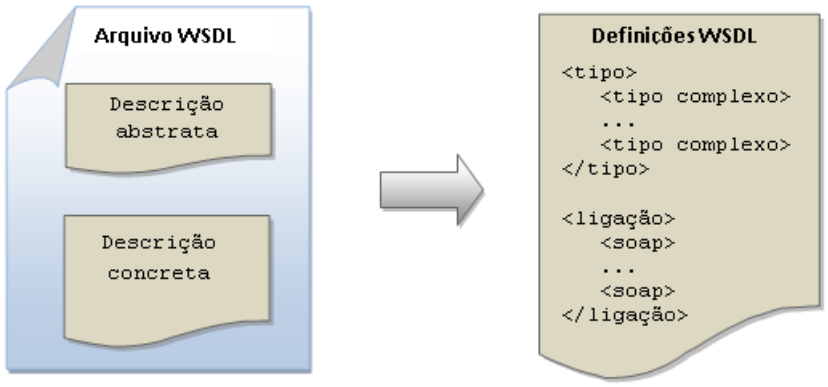

Figura 3.4: Estrutura de um documento WSDL descrevendo a interface de um serviço Web

Além disso, há um atributo opcional no WSDL chamado `documentation` que permite acrescentar informações de documentação sobre o serviço. Esta informação pode ser utilizada por desenvolvedores, pois é uma descrição textual do serviço, e também pode ser automaticamente extraída para um registro de serviço com o intuito de facilitar o processo de descoberta de serviços.

3.3.2.3 UDDI

Esta especificação faz parte da primeira família de padrões de serviços Web. Ela permite a criação de um registro de serviços padronizado, no qual uma empresa ou organização pode armazenar a especificação de seus serviços, tanto para uso próprio quanto para disponibilizar os serviços externamente. Atualmente é o padrão mais utilizado para implementação de registros de serviços.

Os registros UDDI (*Universal Description Descovery and Integration*) (OASIS, 2005) armazenam metadados sobre os serviços Web publicados, de modo a fornecer detalhes dos serviços

para os consumidores, quando requisitados. Além de descrever os serviços Web especificando as interfaces utilizadas, as ferramentas e os tipos de valores do serviço Web, o UDDI também permite que sejam descritas as finalidades dos serviços. A capacidade de descrição da tecnologia UDDI se dá pela utilização de palavras-chave e pequenas *tags* de descrição, possibilitando informar a todos os interessados o que o serviço Web oferece. Dessa forma, o UDDI pode ser utilizado por consumidores que buscam serviços ou provedores de serviços que melhor atendam a suas necessidades (Cerami, 2002).

Segundo Tsai *et al.* (2003), as informações armazenadas em um registro UDDI permitem que os consumidores façam buscas de três tipos:

- **Páginas-brancas:** busca por informações básicas como endereço, contato e identificação de provedores e serviços Web oferecidos.
- **Páginas-amarelas:** busca por serviços Web de um determinado tipo.
- **Páginas-verdes:** busca por informações de um serviço Web específico.

Na Figura 3.5 é apresentado como é organizada uma entrada de um registro de serviços no padrão UDDI. Ela consiste das seguintes partes (Erl, 2005):

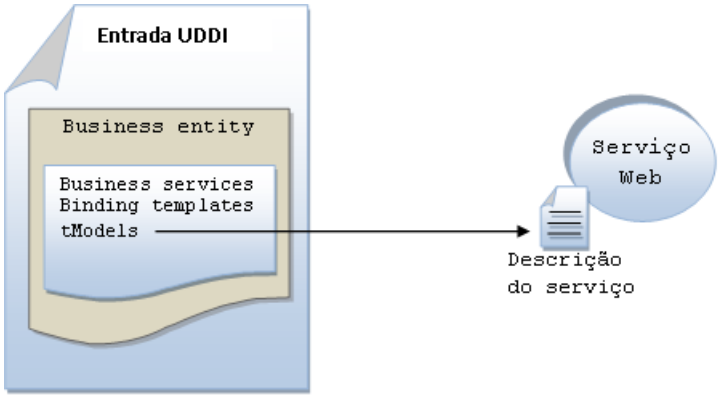

Figura 3.5: Estrutura básica de uma entrada de negócio em um registro UDDI

- *Business entity* (provedor de serviços): armazena um perfil básico de informações sobre a organização ou o provedor de serviços, incluindo informações de contato e descrição sobre serviços oferecidos. É composta de um ou mais *business services*.
- *Business service* (serviço): consiste de uma descrição detalhada dos serviços oferecidos pelo provedor de serviços à qual pertence, ou seja, representa os atuais serviços oferecidos pelo provedor.

- *Binding template* (gabarito de invocação): armazena os detalhes de invocação/ligação com os serviços Web e também ponteiros para informações de implementação. Seguindo a mesma lógica dos arquivos WSDL, no caso dos registros, os detalhes de chamada do serviço também são armazenados de maneira separada.

- *tModels* (modelos técnicos): disponibiliza ponteiros para as descrições dos serviços registrados no repositório e contém detalhes técnicos sobre como os consumidores podem interagir com os serviços Web disponíveis. Estes detalhes podem incluir o formato das mensagens trocadas e protocolos de segurança utilizado, por exemplo.

Exemplo

A título de ilustração dos conceitos apresentados, considere uma organização que ofereça um serviço Web para consultar cotação de ações com atraso e que deseja registrar seu serviço em um sistema de registro. Após efetuar o registro no repositório UDDI, tem-se que a seguinte entrada XML é armazenada no repositório, conforme ilustra a Listagem 3.1 (Møller e Schwartzbach, 2006).

A partir da linha 1 descrevem-se os atributos da tag *businessEntity*. O *businessKey* é um identificador único da organização no repositório, das linhas 4 a 6 encontra-se o atributo *discoveryURL*, que se refere ao endereço utilizado para localizar o documento XML. Além disso, outros atributos são o nome da empresa provedora do serviço – *XMethods* (linha 10), sua descrição na linha subsequente e suas informações de contato, tais como email, telefone, etc.

A partir da linha 13, o atributo *businessServices* descreve os serviços oferecidos pela organização, que no caso contém apenas o serviço de cotação. O serviço também possui um identificador único, dado pelo atributo *serviceKey*, uma referência a organização à qual pertence (linha 14), além de seu nome e descrição (linhas 16 a 19).

Por último, entre as linhas 20 e 33, o atributo *bindingTemplate* contém os detalhes de invocação do serviço. Do mesmo modo, ele possui um identificador único (linha 21) e uma referência ao serviço a qual pertence (linha 22). Por meio dos atributos *accessPoint* (linhas 26 a 28) e *description* (linhas 23 a 25), são conhecidas a localização de acesso do serviço e sua descrição. O atributo *tModelInstanceDetails* contém um ponteiro (linha 30) para a entidade *tModel*, ilustrada na Listagem 3.2.

Listagem 3.1: Exemplo de uma entrada XML no registro UDDI

```
1 <businessEntity businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64"
2                 operator="www.ibm.com/services/uddi"
3                 authorizedName="0100001QS1">
4   <discoveryURLs>
```

```
5     <discoveryURL useType="businessEntity">
6       http://www.ibm.com/services/uddi/uddiget?
7       businessKey=BA744ED0-3AAF-11D5-80DC-002035229C64
8     </discoveryURL>
9   </discoveryURLs>
10  <name>XMethods</name>
11  <description xml:lang="pt-BR">Site provedor de serviços Web</description>
12  <contacts> ... </contacts>
13  <businessServices>
14    <businessService serviceKey="d5921160-3e16-11d5-98bf-002035229c64"
15                     businessKey="ba744ed0-3aaf-11d5-80dc-002035229c64">
16      <name>Serviço de cotação de ações com atraso XMethods</name>
17      <description xml:lang="pt-BR">
18          Cotação de ações nos últimos 10 minutos
19      </description>
20      <bindingTemplates>
21        <bindingTemplate bindingKey="d594a970-3e16-11d5-98bf-002035229c64"
22                         serviceKey="d5921160-3e16-11d5-98bf-002035229c64">
23          <description xml:lang="pt-BR">
24               Serviço Web para consulta de ações por meio de mensagens SOAP
25          </description>
26          <accessPoint URLType="http">
27               http://services.xmethods.net:80/soap
28          </accessPoint>
29          <tModelInstanceDetails>
30            <tModelInstanceInfo tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64" />
31          </tModelInstanceDetails>
32        </bindingTemplate>
33      </bindingTemplates>
34    </businessService>
35  </businessServices>
36 </businessEntity>
```

No caso da Listagem 3.2, da mesma maneira que a *businessEntity*, a entidade *tModel* possui atributos como identificador único, nome e descrição. No entanto, o que merece destaque é que ela aponta para a interface WSDL do serviço registrado no repositório UDDI, conforme pode ser visto pelo atributo *overviewURL*(linhas 8 a 10).

Listagem 3.2: Entidade tModel do exemplo anterior

```
1  <tModel tModelKey="uuid:0e727db0-3e14-11d5-98bf-002035229c64"
2          operator="www.ibm.com/services/uddi"
3          authorizedName="0100001QS1">
4    <name>XMethods cotação de ações simples</name>
5    <description xml:lang="pt-BR">Interface WSDL do serviço de cotações</description>
6    <overviewDoc>
7      <description xml:lang="pt-BR">Link WSDL</description>
8      <overviewURL>
9        http://www.xmethods.net/tmodels/SimpleStockQuote.wsdl
10     </overviewURL>
11   </overviewDoc>
12   <categoryBag>
13     <keyedReference tModelKey="uuid:c1acf26d-9672-4404-9d70-39b756e62ab4"
```

```
14                    keyName="uddi-org:types"
15                    keyValue="wsdlSpec" />
16   </categoryBag>
17 </tModel>
```

Em síntese, para exemplificar uma interação com o registro UDDI, a mensagem SOAP da Listagem 3.3 poderia ser enviada para se realizar uma busca sobre serviços com o termo "cotação de ações com atraso" no nome. É importante ressaltar-se que, as descrições dos serviços armazenados no padrão UDDI podem ser dinamicamente descobertas tanto por aplicações quanto por humanos.

Listagem 3.3: Mensagem SOAP para consultar serviços de cotação de ações

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
3   <Body>
4     <find_service businessKey="*" generic="1.0" xmlns="urn:uddi-org:api">
5       <name>cotação de ações com atraso</name>
6     </find_service>
7   </Body>
8 </Envelope>
```

3.3.3 Exemplo de Utilização de um Serviço Web

Com o objetivo de ilustrar a utilização de serviços Web, é apresentado a seguir um exemplo de um serviço Web que informa as condições climáticas de um dado CEP[1]. Na Figura 3.6 é apresentado o uso do serviço Web de informações sobre o clima por um consumidor, que pode ser qualquer programa que tenha acesso ao serviço. Nesse exemplo, o consumidor conhece a localização do serviço Web utilizado, o que caracteriza um maior acoplamento entre consumidor e provedor.

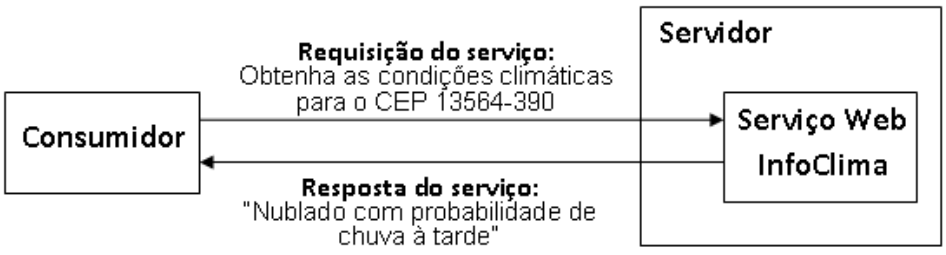

Figura 3.6: Uso de um serviço Web

As arquiteturas orientadas a serviços permitem a construção de sistemas com baixo acoplamento, sem que o consumidor conheça a localização do serviço Web a ser invocado. Na

[1]Parte desta seção foi baseada no trabalho de Eler (2008)

Figura 3.7 é apresentado um caso em que o consumidor precisa de um registro de serviços para descobrir a localização do serviço Web requerido. A seguir são apresentados em mais detalhes cada uma das invocações e respostas entre consumidor e serviço Web:

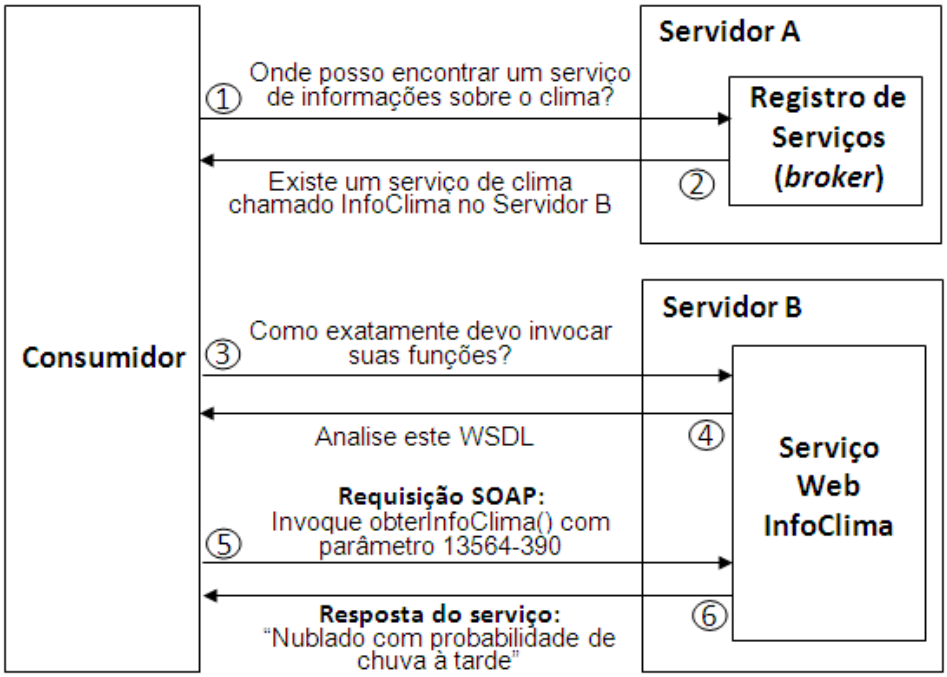

Figura 3.7: Uso de um serviço Web por meio do registro

1. O consumidor não sabe qual serviço será invocado. O primeiro passo então é consultar um serviço de publicação/descoberta que fornecerá a localização de um serviço do tipo procurado. Na listagem 3.4 é apresentado um exemplo de busca em um registro UDDI por um serviço de informação sobre o clima. Primeiramente é escolhido um registro (linha 3) e em seguida é realizada uma busca por serviços Web que tenham "clima" no nome (linha 4). Ao obter a lista de serviços Web que correspondem àquele nome (linha 5), mostra-se, para cada serviço, seu nome (linha 10) e o endereço para acesso (linha 22).

Listagem 3.4: Código para localizar o serviço Web de clima em um repositório UDDI

```
1  public static void main(String [] args){
2       UDDIProxy proxy = new UDDIProxy();
3       proxy.setInquiryURL("http://uddi.sap.com/uddi/api/inquiry/");
4       ServiceList     srvList = proxy.find_service(null, "clima", null, 0);
5       Vector srvVector = srvList.getServiceInfos().getServiceInfoVector();
6
7       for (int k=0; k<srvVector.size(); k++)
8       {
9            ServiceInfo     srvInfo = (ServiceInfo) srvVector.get(k);
10           System.out.println(srvInfo.getNameString());
11           String srvKey = srvInfo.getServiceKey();
12           ServiceDetail  srvDetail = proxy.get_serviceDetail(srvKey);
13           Vector bsrvVector = srvDetail.getBusinessServiceVector();
14           for (int i=0; i<bsrvVector.size(); i++)
15           {
16                BusinessService bsrv = (BusinessService) bsrvVector.get(i);
17                BindingTemplates bdtmpls = bsrv.getBindingTemplates();
18
19                for(int    j=0; j<bdtmpls.size();    j++)
20                {
21                     BindingTemplate bdtmpl = bdtmpls.get(j);
22                     System.out.println(bdtmpl.getAccessPoint().getText());
23                }
24           }
25      }
26 }
```

2. O serviço de publicação/descoberta informará ao consumidor onde encontrar um serviço do tipo procurado. A localização do serviço é expressa por meio de um URI (Uniform Resource Identifier) que, de forma semelhante a um URL (Uniform Resource Locator), consiste de um identificador único que informa onde está localizado o serviço Web. Uma resposta possível seria http://labes.br/InfoClima.

3. Neste momento o consumidor sabe a localização do serviço a ser invocado, mas não sabe como invocá-lo, pois a operação que retornará as condições climáticas pode se chamar tanto obterClima(String) quanto obterCondicoesClimaticas(String). Para saber

exatamente a forma de solicitar o serviço é necessário pedir ao serviço uma descrição de sua interface para que se possa conhecer suas operações.

4. O serviço Web responde e envia para o consumidor um arquivo WSDL. O arquivo WSDL do serviço InfoClima pode ser visto na listagem 3.5. Nele estão informados os tipos envolvidos (linhas 5 a 22), as mensagens trocadas (linhas 23 a 28), as operações (linhas 29 a 34), os protocolos de comunicação e transporte (linhas 35 a 46) e o endereço do serviço Web (Linhas 47 a 51).

Listagem 3.5: Código em WSDL de InfoClima

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tns="http://www.
3 labes.br/InfoClima/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.
4 w3.org/2001/XMLSchema" name="InfoClima" targetNamespace="http://www.labes.br/InfoClima/">
5     <wsdl:types>
6         <xsd:schema targetNamespace="http://www.labes.br/InfoClima/">
7             <xsd:element name="obterInfoClima">
8                 <xsd:complexType>
9                     <xsd:sequence>
10                        <xsd:element name="CEP" type="xsd:string" />
11                    </xsd:sequence>
12                </xsd:complexType>
13            </xsd:element>
14            <xsd:element name="obterInfoClimaResponse">
15                <xsd:complexType>
16                    <xsd:sequence>
17                        <xsd:element name="clima" type="xsd:string"/>
18                    </xsd:sequence>
19                </xsd:complexType>
20            </xsd:element>
21        </xsd:schema>
22    </wsdl:types>
23    <wsdl:message name="obterInfoClimaRequest">
24        <wsdl:part element="tns:obterInfoClima" name="parameters"/>
25    </wsdl:message>
26    <wsdl:message name="obterInfoClimaResponse">
27        <wsdl:part     element="tns:obterInfoClimaResponse" name="parameters"/>
28    </wsdl:message>
29    <wsdl:portType name="InfoClimaInterface">
30        <wsdl:operation name="obterInfoClima">
31            <wsdl:input message="tns:obterInfoClimaRequest"/>
32            <wsdl:output message="tns:obterInfoClimaResponse"/>
33        </wsdl:operation>
34    </wsdl:portType>
35    <wsdl:binding name="InfoClimaSOAP" type="tns:InfoClimaInterface">
36        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
37            <wsdl:operation name="obterInfoClima">
38        <soap:operation soapAction="http://www.labes.br/InfoClima/obterInfoClima"/>
39        <wsdl:input>
40            <soap:body use="literal"/>
```

```
41            </wsdl:input>
42            <wsdl:output>
43                 <soap:body use="literal"/>
44            </wsdl:output>
45       </wsdl:operation>
46 </wsdl:binding>
47 <wsdl:service name="InfoClimaService">
48       <wsdl:port      binding="tns:InfoClimaSOAP" name="InfoClimaSOAP">
49            <soap:address location="http://www.labes.br/"/>
50       </wsdl:port>
51 </wsdl:service>
52 </wsdl:definitions>
```

5. Com o WSDL, o consumidor pode descobrir como invocar o serviço Web de informação sobre o clima. Ferramentas podem automatizar este processo e gerar stubs que representam o serviço. As chamadas às operações são realizadas da mesma forma que os métodos de um objeto são chamados. As invocações às operações do serviços Web são traduzidas em mensagens SOAP. A mensagem SOAP da chamada ao serviço InfoClima pode ser vista na listagem 3.6.

Listagem 3.6: Mensagem SOAP Enviada

```
1 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
2   xmlns:inf="http://labes.com.br/InfoClima/" >
3      <soapenv:Header />
4      <soapenv:Body>
5           <inf:obterInfoClima>
6                <CEP>13566-580</CEP>
7           </inf:obterInfoClima>
8      </soapenv:Body>
9 </soapenv:Envelope>
```

6. O serviço Web de clima receberá a mensagem SOAP e enviará uma resposta SOAP contendo as condições climáticas do CEP enviado ou uma mensagem de erro se a requisição SOAP for incorreta. A mensagem SOAP de resposta de InfoClima pode ser vista na listagem 3.7.

Listagem 3.7: Mensagem SOAP de Resposta

```
1 <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
2      <soapenv:Header />
3           <inf:obterInfoClimaResponse>
4                <clima>Nublado e com probabilidade de chuva à tarde</clima>
5           </inf:obterInfoClimaResponse>
6      </soap:Body>
7 </soapenv:Envelope>
```

3.4 Registro de Serviços

Os registros de serviços (*service broker*) em SOA são um importante mecanismo que tem por objetivo fornecer meios para facilitar o processo de registro, publicação e descoberta de serviços. O sistema de registro conecta e interage com provedores e consumidores de serviços. Por meio deste recurso, é possível que um consumidor encontre o serviço desejado. Segundo Al-Masri e Mahmoud (2008), sem a publicação de serviços Web nos registros, os usuários não serão capazes de localizar os serviços de maneira eficiente e os provedores terão que dedicar um esforço extra para publicar seus serviços por outros meios de divulgação.

Erl (2005) destaca que o único requisito para um serviço Web contatar outro serviço Web é o acesso à descrição do serviço. Assim, conforme a quantidade de serviços aumenta em uma organização, mecanismos para descobrir e publicar a sua descrição tornam-se uma necessidade. Nesse contexto, esta seção tem por objetivo detalhar os diversos aspectos relacionados ao registro de serviços. São apresentadas suas características e funcionamento, os padrões de implementação, as ferramentas que permitem o desenvolvimento de sistemas de registro e, por fim, apresenta-se um estudo sobre registros de domínio público.

3.4.1 Características e Funcionamento

Conforme discutido anteriormente, o registro de serviços oferece uma maneira de aproximar provedores e consumidores. Nesse sentido, assim que os provedores desenvolvem os serviços é necessário armazená-los em um local comum e bem conhecido. Desse modo, sempre que um consumidor necessita de um serviço ele pode realizar uma busca no registro a fim de encontrar o recurso desejado.

Segundo Josuttis (2007), os sistemas de registro podem ser de dois tipos:

- *Registro público*: aceita o registro a partir de qualquer organização, independentemente de ela atuar ou não como uma provedora de serviços.
- *Registro privado*: desenvolvido dentro do contexto de uma organização com o objetivo de oferecer um repositório central para os serviços que a organização desenvolve ou disponibiliza.

O registro de serviços (intermediário ou *broker*) possui um papel importante no gerenciamento de tais recursos, pois permite, a usuários ou aplicações acessar uma ampla quantidade de serviços. Por meio do registro é possível que consumidores, isto é, serviços Web ou inclusive humanos, possam localizar as últimas versões das descrições de um serviço conhecido

e descobrir novos serviços que satisfaçam determinado a critério. Este processo é ilustrado sucintamente na Figura 3.8.

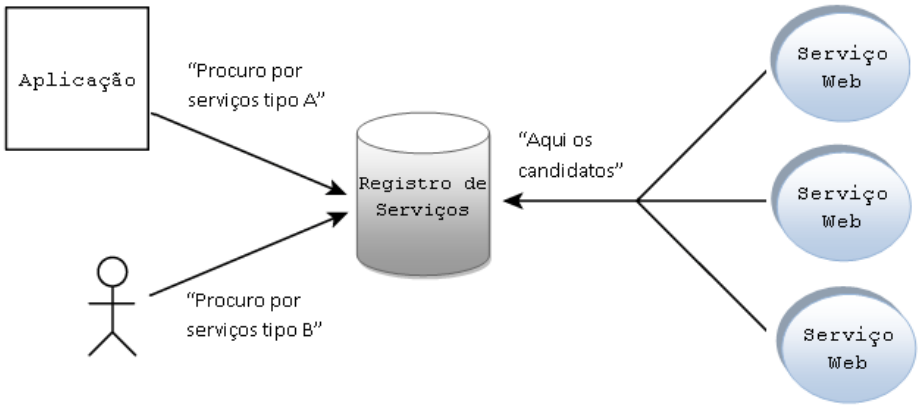

Figura 3.8: Localização dos serviços centralizada em um registro (Adaptado de Erl (2005))

Por meio do processo de registro, os serviços desenvolvidos no contexto de uma organização podem ser reutilizados. Além do reúso, outra vantagem dos registros é o fato de possibilitar o baixo acoplamento entre invocações de serviços Web. Por exemplo, ao invés de especificar o endereço de um serviço Web estaticamente no código, o endereço do provedor pode ser requisitado de um *broker* em tempo de execução.

É importante esclarecer que os sistemas de registro podem assumir diferentes papeis. Os usos mais comuns atualmente são os de *repositórios* e *registros*, termos muitas vezes utilizados de maneira intercambiável. Segundo Josuttis (2007), as diferenças são as seguintes:

- *Repositório*: gerencia serviços e seus artefatos de uma perspectiva do negócio. Ou seja, armazena comportamento, interfaces, contratos, SLA e dependências, com o objetivo de auxiliar no projeto e desenvolvimento dos serviços. Esse tipo de informação deve ser independente de aspectos técnicos, de tal forma que, caso a organização queira mudar sua infraestrutura, o repositório deve permanecer o mesmo.

- *Registro*: gerencia serviços do ponto de vista técnico. Ou seja, é responsável por fornecer todos os detalhes técnicos tais como, informações de implantação (*deployment*) e endereços URLs, necessários para usar os serviços em tempo de execução.

Vale ressaltar, em relação aos arquivos de descrição WSDL, que eles são um formato padrão para especificar informações técnicas sobre como invocar os serviços Web. Por isso, não faz sentido utilizar este formato para os repositórios de serviços, mas sim nos registros (Josuttis, 2007). Em linhas gerais, repositórios e registros são importantes para gerenciar os serviços tanto do ponto de vista técnico como organizacional.

3.4.2 Padrões de Implementação

Diversos padrões de implementação têm sido definidos para regular o desenvolvimento de sistemas de registro. Dentre os que merecem destaque atualmente pode-se citar o UDDI e o ebXML, apresentados brevemente a seguir.

O padrão UDDI oferece uma maneira para descoberta de serviços Web por meio de uma arquitetura centralizada. Ele possui um modelo de dados estruturado hierarquicamente para organização de informações. Atualmente, suas versões de especificação são a 2.0 e a 3.0, sendo que esta última introduziu alguns avanços como o suporte a várias linguagens e o apoio a mecanismos de segurança por meio de assinatura digital (Tsai *et al.*, 2007)

O ebXML (*Electronic Business using eXtensible Markup Language*) (OASIS, 2003) é uma família de padrões baseados em XML, cujo desenvolvimento teve início em 1999. Ele inclui um padrão para especificar registros e repositórios. Seu objetivo é fornecer uma infraestrutura aberta, baseada em XML, para armazenar metadados das entradas do registro. Ressalta-se que um registro ebXML oferece um conceito semelhante ao UDDI. Entretanto, é mais amplo por ser capaz de modelar entradas arbitrárias, como diagramas UML, por exemplo.

3.4.3 Extensões dos Registros de Serviços

Um registro de serviços, além de oferecer suporte à publicação de serviços, também pode armazenar metadados, ontologias, *scripts* de teste e possuir uma interface com o usuário. Nesse cenário, Tsai *et al.* (2007) propõem algumas extensões à especificação UDDI com o objetivo de permitir a inclusão desses recursos, como uma tentativa de prover serviços mais confiáveis. Os autores classificam essas extensões em quatro categorias: entidades armazenadas, interfaces, estilo arquitetural e organização. Na Tabela 3.1 apresenta-se um detalhamento mais completo de todas as categorias.

Um ponto importante a destacar em relação às entidades armazenadas refere-se ao uso de ontologias, pois elas acrescentam informação semântica aos serviços catalogados, facilitando assim o processo de busca e descoberta de serviços catalogados. Devido ao fato de que o uso de apenas palavras-chave nas buscas pode trazer resultados pouco significativos para o usuários.

Em relação às interfaces do registro, um ponto importante refere-se a interfaces de verificação e validação, pois a disponibilização de tais recursos permite o teste das entidades armazenadas no registro, aprimorando assim sua funcionalidade básica. O teste pode ser realizado tanto no momento de registrar um novo serviço no repositório quanto por um consumidor interessado em utilizar um serviço em sua aplicação.

Tabela 3.1: Extensões propostas ao modelo UDDI atual (Tsai *et al.*, 2007)

Entidades	Interfaces	Arquitetura	Organização
Serviços	Consulta/Inscrição	Confiável/ Tolerante a falhas	Centralizada
Templates	Publicação	Segura	Distribuída
Scripts de Teste	Verificação/ Validação	Reconfigurável	Federada
Protocolos	Simulação	Adaptativa	Hierárquica
Ontologia	Visualização	Tempo real	
Especificações	Monitoração	Autônoma	
Perfis	Segurança		

Por último, em relação às arquiteturas dos registros, com o intuito de auxiliar na prevenção de falhas, pode ser utilizado um mecanismo de replicação de dados, tanto internamente quanto externamente, em um servidor remoto. Outro aspecto importante que pode aprimorar o desempenho das buscas se dá pelo fato de que os dados ou consultas realizadas com mais frequência podem ser armazenados em memória *cache* e posteriormente carregados, caso uma nova busca semelhante seja feita.

3.4.4 Implementações

Conforme discutido na Seção 3.4.2, os dois maiores padrões que especificam e regulam o desenvolvimento dos sistemas de registro são UDDI e ebXML. Nesse sentido, com o objetivo de oferecer suporte a estes padrões, tem surgido diversas implementações tanto de código aberto quanto comerciais. Nesta seção são apresentados alguns exemplos dessas plataformas.

3.4.4.1 Código Aberto

Dentre as implementações de código aberto, pode-se destacar a ferramenta **jUDDI** (pronuncia-se "Judy") (Apache jUDDI, 2004). Ela é uma implementação Java da especificação UDDI e possui as seguintes características: oferece independência de plataforma; é compatível com qualquer banco de dados relacional que implemente o padrão SQL, por exemplo MySQL ou PostgreSQL; pode ser implantada em qualquer servidor de aplicação Java, tais como Apache Tomcat, WebSphere, JRun, dentre outros e é compatível com a versão 2.0 ou 3.0 da especificação UDDI.

Outra implementação em conformidade com o padrão UDDI é o **Novell UDDI Server** (Novell, 2007). Ele foi projetado para ser um registro privado, a ser utilizado no contexto de uma organização. Possui como características o fato de oferecer independência de plataforma e ser

disponibilizado em dois modos: servidor e cliente. O modo cliente consiste de uma interface gráfica para gerenciar os serviços armazenados.

O **OpenUDDI Server** (OpenUDDI, 2008) é uma implementação baseada em Java que também implementa o padrão UDDI. Seu desenvolvimento tem como base o Novell UDDI Server descrito no parágrafo anterior, que foi estendido acrescentando diversas novas funcionalidades. Possui como características a possibilidade de ser executado em qualquer servidor de aplicação; oferecer uma biblioteca de acesso ao lado do servidor e outra cliente com APIs de publicação, consulta, assinatura e segurança; e também oferece suporte à persistência baseada em Hibernate.

Por fim, o **freebXML Registry** (OASIS, 2007) é uma iniciativa criada para apoiar o desenvolvimento e adoção da especificação ebXML. É também ferramenta de código aberto composta por um registro e por um repositório de serviços. Possui como uma das principais características a possibilidade de armazenar qualquer tipo de dado no repositório de serviços, incluindo descrições WSDL, arquivos XML ou dados binários.

3.4.4.2 Comerciais

O **Enterprise UDDI Services** (Windows UDDI Services, 2003) consiste de uma infraestrutura dinâmica e flexível para gerencia de serviços Web. Ele está incluído na distribuição do Windows Server 2003 e permite que companhias gerenciem seu diretório de serviços tanto para uso interno quanto externo à organização. Possui uma implementação servidor e também uma cliente. Dentre suas características destaca-se a facilidade de organização dos serviços Web, pois oferece esquemas de localização por geografia e por qualidade de serviço (*QoS*).

Por último, o **Oracle Service Registry** (Oracle, 2008) é uma plataforma para publicação, categorização e descoberta de serviços e outros recursos relacionados. Ele faz parte de uma solução completa de SOA da Oracle e também implementa o padrão UDDI. Podem-se destacar as seguintes vantagens: possui uma interface gráfica com facilidade de navegação; gerencia metadados de maneira extensível; e oferece métricas e estatísticas sobre os serviços registrados.

3.4.5 Registros Públicos

A fim de apresentar um panorama atual dos sistemas de registro, a seguir são abordados alguns registros de domínio público. Ressalta-se que a quantidade de sistemas dessa natureza é pequena atualmente.

3.4.5.1 eSigma

O portal *eSigma* (eSigma, 2003) é uma plataforma para apoiar o processo de descoberta, publicação e gerência dos serviços Web, cujo desenvolvimento foi iniciado em 2003. Ele oferece um repositório de serviços Web, com operações para publicar, consumir e gerenciar os serviços. O site possui opção de busca por palavras-chave e permite navegação de serviços por meio de categorias. Na descrição do serviço Web são mostrados detalhes do serviço e do provedor. Uma característica interessante é a possibilidade de se realizar o teste funcional dos serviços disponíveis. O teste é feito por meio da invocação do serviço e análise da mensagem SOAP enviada/recebida. A Figura 3.9 ilustra a página de busca da ferramenta.

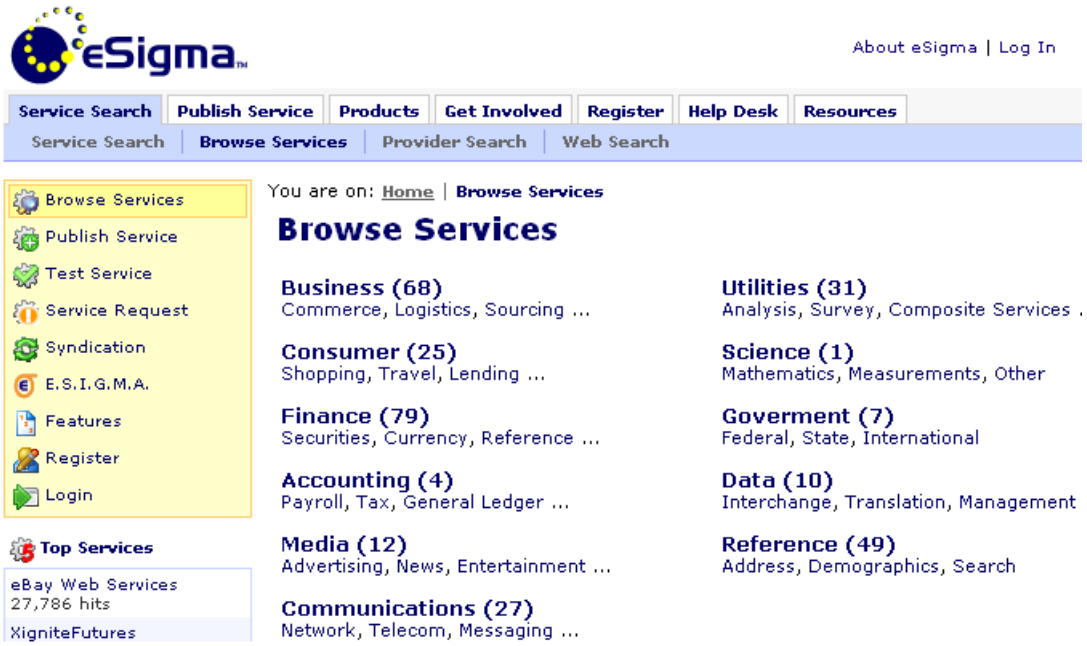

Figura 3.9: Página principal de busca de serviços no registro *eSigma*

3.4.5.2 Xmethods

O *Xmethods* (Xmethods, 2008) é um sistema de registro onde são listados serviços Web de domínio público. Ele atua mais como um catálogo de serviços, pois são apenas listados o serviço, o provedor, uma descrição e a tecnologia de implementação utilizada, conforme ilustra a página inicial da ferramenta na Figura 3.10.

Além disso, o sistema oferece uma ferramenta denominada *WSDL analyzer*, que verifica o conteúdo de um WSDL validando o documento XML a fim de encontrar inconsistências. Como desvantagem da ferramenta pode-se citar que o sistema não monitora os serviços catalogados, deixando essa tarefa a cargo do provedor do serviço.

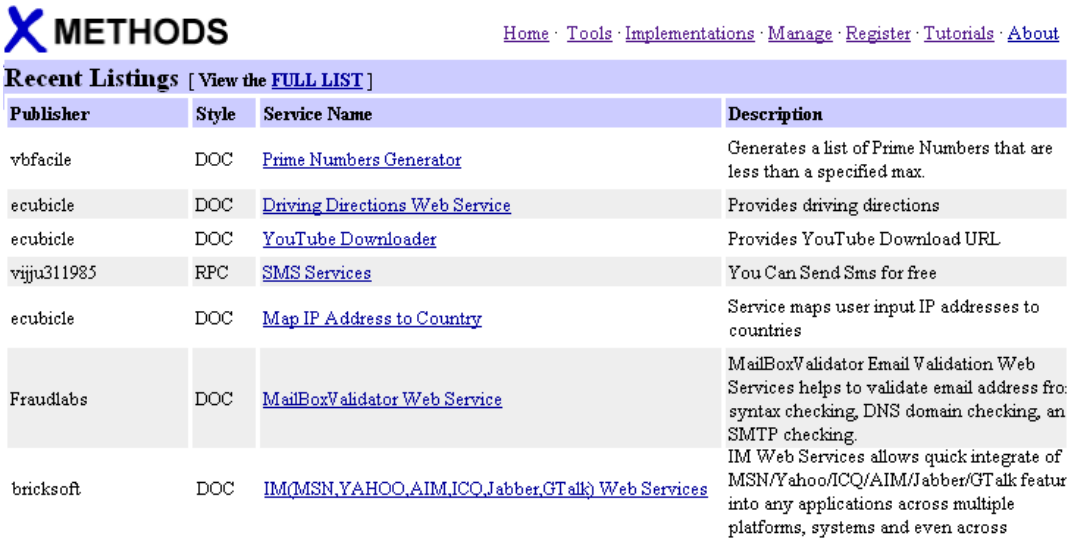

Figura 3.10: Página inicial do repositório de serviços *XMethods*

3.4.5.3 Seekda

O portal *Seekda* (Seekda, 2006) é um sistema que permite a busca de serviços Web públicos. Ele possui um catálogo de milhares de serviços de propósito geral como, por exemplo, validação de email, tradução de texto, serviços de temperatura, etc. O registro tem como objetivo facilitar o uso de serviços Web operando como um *search engine* de serviços Web que estão publicamente disponíveis. Ele permite a busca por meio de palavras-chave ou pela navegação por categorias previamente definidas, conforme ilustra a Figura 3.11.

Figura 3.11: Página de busca do registro de serviços *Seekda*

Como principais características destacam-se: em relação aos consumidores, o registro oferece uma interface que permite realizar o teste funcional e verificar a disponibilidade do serviço Web. Já em relação aos provedores, o registro possui um mecanismo para submissão de serviços; possibilita a interação com os usuários de um determinado provedor; e permite a monitoração do tempo de vida dos serviços.

3.5 Considerações Finais

Neste capítulo foram discutidos os principais conceitos referentes às arquiteturas orientadas a serviços e ao componente de registro. Pode-se perceber que aplicações SOA atuais apoiam e encorajam o mecanismo de publicação e descoberta de serviços. Além disso, uma aplicação em conformidade com os princípios de SOA deve contar com algum mecanismo de registro ou um diretório para gerenciar seus serviços (Erl, 2005). No entanto, um problema que ocorre nos sistemas de registro refere-se a buscas por palavras-chave, pois elas são basicamente sintáticas e podem trazer resultados pouco significativos para o usuários. Isso faz com que pesquisas sobre o desenvolvimento de serviços de registro que incluam opções de busca por meio de ontologia sejam incentivadas.

CAPÍTULO

4

Uma Arquitetura Baseada em Serviços para o Domínio de Engenharia de Software

4.1 Considerações Iniciais

Este capítulo tem por objetivo apresentar uma arquitetura genérica baseada em serviços para o domínio de engenharia de software. A arquitetura tem como objetivo descrever como ferramentas de engenharia de software podem ser adaptadas para a abordagem SOA e com isso auxiliar no entendimento e implementação do registro de serviços específico para ferramentas de teste proposto. É apresentada uma instanciação da arquitetura para o subdomínio de teste de software e também é discutido como esta pode ser instanciada para outro domínio ou subdomínio em particular.

O capítulo inicia com a apresentação de algumas vantagens e apresenta alguns trabalhos referentes à disponibilização de ferramentas de engenharia de software como serviços na Seção 4.2. Em seguida, na Seção 4.3, são apresentados os detalhes da arquitetura baseada em serviços para ferramentas de engenharia de software. Posteriormente, na Seção 4.4 são fornecidas informações específicas referentes à instanciação da arquitetura para o domínio particular de ferra-

mentas de teste de software. Por último, na Seção 4.5 são apresentadas as considerações finais do capítulo.

4.2 Ferramentas de Engenharia de Software como Serviços

Recentemente, pesquisas vêm sendo conduzidas para a disponibilização de ferramentas de engenharia de software como serviços. A ideia geral é criar uma plataforma colaborativa de ferramentas de engenharia de software que ofereça suporte à interoperabilidade de ferramentas, e que seja independente de fronteiras geográficas e organizacionais (Ghezzi e Gall, 2008). O uso de ferramentas como serviços pode contribuir para resolução de alguns problemas de ferramentas de engenharia de software tradicionais. Algumas vantagens do uso de serviços são:

- Serviços podem ser integrados tanto em ambientes de desenvolvimento comerciais como em ambientes de software livre (código aberto).
- Serviços dispensam a necessidade de instalações locais e manutenção por parte dos usuários.
- Serviços são independentes de plataforma, sistemas operacionais e linguagens de programação.
- Serviços podem ser mais facilmente comparados e avaliados.

Nesse contexto, ressalta-se que, o desenvolvedor do serviço torna-o disponível para uso gratuito ou pago e continua atualizando o serviço sem a necessidade que seus usuários tenham que instalar novamente uma nova versão a cada atualização.

Dentre as pesquisas existentes para criação de ferramentas de engenharia de software como serviços, um trabalho na área de análise de software é o de Ghezzi e Gall (2008). Nele os autores propõem o uso de ferramentas de análise de programas como serviços com o objetivo de extrair e analisar informações de artefatos, tais como: código fonte e modelos. Para apoiar o uso das ferramentas como serviços, os autores desenvolveram um registro de serviços onde ferramentas de análise de software são registradas e tornam-se parte de um catálogo para serem encontradas pelos usuários. Uma ontologia de análise de software também foi desenvolvida e incorporada ao registro.

Em particular no domínio de teste de software, alguns pesquisadores têm proposto ferramentas de teste de software como serviços. Além de oferecer apoio ao teste de programas, uma motivação adicional é oferecer apoio ao teste de serviços, o que é difícil, considerando que o código-fonte do serviço em geral não está disponível para o testador realizar a instrumentação.

Eler *et al.* (2009) propõem a disponibilização de uma ferramenta de teste estrutural como serviço. O serviço desenvolvido denomina-se JaBUTiWS e ele realiza a instrumentação do serviço e o cálculo de cobertura de maneira automática. A abordagem é genérica e pode ser aplicada a programas ou serviços em qualquer linguagem de programação.

Bartolini *et al.* (2009) propõem um *framework* baseado em SOA que inclui um serviço denominado TCOV usado como ferramenta de apoio ao teste estrutural de serviços. Nesse *framework* o desenvolvedor do serviço deve instrumentá-lo e invocar uma operação do TCOV para criar um serviço testável. Posteriormente, o serviço testável é executado por um integrador e com auxílio do TCOV, calcula-se a cobertura alcançada para diversos critérios, tais como: todos-nós e todas-arestas.

Percebe-se que a disponibilização de ferramentas como serviços, especialmente de engenharia de software, é uma área de grande interesse e tem aumentado o número ferramentas desenvolvidas. A seguir é detalhada uma arquitetura genérica para ferramentas de engenharia de software no contexto de SOA.

4.3 SOAES: Uma Arquitetura Baseada em Serviço para o Domínio de Engenharia de Software

Uma arquitetura baseada em serviços para o domínio de engenharia de software, em particular teste de software, envolve os três componentes básicos de SOA: serviços de engenharia de software, que são ferramentas de engenharia de software desenvolvidas como serviços; aplicações, que são softwares que estão sendo desenvolvidos no contexto de um processo de desenvolvimento e usam as ferramentas de engenharia de software de modo manual ou num processo de *workflow* automatizado; e um registro de serviços, que oferece facilidades para armazenamento e busca de uma forma centralizada e padronizada. Além disso, um fator importante que caracteriza a arquitetura é o uso de ontologias de domínio específico. Uma visão geral da arquitetura proposta é ilustrada na Figura 4.1.

Os serviços, nesse contexto, são ferramentas de engenharia de software implementadas como serviços [1]. Eles podem apoiar qualquer fase do processo de desenvolvimento de software e deveriam, em princípio, ser ferramentas de engenharia de software projetadas diretamente como serviço. Conforme discutido na seção anterior, pesquisas para o desenvolvimento de tais serviços é uma área com crescente aumento de interesse de pesquisadores (Bartolini *et al.*, 2009; Eler *et al.*, 2009; Ghezzi e Gall, 2008). Os primeiros serviços que vêm sendo desenvolvidos são

[1] Neste capítulo os termos ferramentas e serviços (de engenharia de software) serão usados intercambiavelmente quando não houver dúvida quanto ao seu significado.

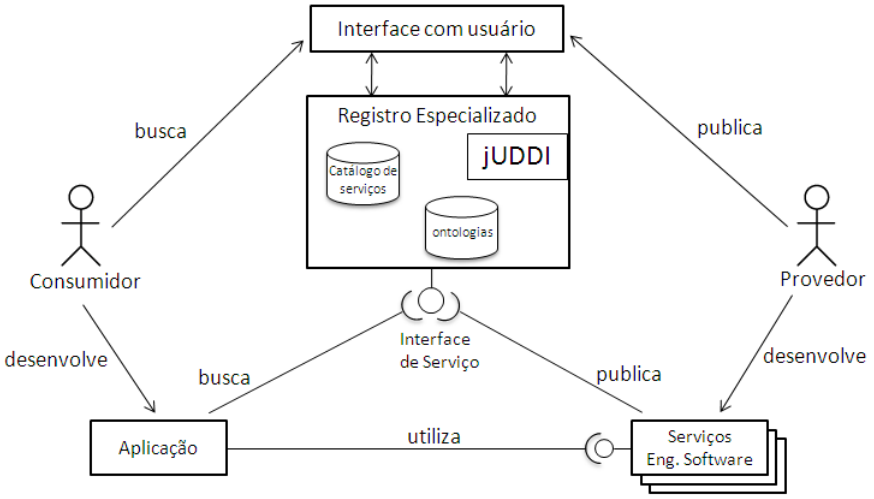

Figura 4.1: SOAES: Uma Arquitetura Baseada em Serviço para o Domínio de ES

principalmente ferramentas tradicionais transformadas em serviços. As propostas de Lee *et al.* (2004) e de Yen *et al.* (2008) são úteis para esta tarefa nas quais os autores propõem técnicas para transformar componentes de software em serviços. Outra possibilidade é encapsular uma ferramenta existente e disponibilizar sua funcionalidade como operações de serviço (Eler *et al.*, 2009; Ghezzi e Gall, 2008).

As aplicações podem ser programas ou processos de desenvolvimento de software (completos ou parciais) que são automatizados em linguagens de *workflow* como BPEL (OASIS, 2007). Elas usam e integram diferentes serviços de engenharia de software ou podem ser simples *drivers* que encaminham um artefato para um único serviço e retornam o resultado para o consumidor.

O registro de serviços possui um papel-chave na arquitetura. Um deles é oferecer uma ontologia com termos de engenharia de software ou outro subdomínio específico para apoiar a busca e armazenamento de serviços. O uso de ontologias para classificar os serviços registrados garante que o registro possa ser adaptado para diversos domínios, a precisão nas buscas realizadas e que padrões possam ser estabelecidos para certos tipos de ferramentas, tais como o nome de operações e suas assinaturas. Além disso, é importante que o registro ofereça acesso aos serviços registrados tanto por meio de uma interface Web quanto por uma interface de serviço para permitir automação nas buscas.

Os atores envolvidos no desenvolvimento e uso de serviços (provedores e consumidores) devem estar de acordo com um conjunto de regras de governança para se beneficiarem da arquitetura (Bertolino e Polini, 2009). Há regras válidas para toda a arquitetura e outras que são específicas para cada serviço publicado. Algumas regras válidas para a arquitetura em geral são:

- O provedor deve obedecer à ontologia implementada no registro de serviços para poder registrar um serviço;
- O provedor deve atender certos padrões requeridos pela arquitetura, como nomes das operações, sequência e tipos dos parâmetros e termos utilizados para classificar um serviço a ser registrado no registro;
- O provedor é responsável por oferecer ferramentas de apoio para apoiar as tarefas dos consumidores, sendo este um requisito opcional.

Por último, para adaptar a arquitetura para um domínio em particular é importante que seja satisfeito o seguinte:

- As ferramentas de engenharia de software disponibilizadas devem ser projetadas e implementadas como serviços ou adaptadas para uso como serviço por meio de invocações em *workflows* ou composição de programas.
- O registro deve ser projetado para facilitar o processo de busca e publicação de serviços no domínio em particular, possivelmente fazendo uso de ontologias específicas do domínio.

4.4 Uma Instanciação de SOAES para o Domínio de Ferramentas de Teste

No contexto do presente trabalho, seguindo a ideia da arquitetura geral, uma arquitetura específica para apoiar a descoberta e publicação de serviços de teste de software deve satisfazer aos seguintes requisitos:

- As ferramentas disponibilizadas devem ser projetadas e implementadas como serviços de teste de software.
- O registro de serviços deve ser projetado fazendo uso de ontologias no domínio de teste.
- O desenvolvedor do serviço de teste atua como provedor de serviços.
- Os desenvolvedores e testadores interessados em realizar o teste de seus programas atuam como consumidores.

Todos os atores devem obedecer a duas regras de governança mais específicas: (1) provedores de serviços de teste devem classificar o serviço de acordo com a ontologia de teste implementada pelo registro; e (2) consumidores devem concordar com regras especificas de governança de cada serviço de teste.

Em relação ao registro de serviços, sua principal função é dar apoio à publicação, busca e classificação de serviços, em particular, àqueles relacionados a ferramentas de teste. Ele tem por objetivo ser um registro de serviços padronizado, no qual um provedor de serviços pode armazenar a especificação de seus serviços, tanto para uso próprio quanto para disponibilizar os serviços externamente. Além disso, ele permite aos consumidores (ou clientes) de serviços encontrarem serviços que melhor atendam suas necessidades de acordo com determinado critério de busca.

É importante destacar que uma dificuldade comum dos sistemas de registro refere-se às buscas realizadas. O problema predominante são as limitações impostas pelo uso de palavras-chave, que geralmente traz resultados que têm pouca ou nenhuma relação com o resultado esperado pelo usuário. No entanto, ao modelar serviços com ontologias, a representação semântica do serviço e seus relacionamentos pode ser explorada de modo que a busca semântica possa ser executada.

4.4.1 Ontologia

Para apoiar a descrição e recuperação semântica de serviços no registro desenvolvido, foi realizada uma adaptação das ontologias de teste de software definidas por Barbosa *et al.* (2006b) e por Bai *et al.* (2008), considerando também diretrizes do modelo de teste U2TP (U2TP, 2007). A ontologia adaptada tem como objetivo incluir conceitos usados para descrever ferramentas ou serviços de teste, conceitos que não estavam presentes na ontologia original, tais como critérios de teste, linguagem de programação ou modelo (MEF, UML, etc.) ao qual o serviço oferece suporte. A ontologia completa possui conceitos que descrevem as técnicas de teste, fases e artefatos usados pelo serviço durante a atividade de teste. Dessa forma, os consumidores podem buscar por serviços que casam com determinado critério, técnica de teste ou linguagem à qual o serviço dá suporte. A estrutura básica da ontologia de teste adaptada é mostrada na Figura 4.2.

A ontologia é dividida em cinco camadas principais: fase de teste, artefato de teste, técnica de teste, critério de teste e linguagem ou modelo de apoio;

1. **Fase de teste**: representa em quais etapas do teste o serviço é utilizado ou as fases às quais ele oferece apoio como, por exemplo, teste de unidade, teste de integração e teste de sistema.

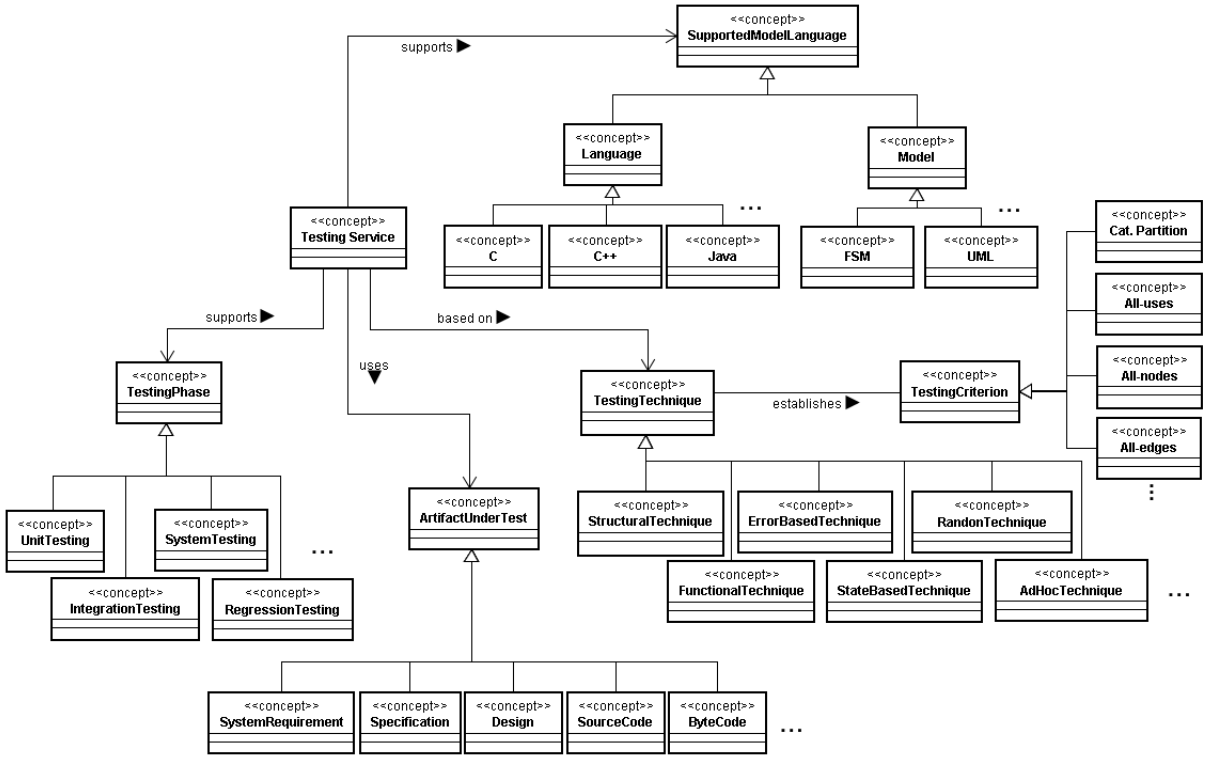

Figura 4.2: Estrutura da ontologia de teste

2. **Artefato de teste**: indica os recursos utilizados ou consumidos pelo serviço durante a atividade de teste. Tais recursos podem variar conforme a técnica ou procedimento de teste utilizado. Exemplos de artefatos são: requisitos, especificação ou código-fonte.

3. **Técnica de teste**: indica a quais técnicas o serviço oferece suporte. Por exemplo, técnica funcional, estrutural, baseada em erros e baseada em estados.

4. **Critério de teste**: estabelecido de acordo com a técnica utilizada. Por exemplo, todos-nós, todas-arestas, todos-usos são instancias de critérios da técnica estrutural.

5. **Linguagem/modelo de apoio**: pode indicar dois conceitos: representa à qual linguagem de programação o serviço de teste dá suporte ou ainda à qual modelo o serviço oferece suporte caso o serviço realize teste baseado em especificação.

Destaca-se que, além dos elementos-base, a ontologia é flexível e permite que os nós-folha possam ser expandidos para acomodar novos tipos de serviços a serem publicados. Por exemplo, caso algum serviço a ser classificado de acordo com a ontologia possua suporte a uma determinada técnica ou critério que não está presente na ontologia, o usuário pode criar um novo critério que satisfaça os requisitos do serviço a ser classificado.

4.4.2 Interfaces de Acesso

Um elemento importante da arquitetura geral refere-se às interfaces de acesso ao registro de serviços. Conforme mencionado anteriormente, é interessante que haja operações de consulta por meio de uma interface com o usuário (interface Web) e também por uma interface de serviço. A interface com o usuário tem como objetivo facilitar a busca e interação dos serviços para os usuários humanos. Já a interface de serviço oferece operações que permitem aos usuários do registro automatizar o processo de descoberta, invocação e busca pelos serviços desejados.

Para se ter uma ideia das possíveis operações de apoio à publicação e consulta aos serviços disponíveis no registro, podem-se projetar genericamente algumas operações de consulta, conforme ilustra a Figura 4.3. Tais operações são descritas sucintamente a seguir:

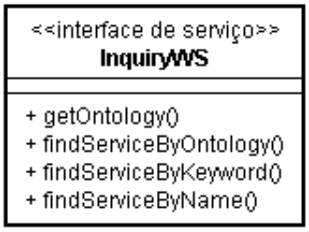

Figura 4.3: Operações de consulta por serviço

- `getOntology()`: operação utilizada para obter a estrutura da ontologia definida durante a publicação de um serviço. Uma vez conhecendo-se a estrutura da ontologia, os usuários podem ter uma visão geral de sua organização e definir por quais conceitos pretendem pesquisar.
- `findServiceByOntology()`: operação para consultar serviços utilizando-se conceitos obtidos da ontologia na operação anterior.
- `findServiceByKeyword()`: operação utilizada para recuperar serviços por meio de palavras-chave.
- `findServiceByName()`: operação utilizada quando se busca por um serviço específico no registro, conhecendo-se o seu nome.

4.5 Considerações Finais

Neste capítulo foi apresentada uma arquitetura genérica baseada em serviços para ferramentas de engenharia de software e uma instanciação para o domínio de teste de software. Para realizar a instanciação foi apresentada uma ontologia de teste de software a ser incorporada no registro para descrever os serviços de teste publicados e também facilitar a recuperação dos serviços. O desenvolvimento dessa abordagem geral é importante, pois contribui para o entendimento e elaboração do registro de ferramentas de teste desenvolvido neste trabalho. O próximo capítulo apresenta os detalhes referentes ao projeto e à implementação desse registro de serviços específico.

CAPÍTULO

5

Projeto e Implementação de um Registro de Ferramentas de Teste

5.1 Considerações Iniciais

Após o estudo geral da arquitetura baseada em serviços para ferramentas de engenharia de software, apresenta-se a seguir o detalhamento do registro de serviços específico para ferramentas de teste desenvolvido neste trabalho. O uso do registro de serviços permite engenheiros de software selecionar e compor serviços que satisfaçam as necessidades de um processo de desenvolvimento particular.

Conforme apresentado na Seção 4.1, é crescente o número de ferramentas de teste desenvolvidas ou disponibilizadas como serviços. Destaca-se que no futuro deverão existir diversos serviços de engenharia de software publicados em registros específicos. Assim torna-se importante o desenvolvimento de um registro de serviços relacionados à atividade de teste de software.

Um problema que ocorre nos serviços de registro refere-se às buscas realizadas, que por serem basicamente sintáticas podem trazer resultados pouco relacionados aos interesses do usuário. Faz-se necessária a definição de um mecanismo que ofereça um melhor apoio às buscas, de modo a facilitar sua execução, além de oferecer resultados mais significativos de acordo com os interesses do usuário.

Este capítulo está organizado da seguinte forma: na Seção 5.2 são apresentados os aspectos de modelagem e a arquitetura do registro. Na Seção 5.3 são apresentados os detalhes relativos à sua implementação. Por último, na Seção 5.4 são apresentados os aspectos operacionais do serviço de registro e na Seção 5.5 as considerações finais do capítulo.

5.2 Modelagem

Segundo Josuttis (2007), as seguintes informações devem ser levadas em consideração para projetar um registro de serviços: (1) *quantidade de informação*, que significa definir se o registro armazena todas as informações de serviços ou apenas detalhes técnicos relevantes; (2) *interface com o usuário*, refere-se ao registro possuir uma interface gráfica ou apenas opções para inserção, atualização e recuperação/busca dos serviços; e (3) *arquitetura do registro*, refere-se ao registro ser centralizado ou uma colaboração entre registros distribuídos.

O registro de ferramentas de teste desenvolvido caracteriza-se pelo seguinte: em relação à quantidade de informação, ele armazena descrições textuais do serviço, detalhes técnicos e informações de classificação de acordo com a ontologia utilizada; o registro possui uma interface com usuário para facilitar o acesso por humanos; e sua arquitetura é centralizada.

Para modelagem do registro de serviços, inicialmente foi realizada a fase de especificação de requisitos, na qual foram identificados e definidos os requisitos funcionais. A partir da definição dos requisitos foram elaborados o diagrama de casos de uso e a especificação dos casos de uso básicos do serviço de registro. A Figura 5.1 mostra o diagrama de casos de uso do sistema.

No diagrama são mostradas as possíveis interações sob a perspectiva do provedor de serviços, do administrador e do cliente do serviço de registro. Note que somente o `Provedor` pode realizar a publicação de um serviço e essa atividade inclui classificar o serviço de acordo com a ontologia implementada no registro. Já as outras opções de busca e gerência dos serviços podem ser realizadas pelos demais usuários.

A partir da definição dos casos de uso, foram elaborados o diagrama de sequência e o de classes para representar o sistema a partir de diferentes visões. A Figura 5.2 ilustra o diagrama de classes desenvolvido, no qual são representadas apenas as classes de domínio do sistema (entidades) e a forma como estão relacionadas. As classes `Provider` e `ProviderDetails` descrevem informações sobre os provedores de serviços. A Classe `Provider` possui um relacionamento com a classe `Service`, indicando que o provedor é quem realiza a publicação de um serviço no registro.

As classes `Service` e `ServiceDetails` representam o serviço de teste e seus detalhes. A classe `Service` possui relacionamento com as classes `Operation`, `Technique`, `Phase`,

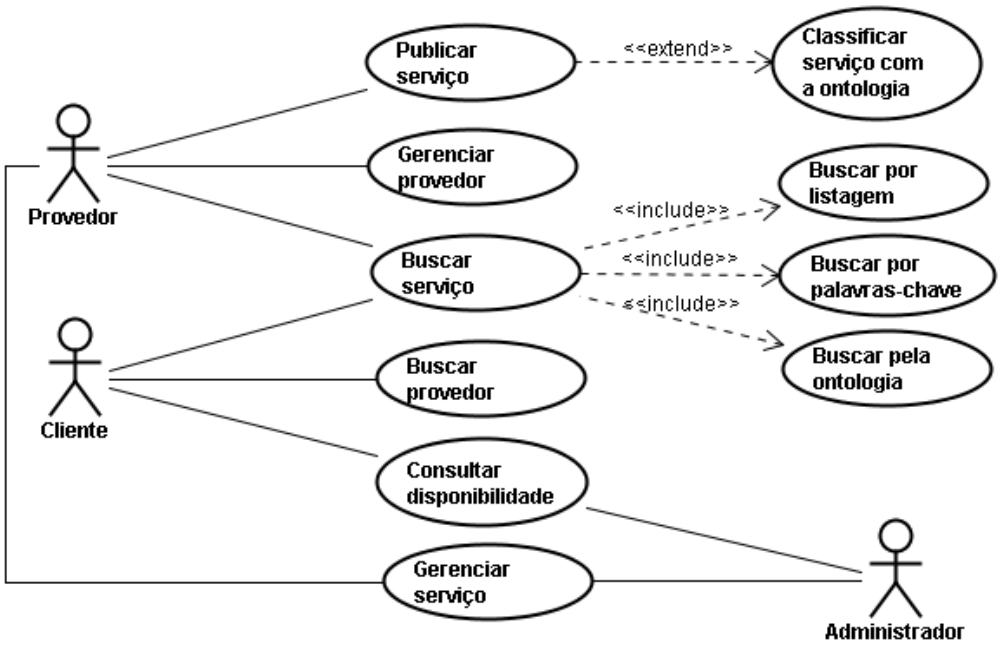

Figura 5.1: Extrato do diagrama de casos de uso do registro de serviços

`Keyword` e `SML` (*Supported Model or Language*). Tal relacionamento deve-se a que, durante a publicação de um serviço, são definidas as operações, técnicas, fases, critérios, palavras-chave e a qual linguagem o serviço dá suporte. As classes `Technique`, `Criterion`, `Phase`, `Artifact` e `SML` representam os conceitos da ontologia de teste que são armazenados juntamente com o serviço que é publicado no registro.

Com a definição dos requisitos e a modelagem do registro, a próxima etapa realizada foi a definição da arquitetura do sistema de registro, que é constituída de quatro camadas, conforme ilustrada na Figura 5.3.

1. **Primeira camada**: representa a interface com o usuário e tem por objetivo oferecer facilidades de busca e publicação de serviços por humanos. Ela é disponibilizada por meio de uma interface Web.

2. **Segunda camada**: representa o núcleo do registro de serviços. Ela define os módulos funcionais e as estruturas de dados para manipulação das classes do sistema. As informações referentes aos detalhes de provedores, serviços e ontologia estão acessíveis por meio de uma interface de serviço de modo a permitir a consulta ao registro por meio de programas ou em uma composição de serviços.

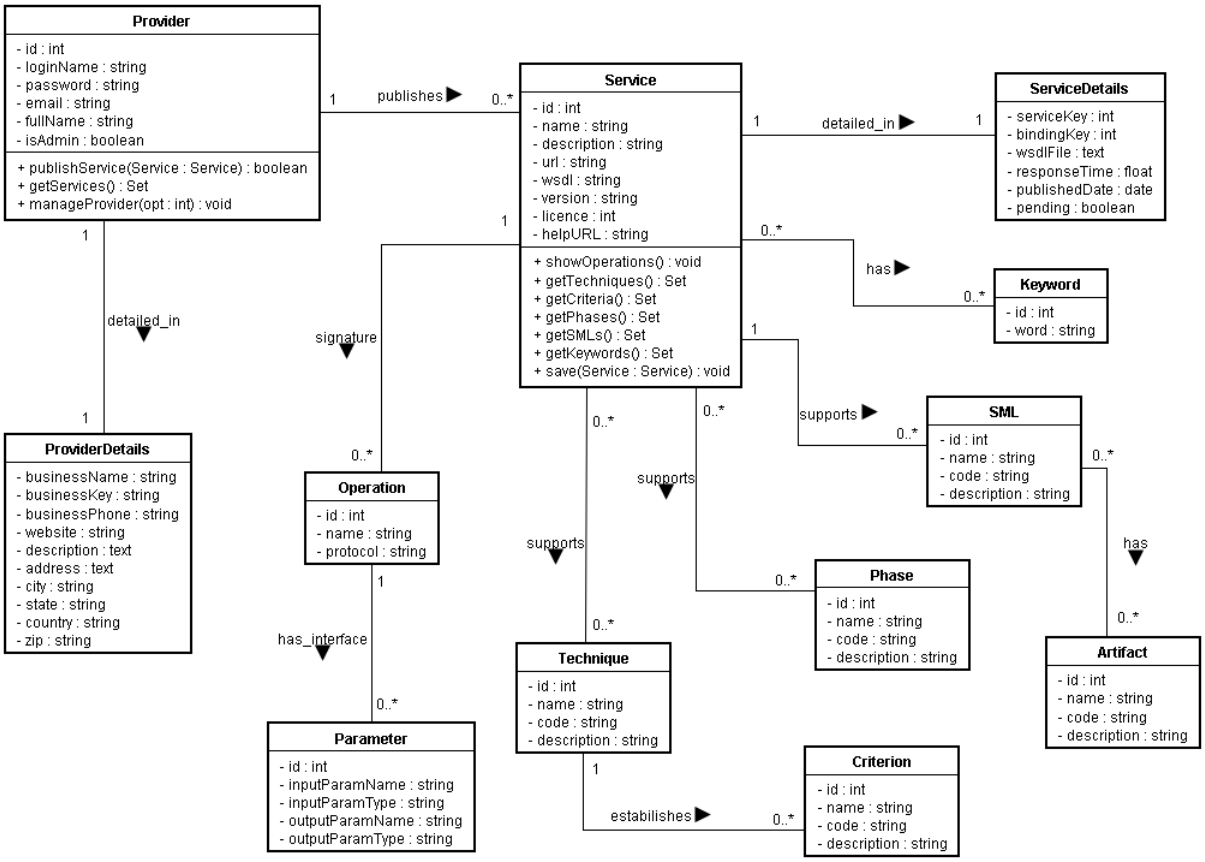

Figura 5.2: Diagrama de classes do registro de serviços

3. **Terceira camada**: é utilizada para processar informações conforme o padrão UDDI (OASIS, 2005). Ela utiliza o *framework* jUDDI (Apache jUDDI, 2004) que oferece APIs (*Application Programming Interface*) de consulta, publicação e autenticação e um mecanismo para armazenar os detalhes técnicos de provedores e serviços de acordo com este padrão.

4. **Quarta camada**: é responsável pela busca e persistência dos dados utilizando o Hibernate (2005) – um *framework* para mapeamento objeto-relacional (*ORM - Object-Relational Mapping*). Dessa forma, a comunicação entre a camada de aplicação e o banco de dados é realizada de maneira independente do banco escolhido. Neste caso específico, foi utilizado o banco PostgreSQL (1996).

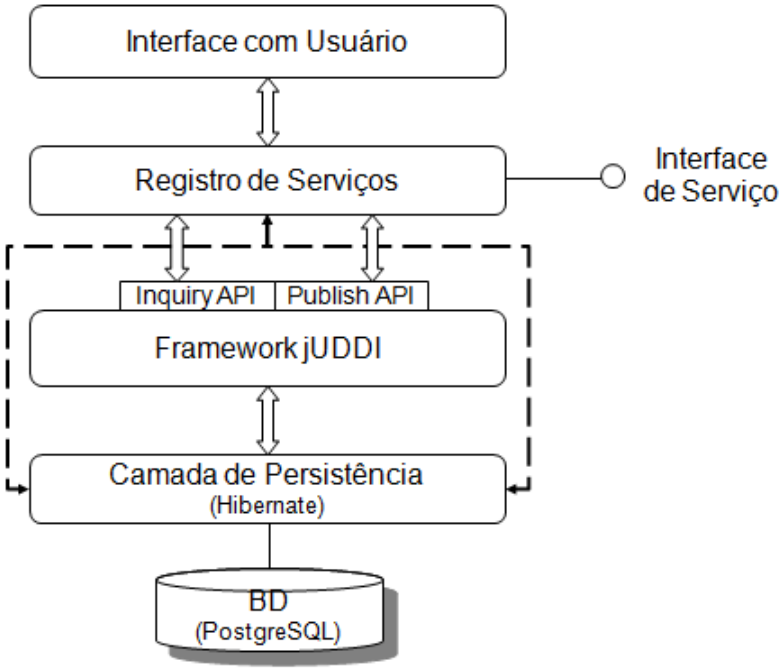

Figura 5.3: Arquitetura do registro de serviços de ferramentas de teste

5.3 Implementação

O registro de ferramentas de teste foi modelado seguindo a notação UML na ferramenta Jude (Jude, 2006). Uma vez concluídos os requisitos, o modelo de análise e a arquitetura, foi iniciada a definição da infraestrutura de implementação e de apoio ao funcionamento do sistema de registro.

A implementação do registro de serviços foi feita usando a linguagem de programação orientada a objetos Java no ambiente Eclipse (2004), seguindo diretrizes do modelo JavaBeans. Como camada de apresentação foi utilizado o *framework* Ajax ZK (Potix-Corporation, 2005), que oferece um mecanismo de geração de páginas Web e uma maneira de desacoplar a camada do cliente da lógica de negócios.

Considerando que o registro foi desenvolvido como uma aplicação Web, o servidor de aplicações adotado foi o Tomcat, que é um Web *container* de código aberto (Apache, 2004). O Sistema de Gerência de Banco de Dados (SGBD) adotado foi o PostgreSQL (1996). No entanto, é importante observar que a aplicação pode ser executada em qualquer outra base de dados relacional, pois realiza o mapeamento objeto-relacional com o *framework* Hibernate (2005).

É importante ressaltar que a maioria das tecnologias citadas anteriormente enquadra-se na filosofia de software livre ou faz uso de recursos subjacentes de software livre para disponibilizar suas funcionalidades.

Para permitir uma maior separação entre a camada de apresentação e a lógica de negócios no sistema de registro, foi utilizado o padrão MVC (*Model-View-Controller*) (Buschmann *et al.*, 1996). O padrão MVC vem sendo utilizado principalmente em aplicações Web por promover a separação das funcionalidades nas seguintes camadas: (1) lógica de apresentação, (2) lógica de negócio e (3) lógica de acesso a dados. Dessa forma, o sistema torna-se mais flexível e permite que as partes possam ser alteradas sem prejuízo para outros subsistemas. Um extrato da organização da implementação do sistema de registro pode ser visto na Figura 5.4. O pacote `service` contém classes que são responsáveis por disponibilizar operações de consulta ao registro por meio de uma interface de serviço.

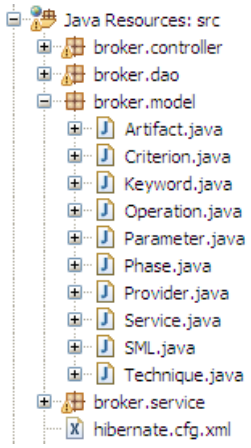

Figura 5.4: Organização da implementação do registro de serviços

Conforme discutido anteriormente, optou-se por utilizar o *framework* jUDDI (Apache jUDDI, 2004) em relação ao padrão adotado para construção de registros de serviços por ser uma imple-

mentação de código aberto em Java do padrão UDDI. O jUDDI oferece um conjunto de APIs utilizadas para interação com o registro de serviços. Elas são agrupadas por funcionalidades, com operações para publicação, consulta e controle de acesso (segurança) dos serviços, conforme é exibido na Figura 5.5. O mecanismo de acesso padrão do jUDDI é realizado por meio de um console sem a presença de uma interface gráfica. Nesse console são oferecidas operações de consulta no qual o usuário manipula mensagens SOAP diretamente.

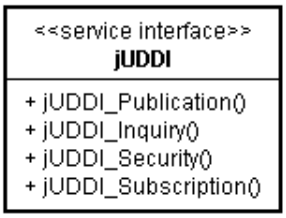

Figura 5.5: APIs do *framework* jUDDI

O UDDI é um dos padrões mais utilizados atualmente para construção de registros de serviços, ele resolve o problema da descrição e descoberta universal. Entretanto, uma desvantagem refere-se a não oferecer apoio à descrição semântica dos serviços (Atkinson *et al.*, 2007). Como a proposta deste trabalho é oferecer um registro de serviços com facilidades de busca semanticamente mais ricas com base em uma ontologia de teste, foram elaboradas operações adicionais de consulta ao registro, além daquelas existentes no jUDDI, levando em consideração a ontologia de teste discutida na Seção 4.4 e a semântica de seus relacionamentos. Estas operações são discutidas em detalhes na Seção 5.4.2 a seguir.

Durante o desenvolvimento do registro de serviços foram realizados testes funcionais manuais para verificação e validação das funcionalidades. Foram elaborados testes unitários apenas para classes de maior grau importância em termos de negócio, nesse caso usando o *framework* JUnit (Beck e Gamma, 2005).

Para dar uma ideia do esforço de implementação do sistema de registro, na Tabela 5.1 são mostradas algumas métricas gerais obtidas usando o *plugin* Metrics (2009). As métricas para contagem do número de linhas de código (LOC - *Lines of Code*) levam em consideração o código em Java do sistema de registro, as classes que foram reusadas do *framework* jUDDI e o código de geração das páginas Web feitos com o *framework* ZK. Não são considerados comentários, linhas em branco e contagem das linhas referentes às classes de teste.

Tabela 5.1: Métricas do registro de serviços de teste

Métrica	Total
Número de Páginas Web	23
Número de Classes	33
Número de Atributos	158
Número de Métodos	252
Número de Linhas de Código	4391

5.4 Aspectos Operacionais

O registro desenvolvido permite a busca e publicação de serviços, podendo-se buscar serviços com o uso de palavras-chave, por ontologia ou por uma interface de serviço. A publicação dos serviços pode ser feita manualmente por um humano ou automaticamente por um programa. O sistema permite a navegação por meio de ontologia para auxiliar a busca dos serviços que atendam aos critérios selecionados. Destaca-se que o protocolo de descoberta dos serviços é do tipo página amarela, no qual o cliente descobre o serviço que necessita no registro, recebe as informações pertinentes e depois se comunica diretamente com o prestador do serviço.

5.4.1 Interface Web

A seguir descreve-se a interface Web e os aspectos operacionais do registro de serviços desenvolvido. Conforme discutido anteriormente, a disponibilização de uma interface Web tem por objetivo facilitar o processo de descoberta de serviços por humanos. Na Figura 5.6 é mostrada a página inicial do sistema de registro e publicação de serviços de teste.

Neste sistema os usuários podem consultar, atualizar e gerenciar os serviços catalogados e inclusive expandir a ontologia proposta no momento da publicação de um novo serviço, caso este possua um novo conceito que não esteja presente na ontologia original. A página inicial exibe a ontologia de teste em formato de árvore e uma área de autenticação. O sistema de registro possui um menu de navegação à esquerda com operações de publicação e de busca de serviços. Logo abaixo das operações é exibida a quantidade de serviços atualmente publicados.

As operações de busca por meio da interface Web são apresentadas a seguir (mais detalhes são apresentados no próximo capítulo na Seção 6.3):

- **Ordered search**: esta operação exibe uma listagem contendo todos os serviços atualmente registrados em ordem alfabética, são mostrados também alguns detalhes do serviço como nome, descrição e licença de uso, dentre outros.

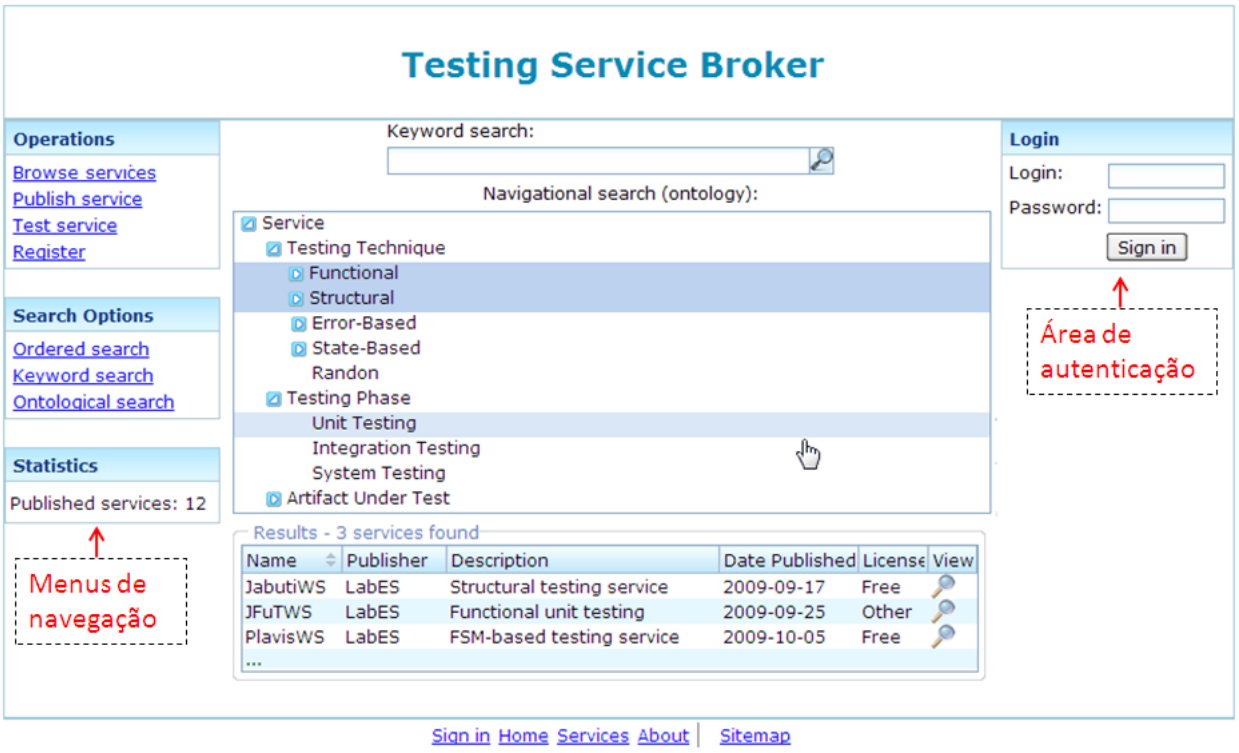

Figura 5.6: Página principal do registro de serviços de ferramentas de teste

- **Keyword search**: esta opção de busca usa uma lista de palavras-chave e procura, fazendo uso do operador OR, no nome, descrição e palavras-chave dos serviços registrados.
- **Ontological search**: esta opção de busca exibe primeiramente a estrutura da ontologia em formato de árvore. Em seguida, permite ao usuário do registro buscar por serviços que casam com os conceitos da ontologia selecionados. Os serviços retornados devem satisfazer todos os conceitos informados.

Para realizar a publicação de um novo serviço no registro, o usuário deve estar cadastrado no sistema. Uma vez cadastrado no sistema, o usuário torna-se provedor e pode incluir os serviços que possui. Para realizar o cadastro há algumas informações obrigatórias e outras opcionais. As obrigatórias incluem dados de login, email, nome do provedor e senha e as opcionais são uma descrição do provedor, website e endereço. Na Figura 5.7 é mostrado a página de cadastro no sistema de registro.

Quando o usuário acessa o sistema é exibida a área restrita que possui operações de publicação, busca e gerenciamento de seus dados. Além disso, são exibidos os serviços atualmente publicados pelo provedor atual com opções de visualizar, editar ou remover os serviços, conforme é mostrado na Figura 5.8. No próximo capítulo serão detalhados exemplos de publicação e uso dos serviços registrados.

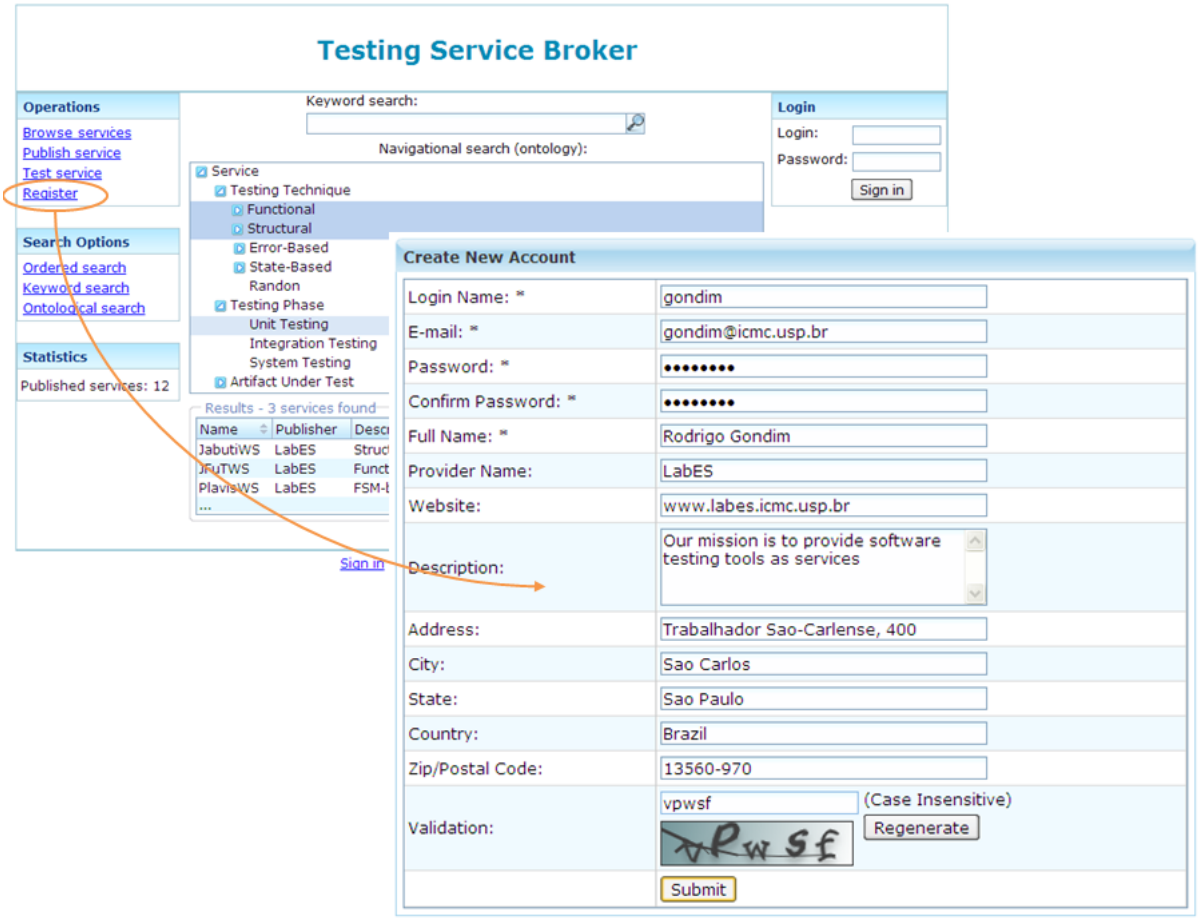

Figura 5.7: Página para inserção de provedor no sistema de registro

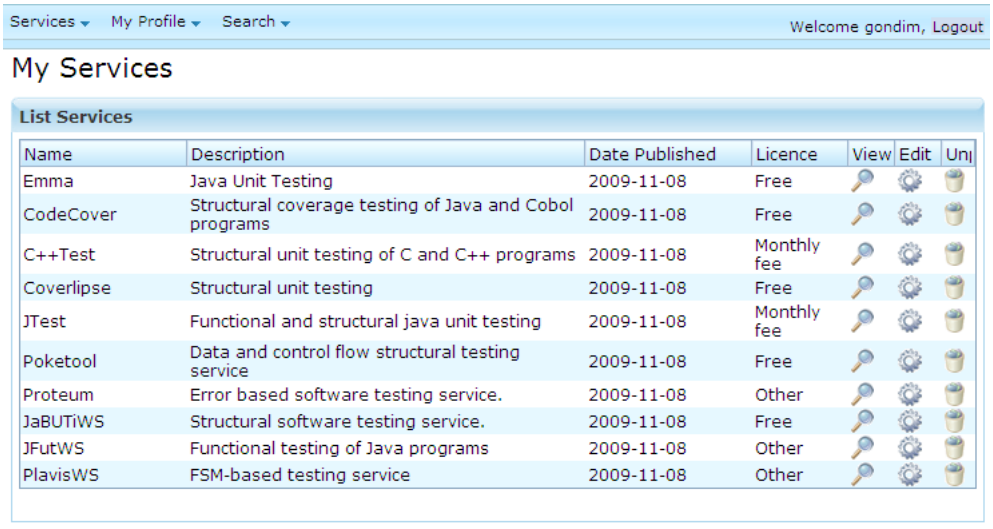

Figura 5.8: Área restrita do registro de serviços de ferramentas de teste

5.4.2 Interface de Serviço

Conforme discutido anteriormente, as operações do serviço de registro permitem aos usuários automatizar o processo de descoberta, invocação e busca pelos serviços desejados. Para oferecer apoio às buscas por serviços de teste por meio da interface de serviço, as operações ilustradas na Figura 5.9 estão disponíveis. Destaca-se que podem ser utilizadas no registro tanto as operações já disponíveis no jUDDI quanto as novas operações de busca por ontologia desenvolvidas.

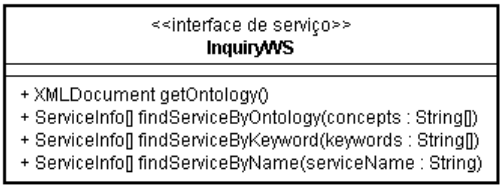

Figura 5.9: Operações de consulta ao registro por meio da interface de serviço

- `public XMLDocument getOntology()`
  A operação `getOntology()` é utilizada para obter a estrutura da ontologia definida durante a publicação de um serviço. A operação retorna um `XMLDocument` (Dom4j, 2010) que representa a ontologia em formato XML contendo todos os conceitos definidos pelos usuários até o momento da consulta. Alguns exemplos de conceito são: instâncias de técnicas de teste, fases, artefatos e linguagem de suporte que os serviços registrados oferecem. Conhecendo a estrutura da ontologia, os usuários podem ter uma visão geral de sua organização e definir por quais conceitos pretendem pesquisar.

- `public ServiceInfo[] findServiceByOntology(String[] concepts)`
  A operação `findServiceByOntology()` é utilizada para consultar serviços com base na ontologia. Ela recebe como entrada uma lista de instâncias dos conceitos e busca por serviços que casam com tais conceitos. A operação retorna uma lista de elementos do tipo `ServiceInfo`, como ilustrado na Figura 5.10. A estrutura `ServiceInfo` contém detalhes do serviço como nome, descrição, endereço de acesso, endereço WSDL, URL de documentação, nome e website do provedor de serviços. O usuário pode utilizar tais informações para interagir com o serviço desejado.

- `public ServiceInfo[] findServiceByKeyword(String[] keywords)`
  A operação `findServiceByKeyword()` é utilizada para recuperar serviços por meio de palavras-chave. Ela recebe como entrada uma lista de palavras-chave e executa a busca por nome, descrição e palavras-chave dos serviços. A operação também retorna uma lista de elementos `ServiceInfo`.

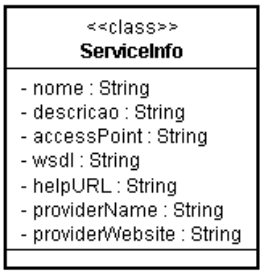

Figura 5.10: Tipo retornado nas operações de busca contento informações de serviço

- `public ServiceInfo[] findServiceByName(String serviceName)`
  A operação `findServiceByName()` é utilizada quando se conhece ou se busca por um serviço específico no registro, a partir do seu nome.

5.5 Considerações Finais

Este capítulo apresentou os detalhes de projeto e implementação do registro de serviços desenvolvido neste trabalho. Inicialmente foi realizada a etapa de modelagem, com a definição dos casos de uso e diagrama de classes. Em seguida foi discutido os detalhes de implementação e a arquitetura do registro de serviços. Considerando esses artefatos, foram então definidas as tecnologias de implementação do serviço de registro.

Em relação ao registro de serviços foram descritas suas funcionalidades para publicação e buscas sob dois aspectos: do ponto de vista da interface Web e também da interface de serviço. No próximo capítulo são apresentados exemplos de uso do serviço de registro e também aspectos de validação por meio de um exemplo de interação do registro com o serviço de teste JaBUTiWS.

CAPÍTULO

6

Uso e Validação do Registro e do Serviço JaBUTiWS

6.1 Considerações Iniciais

O registro de serviços oferece um ponto central de armazenamento e busca de serviços que implementam funcionalidades de teste de software. Os serviços publicados podem ser originalmente desenvolvidos como serviços ou ferramentas de teste *standalones* convertidas em serviços. Neste capítulo são apresentados os serviços de teste atualmente publicados no registro e também são apresentados exemplos de uso do registro de serviço. O capítulo começa apresentando os serviços atualmente publicados na Seção 6.2. Na Seção 6.3 são apresentados exemplos de publicação de um serviço e de busca por meio da interface Web. Também é descrito um exemplo de busca por meio da interface de serviço e a interação com o serviço JaBUTiWS. Por fim, na Seção 6.4 são feitas as considerações finais do capítulo.

6.2 Serviços Publicados

Os serviços a serem publicados no registro desenvolvido são aqueles que implementam funcionalidades de teste de software. Eles podem ser ferramentas de teste convertidas e disponi-

bilizadas como serviços ou implementações feitas diretamente como serviços. Atualmente existem três serviços de teste publicados no registro e todos são ferramentas de teste disponibilizadas como serviços a partir de ferramentas previamente desenvolvidas pelo grupo de engenharia de software do ICMC-USP: JaBUTiWS (Eler *et al.*, 2009), PlavisWS (Dusse, 2009) e JFutWS (Rocha *et al.*, 2005).

O JaBUTiWS é um serviço para o teste estrutural de programas Java, ele possui um conjunto de operações para executar instrumentação de *bytecode* e análise de cobertura com base nos seguintes critérios: todos-nós, todas-arestas, todos-usos e todos-p-usos. Um cliente do JaBUTiWS deve seguir uma sequência de operações para sua execução. Primeiramente, o testador cria um projeto e envia o programa para ser instrumentado. Em seguida, o testador obtém o programa instrumentado e executa-o com base em seus casos de teste. Um arquivo de rastreamento da execução é gerado e enviado ao JaBUTiWS, que utiliza o arquivo para analisar quais requisitos foram ou não cobertos para gerar o relatório de cobertura de acordo com os critérios implementados.

O JaBUTiWS também apoia o teste de serviços (Eler *et al.*, 2010). Desenvolvedores de serviços podem utilizar o JaBUTiWS para instrumentar seus serviços e gerar serviços testáveis. A instrumentação é feita para gerar um arquivo de rastreamento com informações da execução do serviço. O JaBUTiWS insere três operações no serviço Web: `startTrace` e `stopTrace` para definir os limites de uma sessão de teste; e `getCoverage` para obter a análise de cobertura feita por meio do arquivo de trace durante a sessão de teste. Durante a etapa de instrumentação, os desenvolvedores podem decidir em qual nível de detalhe realizar análise de cobertura (por serviço, operações da interface, classe ou métodos).

O PlavisWS (Dusse, 2009) é um serviço de teste baseado na ferramenta PlavisFSM (Simão *et al.*, 2005). Ele foi implementado utilizando abordagem SOA e o núcleo do serviço é escrito em linguagem C. Ele tem como objetivo oferecer suporte a experimentos de teste baseados em MEF para comparação de métodos e critérios de teste. Os testadores podem definir uma MEF, gerar mutantes, gerar diagramas MEF com transições, derivar casos de teste e executar análise de mutação utilizando o serviço.

O JFuTWS é um serviço de teste baseado na ferramenta JFuT (Rocha *et al.*, 2005). Ele encontra-se em em fase de desenvolvimento e tem por objetivo apoiar o teste funcional de programas Java. O serviço permite aos testadores instrumentar o código fonte e obter análise de cobertura baseado nos seguintes critérios funcionais: particionamento em classes de equivalência e análise do valor limite. O serviço também permite realizar avaliação de pré e pós-condições.

Ressalta-se que, além dos serviços de teste apresentados, diversas outras ferramentas de engenharia de software ou teste de software podem ser convertidas em serviços e disponibilizadas

no registro. A título de ilustração, potenciais ferramentas para serem publicadas no registro são: Cobertura, ferramenta utilizada para realizar análise de cobertura em programas Java por classes, pacotes e todo o projeto (Cobertura, 2010) e XPlanner, uma ferramenta de gestão de projetos que auxilia no planejamento, interação e geração de métricas no contexto de métodos ágeis (XPlanner, 2006). Esta última ferramenta refere-se a um domínio mais amplo e poderia ser integrada no registro com o uso de ontologias de outros domínios, no caso de métodos ágeis.

Na Tabela 6.1 é apresentado um quadro resumo com a classificação por ontologia dos serviços específicos de teste de software que foram publicados no registro. Com o próposito de teste e para cobrir a amplitude da ontologia também são apresentadas outras ferramentas *standalones* que foram publicadas no registro como sendo consideradas serviços de teste.

Tabela 6.1: Classificação das serviços ou ferramentas de teste publicados no registro

Serviço	Técnica	Critério	Fase	Artefato	SML
Emma	structural	all-nodes; all-edges;	unit	*bytecode*	java
CodeCover	structural	all-nodes; all-edges;	unit	source-code	java; cobol
C++Test	structural; adhoc	all-nodes; all-edges; adhoc	unit; regression	source-code	c; c++
Coverlipse	structural	all-nodes; all-edges; all-uses	unit; integra-tion	source-code	java
JTest	structural; functional	all-nodes; boundary value analysis	unit; integration; regression	source-code	java
Poke-Tool	structural;	all-nodes; all-edges; all-uses;	unit; integra-tion;	source-code	c; fortran
Proteum	error-based	mutation analysis	unit; integra-tion;	source-code	c
JaBUTiWS	structural	all-nodes; all-edges; all-uses	unit	*bytecode*	java
JFutWS	functional	equivalence partition; bondary-value analysis	unit; system	source-code	java
PlavisWS	state-based; error-based	all-states; all-transitons; mutation analysis	unit	specification	FSM (finite state machine)

6.3 Exemplo de Uso do Registro

Nesta seção são detalhados exemplos de como o serviço de registro pode ser utilizado por um humano para buscas por meio da interface Web e também é descrito o seu uso por meio da interface de serviço. Neste caso é realizada uma interação do registro com o serviço JaBUTiWS.

6.3.1 Publicação de um Serviço

A publicação de um serviço de teste é realizada em duas etapas. Ao selecionar a operação de publicação de um serviço, exibe-se a tela principal de publicação na qual o usuário pode inserir informações básicas do serviço e, em seguida, informações referentes à classificação ontológica, conforme ilustrado na Figura 6.1.

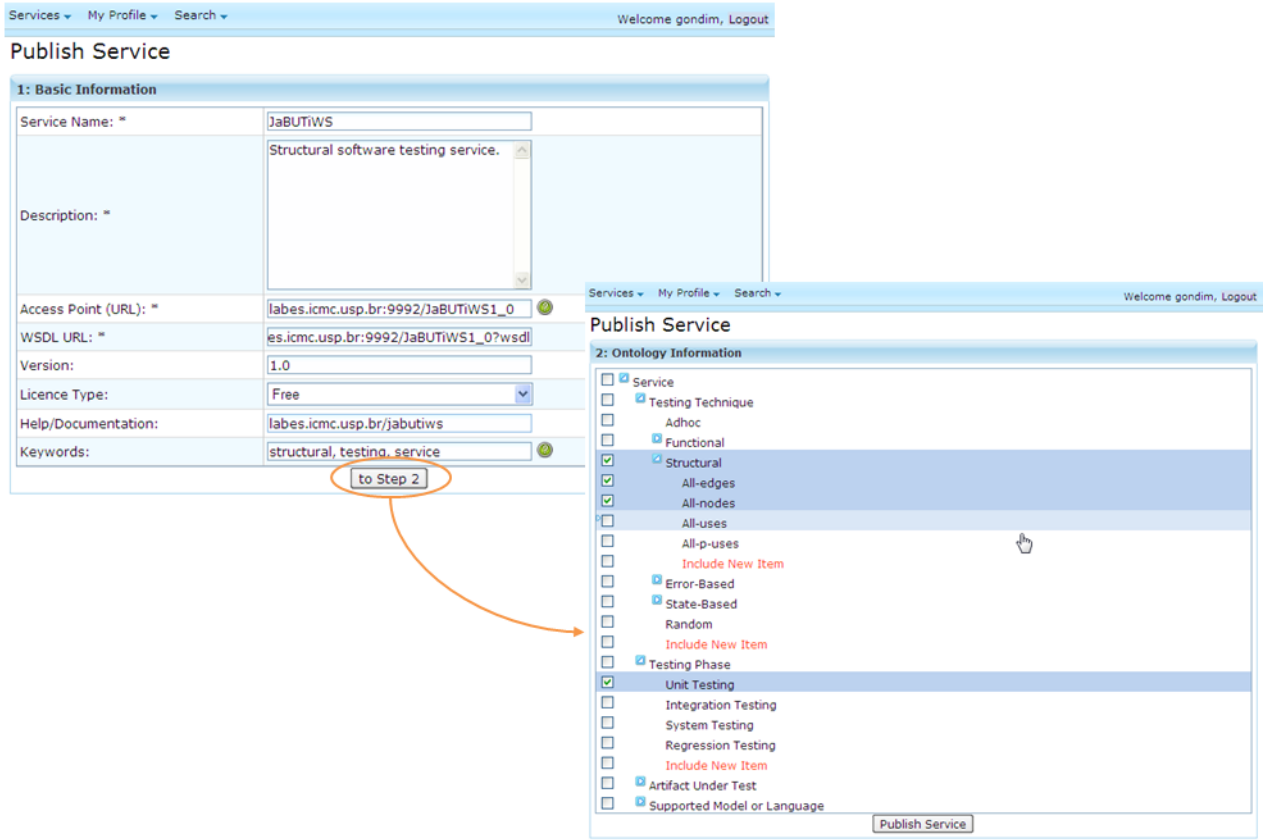

Figura 6.1: Interface do registro para publicação de um novo serviço de teste

As informações básicas solicitadas incluem nome, descrição, endereço do WSDL, versão, licença e palavras-chave do serviço. Em seguida, a classificação ontológica é realizada de acordo com os conceitos da taxonomia de teste, na qual são informadas as fases, técnicas,

critérios e artefatos de teste e a linguagem que o serviço a ser publicado apoia. Neste exemplo, o serviço JaBUTiWS está sendo publicado e o seu provedor seleciona os elementos da ontologia de acordo com os requisitos estabelecidos pelo serviço (conforme apresentado na Tabela 6.1).

Conforme discutido anteriormente, a ontologia de teste é flexível. Assim, durante a publicação, os usuários podem atualizar a ontologia com a inclusão de novos conceitos para cada um dos itens informados. Por exemplo, com a inclusão de um novo artefato de teste ou linguagem que o serviço apoia (Figura 6.2). Por fim, uma vez concluída a classificação ontológica, o serviço pode ser publicado.

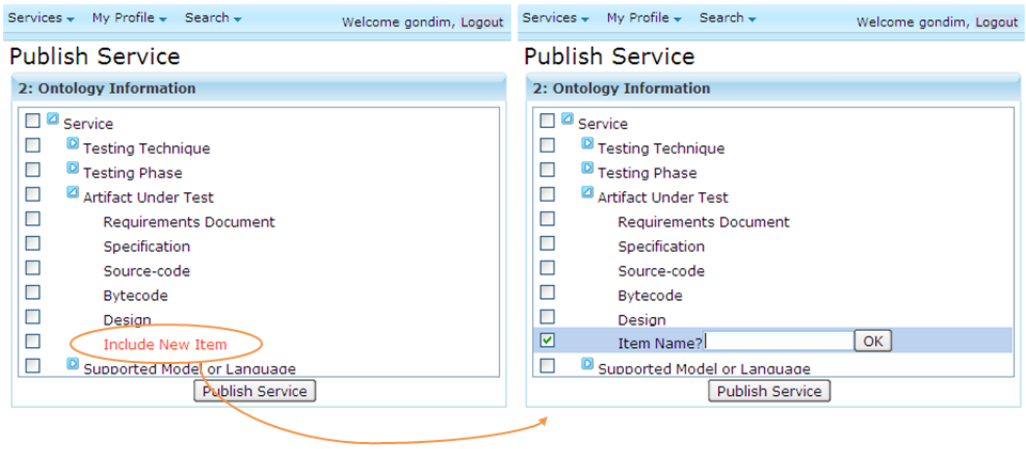

Figura 6.2: Interface do registro para inclusão de um novo conceito na ontologia de teste

Destaca-se que, na versão atual do registro, a etapa de classificação ontológica não verifica os conceitos informados. Entretanto, esta etapa poderia passar por um processo de autorização, no qual os conceitos informados são avaliados por um especialista do domínio ou um administrador, então após a aprovação poder-se-ia permitir seu uso.

6.3.2 Busca pela Interface Web

Conforme apresentado na seção anterior, suponha que o desenvolvedor do serviço JaBUTiWS (provedor) publicou o serviço no registro e classificou-o de acordo com a ontologia de teste. Um humano (testador) quer agora realizar uma busca por um serviço de teste que satisfaça os seguintes critérios: o serviço deve oferecer suporte a técnica estrutural para programas Java e também apoiar a aplicação dos critérios: todos-nós, todas-arestas e todos-usos.

O testador acessa primeiramente a interface Web do registro para buscar por serviços de interesse que atendam seus requisitos. Em seguida, caso for encontrado algum serviço com os critérios selecionados, o testador utiliza o serviço de teste obedecendo às regras de governança específicas do serviço de teste.

A Figura 6.3 ilustra a interação do testador com o mecanismo de busca ontológica do registro de serviços. O testador seleciona os elementos da ontologia de acordo com os requisitos da atividade de teste atual. Neste caso, foram selecionados os seguintes conceitos: teste estrutural, todos-nós, todas-arestas e todos-usos. Uma vez selecionados os conceitos da ontologia o registro realiza uma busca por serviços que casam com os critérios selecionados. Destaca-se que, neste tipo de busca, é utilizado o operador AND e os serviços retornados devem satisfazer todos os conceitos selecionados. Por fim, o registro exibe uma listagem com os serviços retornados.

A lista de resultado exibe apenas um resumo dos serviços retornados. O testador pode visualizar mais detalhes usando a opção lupa (Figura 6.3). Neste caso, são exibidos os detalhes do serviço de teste encontrado, que inclui a URL do WSDL, palavras-chave e uma lista das operações oferecidas pelo serviço, e também os detalhes referentes à classificação ontológica do serviço. Neste exemplo, o serviço de teste que casou com os critérios informados foi o JaBUTiWS.

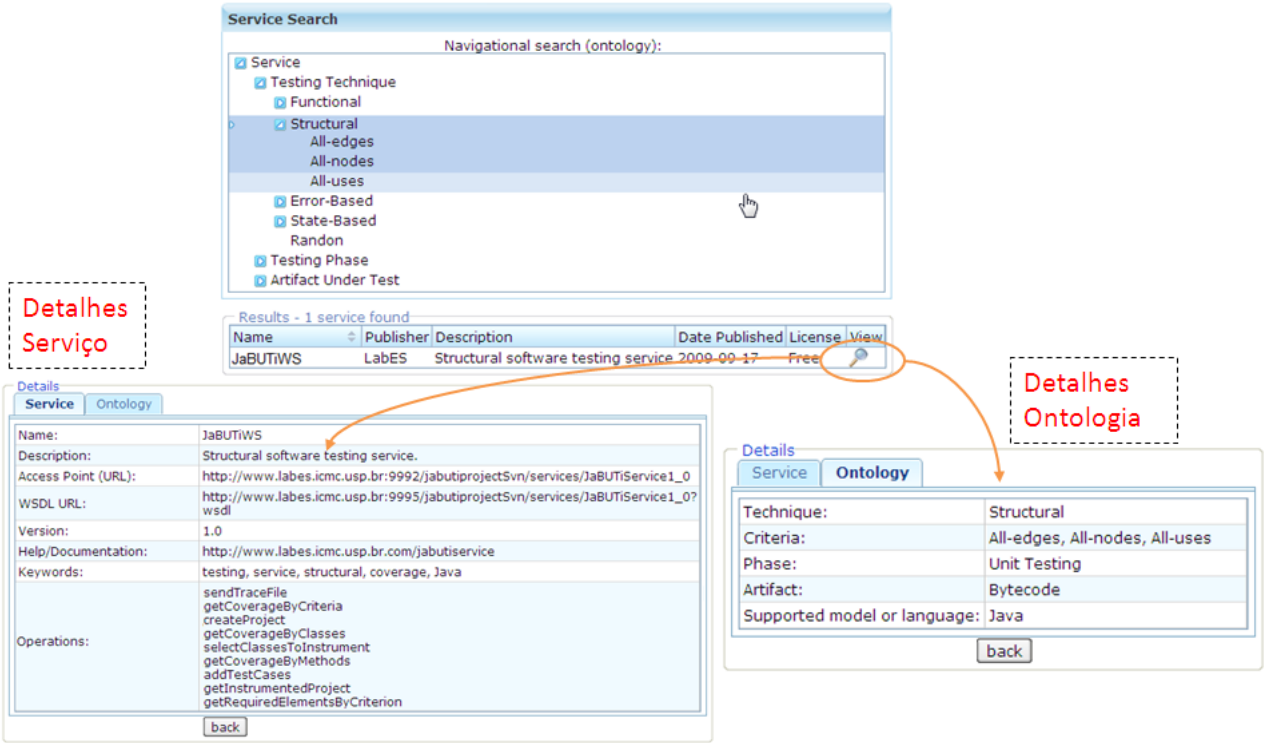

Figura 6.3: Interface com o usuário para o mecanismo de busca por ontologia e detalhes do serviço JaBUTiWS retornado

Outra possibilidade de busca é por meio de palavras-chave. Nela o usuário informa uma lista de palavras que são utilizadas para buscar no nome e na descrição dos serviços registrados. Neste tipo de busca utiliza-se o operador OR para casamento entre os termos.

A título de ilustração considere o seguinte exemplo: recuperar um serviço que atenda as seguintes palavras-chave: *functional*, *unit*, *test*. Neste caso seriam retornados todos os serviços que possuem tais palavras em sua descrição, nome ou lista de palavras-chave, conforme ilustrado na Figura 6.4.

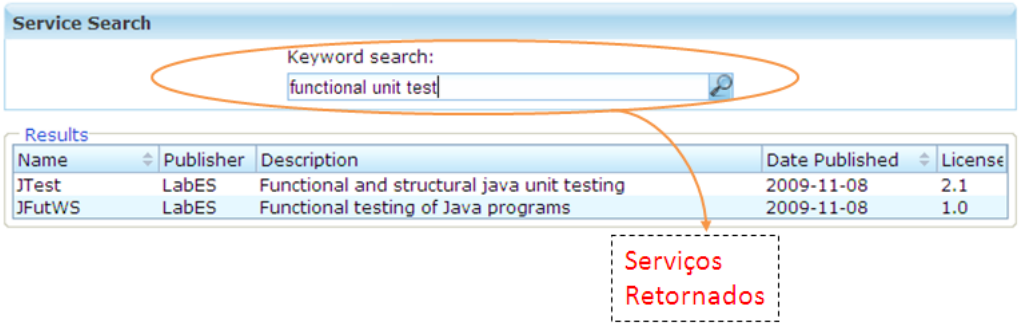

Figura 6.4: Interface de busca por palavras-chave e detalhes dos serviços encontrados

6.3.3 Busca pela Interface de Serviço

Para ilustrar um exemplo de uso do registro por meio da interface de serviço, suponha que um testador está interessado em testar um programa e obter um relatório de cobertura obtido. Um possível cenário de uso para busca é mostrado na Figura 6.5 que exibe um processo genérico simplificado (*workflow*) de interação entre o registro, o testador e o serviço de teste.

Supondo que o testador queira realizar uma busca por ontologia ele primeiramente cria um programa cliente para interagir com as operações da interface de serviço. Ele invoca a operação `getOntology` para conhecer a estrutura da ontologia implementada pelo registro e uma vez conhecendo-a o testador poderá buscar por serviços de acordo com critérios que melhor atendam seus requisitos.

O próximo passo é invocar a operação `findServiceByOntology` fornecendo como argumentos instâncias de conceitos obtidos da ontologia no passo anterior. A operação findServiceByOntology busca de fato por serviços que casam com os conceitos da ontologia informados. Como resultado da operação é retornado uma lista de possíveis serviços e o cliente decide por aquele que melhor satisfaz suas necessidades; caso não seja retornado nenhum serviço ou nenhum atenda a seus requisitos, ele pode encerrar o processo.

Supondo que foi encontrado o serviço de teste JaBUTiWS, o cliente deve então analisar o WSDL do serviço para conhecer a interface de suas operações e em seguida invocar a operação que inicia a atividade de teste, passando o seu programa para ser instrumentado. No caso do serviço JaBUTiWS, a execução do teste envolve algumas interações, entretanto há uma macro-operação que executa os diversos passos de uma única vez.

Obedecendo as regras de governança do serviço JaBUTiWS e de acordo com os requisitos informados, é retornado para o cliente um relatório com a cobertura alcançada de acordo com os critérios escolhidos. Caso o cliente não fique satisfeito com a cobertura, pode-se invocar novamente a operação até que seja atingida a cobertura desejada.

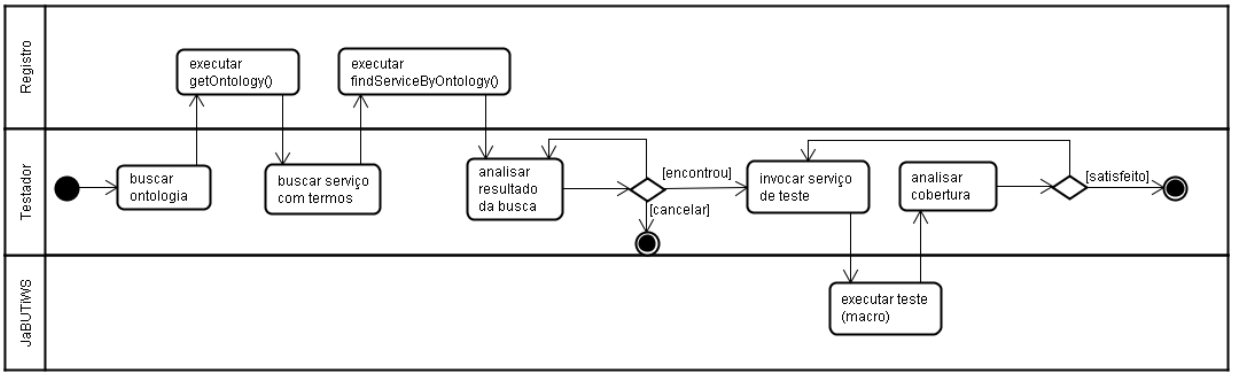

Figura 6.5: Workflow de teste

Este processo discutido de maneira genérica é apresentado em mais detalhes a seguir, conforme ilustra a Figura 6.6 numa visão diferenciada. O testador cria uma aplicação cliente (*script* de teste) para interagir com o registro e encontrar o serviço JaBUTiWS para então realizar o teste de seu programa. O programa em teste é o PMD (PMD, 2008). O PMD é um sistema que consiste de classes para analisar código fonte Java e detectar possíveis defeitos, tais como: código duplicado, possíveis bugs, código não utilizado e expressões complexas.

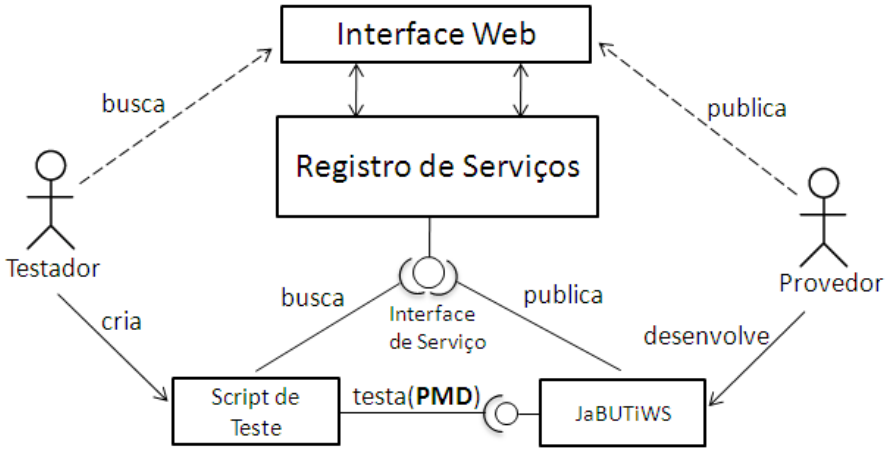

Figura 6.6: Cenário de busca pela interface de serviço

Supondo que o testador irá utilizar o serviço JaBUTiWS, o testador deve concordar as seguintes regras de governança.

- O testador deve permitir que o JaBUTiWS instrumente o *bytecode* Java de sua aplicação em teste.
- O testador deve permitir que o JaBUTiWS faça análise de cobertura com o *bytecode* instrumentado.

A Listagem 6.1 exibe o *script* implementado pelo testador para consultar o serviço de registro e obter o endereço do WSDL do serviço JaBUTiWS. Nas linhas 1 a 2 é invocada a operação para obter a estrutura da ontologia com os conceitos de teste. O elemento `wsInquiry` representa a interface de consulta do serviço de registro. Nas linhas 3 a 4 o testador imprime a ontologia e seleciona os conceitos de interesse que pretende buscar. Na linha 5 é invocada a operação que busca pelo serviço de acordo com os conceitos informados. Na linha 6 é retornado o endereço do serviço de teste encontrado (`accessPoint`).

Listagem 6.1: Script para consultar o registro e obter o WSDL do JaBUTiWS

```
01 TSBInquiry wsInquiry = new TSBInquiry();
02 Document ontology = wsInquiry.getOntology();
03 printOntology(ontology);
04 String selectedConcepts = Console.readString("Select concepts from ontology: ");
05 ServiceInfo[] resultVector = wsInquiry.findServiceByOntology(selectedConcepts);
06 String accessPoint = resultVector[0].getAccessPoint();
```

Uma vez obtido o endereço do serviço de teste desejado (JaBUTiWS), o próximo passo é utilizar o WSDL obtido para gerar um conjunto de programas controladores (*stubs*) que invocam operações do serviço. O testador acessa o JaBUTiWS usando os *stubs* criados. Ele deve selecionar a classe ou lista de classes a serem testadas do sistema PMD.

A Listagem 6.2 exibe o *script* implementado para invocar as operações do serviço JaBUTiWS e instrumentar o sistema PMD. Nas linhas 2 a 6 é criado um projeto de teste. O arquivo `pmd.jar` criado pelo testador contém todas as classes do sistema PMD. Na linha 7 é retornada a identificação do projeto. Nas linhas 8 a 12 são selecionadas as classes a serem instrumentadas. Observe que o testador deseja unicamente testar a classe `Metric` do sistema. Nas linhas 13 a 18 seleciona-se o conjunto de casos de teste para testar o código instrumentado. Nas linhas 19 a 23 é retornado o pacote instrumentado para que o testador possa executar os casos de teste localmente.

Listagem 6.2: Script para criar um projeto de teste e instrumentar o sistema PMD

```
01 JaBUTiWS = new JaBUTiService1_0Stub(acessPoint);
02 CreateProject prjInfo = new CreateProject();
03 prjInfo.setProjectName("PMD-metric");
04 dh = new DataHandler(new FileDataSource(new File("pmd.jar")));
05 prjInfo.setProjectFile(dh);
06 prjID = JaBUTiWS.createProject(prjInfo);
```

```
07 projectid= prjID.get_return();
08 instCl = new SelectClassesToInstrument();
09 instCl.setProjectId(projectid);
10 classes[0] = "net.sourceforge.pmd.stat.Metric";
11 instCl.setClasses(classes);
12 JaBUTiWS.selectClassesToInstrument(instCl);
13 AddTestCases tc = new AddTestCases();
14 tc.setProjectId(projectid);
15 tc.setTestSuiteClass("test.net.sourceforge.pmd.stat.MetricTest");
16 dh = new DataHandler(new FileDataSource(new File("TC.jar")));
17 tc.setTestCaseFile(dh);
18 JaBUTiWS.addTestCases(tc);
19 instProj= new GetInstrumentedProject();
20 instProj.setProjectId(projectid);
21 respInstProj = JaBUTiWS.getInstrumentedProject(instProj);
22 dh = respInstProj.get_return().getFile();
23 dh.writeTo(new FileOutputStream(new File("InstrPack.jar")));
```

Em seguida, para se obter o relatório de cobertura, deve-se executar o pacote instrumentado com os casos de teste para gerar um arquivo de trace. Este arquivo é enviado para o JaBUTiWS usando o *script* mostrado na Listagem 6.3. Nas linhas 1 a 5 é enviado o arquivo para o JaBUTiWS e nas linhas 6 a 9 é realizada uma consulta de cobertura por método.

Listagem 6.3: Script para obter o relatório de cobertura do sistema em teste

```
01 SendTraceFile trace = new SendTraceFile();
02 trace.setProjectId(projectid);
03 dh = new DataHandler(new FileDataSource(new File("pmd.trc")));
04 trace.setTracefile(dh);
05 JaBUTiWS.sendTraceFile(trace);
06 GetCoverageByMethods coverage = new GetCoverageByMethods();
07 coverage.setProjectId(projectid);
08 coverageMethod = JaBUTiWS.getCoverageByMethods(coverage);
09 CoverageDetails coverageMethods[]=coverageMethod.get_return();
```

A Tabela 6.2 mostra o relatório de cobertura obtido. A porcentagem de cobertura é informada para cada método e considera o número de elementos cobertos em relação aos elementos requeridos. Neste caso apenas o construtor possui elementos requeridos para os critérios todas-arestas e todos-usos.

Tabela 6.2: Relatório de cobertura por método

Método	Todos-nós	Todas-arestas	Todos-usos
Metric.Metric()	2/2(100%)	1/1(100%)	8/8(100%)
Metric.getAverage()	1/1(100%)	0/0(0%)	0/0 (0%)
Metric.getCount()	1/1(100%)	0/0(0%)	0/0 (0%)
Metric.getHighValue()	1/1(100%)	0/0(0%)	0/0 (0%)
Metric.getLowValue()	1/1(100%)	0/0(0%)	0/0 (0%)
Metric.getMetricName()	1/1(100%)	0/0(0%)	0/0 (0%)
Metric.getStandardDeviation()	1/1(100%)	0/0(0%)	0/0 (0%)
Metric.getTotal()	1/1(100%)	0/0(0%)	0/0 (0%)

6.4 Considerações Finais

Este capítulo mostrou como o serviço de registro desenvolvido neste trabalho pode ser usado. Em especial, foram apresentadas e discutidas três ferramentas de teste que foram convertidas em serviços e publicadas no registro. Adicionalmente, foram discutidos exemplos de uso do registro por meio da publicação de um serviço e posterior busca. Em relação ao serviço JaBUTiWS, foi mostrado um exemplo de interação entre o serviço e o registro desenvolvido neste trabalho para o teste de um programa Java.

No próximo capítulo são apresentadas as conclusões finais deste trabalho e apresentadas as principais contribuições e trabalhos futuros relacionados com esta dissertação.

CAPÍTULO

7

Conclusão

7.1 Considerações Finais

Este trabalho teve por objetivo oferecer uma contribuição para o domínio de teste de software, por meio da disponibilização de um registro aberto à comunidade para publicação de serviços relacionados a teste de software. O uso do registro de serviços permite aos engenheiros de software selecionarem e comporem serviços que satisfaçam às necessidades de um processo de desenvolvimento particular.

Para concepção do serviço de registro, foi inicialmente detalhada uma arquitetura baseada em serviços para o domínio de engenharia de software, que em seguida foi instanciada para o domínio de teste de software incluindo o registro especializado para ferramentas de teste. Com o objetivo de definir os conceitos utilizados para descrever os serviços registrados e auxiliar nas buscas realizadas, foi estudado e incorporado ao registro uma ontologia que consiste de uma adaptação das principais ontologias no domínio de teste de software.

7.2 Contribuições

Como principal contribuição destaca-se a conceitualização, projeto e implementação de um registro de serviços de ferramentas de teste, sendo o único deste tipo que o autor tem conhecimento.

O registro desenvolvido oferece facilidades de busca por ontologia, que permite que os serviços encontrados sejam mais prováveis de serem aqueles que o usuário procura. Atualmente, três serviços de teste estão publicados no registro: JaBUTiWS (Eler *et al.*, 2009), PlavisWS (Dusse, 2009) e JFutWS (Rocha *et al.*, 2005).

Para inclusão de facilidades de busca por ontologia foi estudada e definida uma ontologia de teste de software específica para ferramentas de teste que inclui conceitos para descrever serviços deste domínio. O estudo realizado considerou algumas das principais ontologias da área de teste, tais como a de Barbosa *et al.* (2006b), Bai *et al.* (2008) e o modelo de teste U2TP (2007).

Com o aumento do número de ferramentas de engenharia de software disponibilizadas como serviços, a abordagem proposta tem como objetivo oferecer um ponto de acesso para publicação e divulgação de tais serviços e, assim, contribuir para a adoção e uso de serviços de engenharia de software nos processos de desenvolvimento de software.

7.3 Trabalhos Futuros

Como trabalhos futuros decorrentes desta dissertação destacam-se: a inclusão de outras ferramentas de teste com o objetivo de descobrir melhorias na versão atual e contribuir com a validação do registro. Também poder-se-ia estender o registro desenvolvido com a incorporação de outras ontologias de domínios distintos. Destaca-se que a ontologia utilizada na implementação atual também pode ser estendida para definir os nomes das operações dos serviços de teste e os tipos de parâmetros. Isso permite que serviços possam seguir regras de governança para facilitar a interoperabilidade e avaliação dos serviços.

Além disso, o processo de teste simplificado desenvolvido para validação do registro e sua interação com o serviço de teste JaBUTiWS pode ser expandido para um contexto mais genérico, no qual se identifica o serviço de teste dinamicamente e executa-se o teste conforme os requisitos do usuário. Desse modo, um aspecto a ser explorado refere-se à execução de experimentos no contexto de um processo de desenvolvimento de software, com o objetivo de auxiliar na definição desse processo genérico de busca e automatização do uso de serviços de teste disponíveis no registro.

Outro trabalho futuro está relacionado com a padronização das interfaces dos serviços de teste registrados. Em geral, as operações e os tipos de parâmetros diferem de serviço para serviço, mesmo quando são oferecidas funcionalidade similares. Isso dificulta a descoberta automática e a integração de serviços. Desse modo, a definição de uma interface padrão para

os serviços de teste é um passo essencial para oferecer o dinamismo e a integração de serviços como prometido pela abordagem SOA.

É importante destacar que no futuro deverá existir um número grande de serviços de engenharia de software publicados em registros específicos. Possivelmente nem todos os tipos de ferramentas de engenharia de software possam ser transformadas em serviços. Por exemplo, ferramentas com alto grau de interação com o usuário, provavelmente não são boas candidatas a serem transformadas em serviços. Assim, outra linha de pesquisa seria estudar e definir quais características as ferramentas devem possuir para serem convertidas em serviços ou desenvolvidas diretamente como serviços.

Referências Bibliográficas

AL-MASRI, E.; MAHMOUD, Q. H. Investigating web services on the world wide web. In: *Proceeding of the 17th International Conference on World Wide Web (WWW '08)*, New York, NY, USA: ACM, 2008, p. 795–804.

ALEXANDER, R. T. Aspect-Oriented Programming: the Real Costs? *IEEE Software*, v. 20, n. 6, p. 90–93, 2003.

ALEXANDER, R. T.; BIEMAN, J. M.; ANDREWS, A. A. *Towards the systematic testing of aspect-oriented programs*. Technical report, Department of Computer Science, Colorado State University, 2004.

APACHE Apache Tomcat. Disponível em: `http://tomcat.apache.org/`. Último acesso: 21/02/2010, 2004.

APACHE JUDDI Apache Web Services Project. Disponível em: `http://ws.apache.org/juddi/`. Último acesso: 28/03/2010, 2004.

ATKINSON, C.; BOSTAN, P.; HUMMEL, O.; STOLL, D. A practical approach to web service discovery and retrieval. In: *Proceedings of the 5th IEEE International Conference on Web Services (ICWS '07)*, Salt Lake City, UT, USA: IEEE Computer Society, 2007, p. 241–248.

BAI, X.; LEE, S.; TSAI, W.-T.; CHEN, Y. Ontology-based test modeling and partition testing of web services. In: *Proceedings of the 6th IEEE International Conference on Web Services (ICWS '08)*, Washington, DC, USA: IEEE Computer Society, 2008, p. 465–472.

BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R.; DELAMARO, M. E. *Teste estrutural e de mutação no contexto de programas OO*. Instituto de Ciências Matemáticas e de Computação – ICMC-USP, Nota Didática n. 69, 2006a.

BARBOSA, E. F.; NAKAGAWA, E. Y.; MALDONADO, J. C. Towards the establishment of an ontology of software testing. In: *Proceedings of the 18th International Conference on Software Engineering & Knowledge Engineering (SEKE'06)*, San Francisco, CA, USA, 2006b, p. 522–525.

BARBOSA, E. F.; NAKAGAWA, E. Y.; RIEKSTIN, A. C.; MALDONADO, J. C. Ontology-based development of testing related tools. In: *Proceedings of the 20th International Conference on Software Engineering & Knowledge Engineering (SEKE'08)*, Knowledge Systems Institute Graduate School, 2008, p. 697–702.

BARTOLINI, C.; BERTOLINO, A.; ELBAUM, S.; MARCHETTI, E. Whitening SOA Testing. In: *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symp on the FSE*, New York, NY, USA, 2009, p. 161–170.

BECK, K.; GAMMA, E. JUnit Cookbook. Disponível em: `http://junit.sourceforge.net/doc/cookbook/cookbook.htm`. Último acesso: 22/02/2010, 2005.

BERTOLINO, A.; POLINI, A. SOA test governance: Enabling service integration testing across organization and technology borders. In: *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, Washington, DC, USA: IEEE Computer Society, 2009, p. 277–286.

BINDER, R. V. *Testing object-oriented systems: Models, patterns, and tools*, v. 1. Addison Wesley Longman, Inc., 1999.

BUDD, T. A. Mutation analysis: Ideas, example, problems and prospects. In: *Software Testing*, North Holland Publishing Company, p. 129–148, 1981.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. *Pattern-oriented software architecture: A system of patterns*, v. 1. New York, NY, USA: John Wiley & Sons, Inc., 1996.

CANFORA, G.; PENTA, M. D. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, v. 8, n. 2, p. 10–17, 2006.

CERAMI, E. *Web Services Essentials*. 1 ed. O'Reilly, 2002.

CHAIM, M. L. *Poke-Tool - Uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados*. Dissertação de Mestrado, DCA/FEEC/UNICAMP, Campinas, SP, Brasil, 1991.

CHAIM, M. L.; CARNIELLO, A.; JINO, M. Teste baseado em casos de uso. Boletim de Pesquisa e Desenvolvimento. Disponível em: `http://www.cnptia.embrapa.br/files/bp10.pdf`. Último acesso: 08/04/2010, 2003.

COBERTURA Java coverage testing tool. Disponível em: `http://cobertura.sourceforge.net/`. Último acesso: 05/04/2010, 2010.

CODECOVER TESTING An open-source glass box testing tool. Disponível em: `http://codecover.org/`. Último acesso: 24/02/2010, 2007.

COVERLIPSE PLUGIN An open-source eclipse plugin for JUnit. Disponível em: `http://coverlipse.sourceforge.net/`. Último acesso: 24/02/2010, 2005.

DAN, X.; SHI, Y.; TAO, Z.; XIANG-YANG, J.; ZAO-QING, L.; JUN-FENG, Y. An approach for describing SOA. In: *Proceedings of International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM '06)*, 2006, p. 1–4.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao Teste de Software*, v. 1. 1 ed. Rio de Janeiro: Elsevier, 2007.

DEMILLO, R. A. Mutation analysis as a tool for software quality assurance. In: *Proceedings of the 4th Annual International Conference on Computer Software and Applications (COMPSAC'80)*, Chicago, IL, USA, 1980.

DEMILLO, R. A.; MCCRACKEN, W. M.; MARTIN, R. J.; PASSAFIUME, J. F. *Software testing and evaluation*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1987.

DOM4J Open source XML library. Disponível em: `http://dom4j.sourceforge.net/`. Último acesso: 23/04/2010, 2010.

DOMINGUES, A. L. S. *Avaliação de critérios e ferramentas de teste para programas OO*. Dissertação de Mestrado, ICMC-USP, São Carlos, SP, Brasil, 2002.

DUARTE, K. C.; FALBO, R. A. Uma ontologia de qualidade de software. In: *Anais do Workshop de Qualidade de Software (WQS'00) – 14º Simpósio Brasileiro de Engenharia de Software*, João Pessoa, PB, Brasil, 2000.

DUSSE, F. *Avaliação de custo e eficácia de métodos e critérios de teste baseado em Máquinas de Estados Finitos*. Dissertação de Mestrado, ICMC-USP, São Carlos, SP, Brasil, 2009.

ECLEMMA PLUGIN Java code coverage plugin for eclipse. Disponível em: `http://www.eclemma.org/`. Último acesso: 24/02/2010, 2006.

ECLIPSE The Eclipse Fundation. Disponível em: `http://eclipse.org/`. Último acesso: 20/02/2010, 2004.

ELER, M. M. *Um serviço para o teste estrutural e certificação de serviços web e componentes de software*. Qualificação de doutorado, ICMC-USP, São Carlos, SP, Brasil, 2008.

ELER, M. M.; DELAMARO, M. E.; MALDONADO, J. C.; MASIERO, P. C. Built-in structural testing of web services. In: *(submitted for publication)*, 2010.

ELER, M. M.; ENDO, A. T.; MASIERO, P. C.; DELAMARO, M. E.; MALDONADO, J. C.; VINCENZI, A. M. R.; CHAIM, M. L.; BEDER, D. M. JaBUTiWS: A web service for structural testing of java programs. *In: Software Engineering Workshop. IEEE Computer Society*, 2009.

EMMA A free Java code coverage tool. Disponível em: `http://emma.sourceforge.net/`. Último acesso: 28/02/2010, 2006.

ERL, T. *Service-oriented architecture: Concepts, technology, and design*. 1 ed. Prentice Hall PTR, 2005.

ESIGMA Registro público esigma. Disponível em: `http://www.esigma.com/`. Último acesso: 08/04/2010, 2003.

FALBO, R. A.; MENEZES, C. S.; ROCHA, A. R. A systematic approach for building ontologies. In: *Proceedings of the 6th Ibero-American Conference on AI (IBERAMIA'98)*, Lisbon, Portugal: Springer, 1998, p. 349–360 (*Lecture Notes in Computer Science*, v.1484).

FENSEL, D. *Ontologies: A silver bullet for knowledge management and electronic commerce*. 2 ed. Springer, 2003.

FREITAS, F.; STUCKENSCHMIDT, H.; NOY, N. F. Ontology issues and applications - guest editors' introduction. *Journal of the Brazilian Computer Society (JBCS)*, v. 11, n. 2, 2005.

FUGGETTA, A. Software process: a roadmap. In: *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, New York, NY, USA: ACM, 2000, p. 25–34.

GHEZZI, G.; GALL, H. C. Towards software analysis as a service. In: *23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, p. 1–10.

GRUBER, T. R. Toward principles for the design of ontologies used for knowledge sharing. *International J. Hum.-Comput. Stud.*, v. 43, n. 5-6, p. 907–928, 1995.

HARROLD, M. J. Testing: A Roadmap. In: *Proceedings of the 22th International Conference on Software Engineering - Future of SE Track (ICSE '00)*, ACM Press, 2000, p. 61–72.

HARROLD, M. J.; ROTHERMEL, G. Performing data flow testing on classes. In: *Proceedings of the 2nd ACM SIGSOFT symposium on Fundations of software engineering*, New York, NY: ACM Press, 1994, p. 154–163.

HECKEL, R.; MARIANI, L. Automatic conformance testing of web services. In: *Fundamental Approaches to Software Engineering (FASE '05)*, Edinburgh, Scotland, UK, 2005, p. 34–48.

HIBERNATE Relational Persistence for Java and .NET. Disponível em: `http://hibernate.org/`. Último acesso: 16/02/2010, 2005.

HUO, Q.; ZHU, H.; GREENWOOD, S. A multi-agent software environment for testing web-based applications. In: *Proceedings of the 27th Annual International Conference on Computer Software and Applications (COMPSAC '03)*, Washington, DC, USA: IEEE Computer Society, 2003, p. 210.

HYLAND-WOOD, D.; CARRINGTON, D.; KAPLAN, S. Towards a software maintenance methodology using semantic web techniques. In: *Proceedings of the 2nd International IEEE Workshop on Software Evolvability*, Washington, DC, USA: IEEE Computer Society, 2006, p. 23–30.

IEEE *IEEE standard glossary of Software Engineering terminology*. Padrão 610.12, IEEE Press, 1990.

JOSUTTIS, N. M. *SOA in practice*. 1 ed. O'Reilly, 2007.

JUDE Jude UML Modeling Tool. Disponível em: `http://jude.change-vision.com/jude-web/index.html`. Último acesso: 26/02/2010, 2006.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. An overview of AspectJ. In: *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, 2001, p. 327–353.

KICZALES, G.; LAMPING, J.; MENHDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.-M.; IRWIN, J. Aspect-Oriented Programming. In: AKŞIT, M.; MATSUOKA, S., eds. *Proceedings European Conference on Object-Oriented Programming*, v. 1241, Berlin, Heidelberg, and New York: Springer-Verlag, p. 220–242, 1997.

KREGER, H. Web Services Conceptual Architecture (WSCA 1.0). *International Business Machines Corporation (IBM)*, 2001.

LEE, R.; KIM, H. K.; YANG, H. S. An architecture model for dynamically converting components into web services. In: *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC '04)*, Washington, DC, USA: IEEE Computer Society, 2004, p. 648–654.

LEMOS, O. A. L. *Teste de programas orientados a aspectos: uma abordagem estrutural para AspectJ*. Dissertação de Mestrado, ICMC-USP, São Carlos, SP, Brasil, 2005.

LEMOS, O. A. L.; MALDONADO, J. C.; MASIERO, P. C. Data flow integration testing criteria for aspect-oriented programs. In: *Anais do* 1^o *Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP'2004)*, Brasília, DF, Brasil, 2004.

LINDVALL, M.; RUS, I. Guest editors' introduction: Process diversity in software development. *IEEE Software*, v. 17, n. 4, p. 14–18, 2000.

MA, K. J. Web services: What's real and what's not? *IT Professional*, v. 7, n. 2, p. 14–21, 2005.

MALDONADO, J. C. *Critérios potenciais usos: Uma contribuição ao teste estrutural de software*. Tese de Doutorado, DCA/FEE/UNICAMP, Campinas, SP, Brasil, 1991.

MASIERO, P. C.; LEMOS, O. A. L.; FERRARI, F. C.; MALDONADO, J. C. *Atualizações em Informática*, cap. Teste de Software Orientado a Objetos e a Aspectos: Teoria e Prática Rio de Janeiro: Editora PUC-Rio, p. 13–71, 2006.

MCGREGOR, J. D.; KORSON, T. D. Integrated object-oriented testing and development processes. *Communications ACM*, v. 37, n. 9, p. 59–77, 1994.

METRICS Eclipse Metrics. Disponível em: `http://metrics.sourceforge.net/`. Último acesso: 16/04/2010, 2009.

MØLLER, A.; SCHWARTZBACH, M. I. *An introduction to XML and Web technologies*. Addison-Wesley, 2006.

MONTANGERO, C.; DERNIAME, J. C.; KABA, B. A.; WARBOYS, B. The software process: Modelling and technology. In: *Software Process: Principles, Methodology, Technology*, Springer, 1999, p. 1–14 (*Lecture Notes in Computer Science*, v.1500).

MYERS, G. J.; SANDLER, C.; BADGETT, T.; THOMAS, T. M. *The art of software testing*. John Wiley & Sons, Inc., Hoboken, New Jersey, USA, 2004.

NOVELL Novell UDDI Server. Disponível em: `http://developer.novell.com/wiki/index.php/Uddiserver`. Último acesso: 24/02/2010, 2007.

OASIS ebXML Specifications TC. Disponível em: `http://www.oasis-open.org/committees/regrep/`. Último acesso: 22/02/2010, 2003.

OASIS UDDI Specifications TC. Disponível em: `http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm`. Último acesso: 16/04/2010, 2005.

OASIS OASIS ebXML Registry. Disponível em: `http://ebxmlrr.sourceforge.net/`. Último acesso: 25/03/2010, 2007.

OASIS Web services business process execution language (WSBPEL) v2.0. Disponível em: `http://docs.oasis-open.org/wsbpel/2.0/`. Último acesso: 18/05/2010, 2007.

OFFUTT, J.; LIU, S.; ABDURAZIK, A.; AMMANN, P. Generating test data from state-based specifications. *The Journal of Software Testing, Verification and Reliability*, v. 13, n. 1, p. 25–53, 2003.

OPENUDDI Open Source UDDI. Disponível em: `http://openuddi.sourceforge.net/`. Último acesso: 10/04/2010, 2008.

ORACLE Oracle Service Registry. Disponível em: `http://www.oracle.com/technologies/soa/service-registry.html`. Último acesso: 24/03/2010, 2008.

ORT, E. Service-oriented architecture and web services: Concepts, technologies, and tools. *Sun Microsystems Inc.*, Disponível em: `http://java.sun.com/developer/technicalArticles/WebServices/soa2/WSProtocols.html`. Último acesso: 20/01/2010, 2005.

PAPAZOGLOU, M. P. Service -oriented computing: Concepts, characteristics and directions. In: *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE '03)*, Washington, DC, USA: IEEE Computer Society, 2003, p. 3.

PAPAZOGLOU, M. P.; HEUVEL, W. J. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, v. 16, n. 3, p. 389–415, 2007.

PARASOFT CORPORATION Parasoft C++ Test. Disponível em: `http://www.parasoft.com/ctest/`. Último acesso: 25/02/2010, 2005.

PARASOFT CORPORATION Parasoft JTest. Disponível em: `http://www.parasoft.com/jtest/`. Último acesso: 11/03/2010, 2009.

PMD Projeto PMD. Disponível em: `http://pmd.sourceforge.net/`. Último acesso: 14/03/2010, 2008.

POSTGRESQL PostgreSQL Global Development Group. Disponível em: `http://postgresql.org/`. Último acesso: 26/02/2010, 1996.

POTIX-CORPORATION Open Source Ajax. Disponível em: `http://www.zkoss.org/`. Último acesso: 25/02/2010, 2005.

PRESSMAN, R. S. *Software engineering: A practitioner's approach*. 6 ed. McGraw-Hill, 2005.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, v. 11, n. 4, p. 367–375, 1985.

ROCHA, A. D.; MALDONADO, J. C.; MASIERO, P. C. Uma ferramenta baseada em aspectos para o teste funcional de programas Java. In: *Anais do* 19^o *Simpósio Brasileiro de Engenharia de Software*, Uberlandia, MG, Brasil, 2005.

SEEKDA Web services search engine. Disponível em: `http://webservices.seekda.com/`. Último acesso: 09/04/2010, 2006.

SIMÃO, A. S.; AMBRÓSIO, A. M.; FABBRI, S. C. P. F.; AMARAL, A. S. M. S.; MARTINS, E.; MALDONADO, J. C. Plavis/FSM: An integrated plataform for validating FSM based system. In: *Latin-American Symposium on Dependable Computing*, Salvador, BA, Brazil, 2005.

SINHA, S.; HARROLD, M. Criteria for testing exception-handling constructs in java programs. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*, p. 265–274, 1999.

SOAP SOAP Specifications. Disponível em: `http://www.w3.org/TR/soap/`. Último acesso: 20/02/2010, 2003.

TSAI, W.; PAUL, R.; CAO, Z.; YU, L.; SAIMI, A. Verification of web services using an enhanced UDDI server. *Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '03)*, p. 131–138, 2003.

TSAI, W. T.; CHEN, Y.; PAUL, R. A. Specification-based verification and validation of web services and service-oriented operating systems. In: *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '05)*, Washington, DC, USA: IEEE Computer Society, 2005, p. 139–147.

TSAI, W. T.; ZHOU, X.; CHEN, Y.; XIAO, B.; PAUL, R. A.; CHU, W. Roadmap to a full service broker in service-oriented architecture. In: *Proceedings of the IEEE International Conference on e-Business Engineering (ICEBE '07)*, Washington, DC, USA: IEEE Computer Society, 2007, p. 657–660.

U2TP UML Testing Profile Specification. Disponível em: `http://www.omg.org/technology/documents/formal/test_profile.htm`. Último acesso: 26/02/2010, 2007.

VILLELA, K. *Definição e construção de ambientes de desenvolvimento de software orientados à organização*. Tese de Doutorado, COPPE-UFRJ, Rio de Janeiro, RJ, Brasil, 2004.

VINCENZI, A. M. R. *Orientação a objeto: Definição e análise de recursos de teste e validação*. Tese de Doutorado, ICMC-USP, São Carlos, SP, Brasil, 2004.

W3C Web Services Description Language (WSDL). Disponível em: `http://www.w3.org/TR/wsdl`. Último acesso: 02/02/2010, 2001.

WINDOWS UDDI SERVICES Microsoft Windows Enterprise UDDI Services. Disponível em: `http://uddi.microsoft.com/`. Último acesso: 24/02/2010, 2003.

XIE, T.; ZHAO, J.; MARINOV, D.; NOTKIN, D. Detecting redundant unit tests for AspectJ programs. In: *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE '06)*, Washington, DC, USA: IEEE Computer Society, 2006, p. 179–190.

XMETHODS Registro público xmethods. Disponível em: `http://www.xmethods.net/`. Último acesso: 25/02/2010, 2008.

XPLANNER A web-based project planning and tracking tool for xp teams. Disponível em: `http://www.xplanner.org/`. Último acesso: 05/04/2010, 2006.

YEN, I.-L.; MA, H.; BASTANI, F. B.; MEI, H. QoS-reconfigurable web services and compositions for high-assurance systems. *Computer magazine, IEEE Computer Society*, v. 41, n. 8, p. 48–55, 2008.

ZHAO, J. Tool support for unit testing of aspect-oriented software. In: *Workshop on Tools for Aspect-Oriented Software Development (OOPSLA '02)*, Seattle, WA, USA, 2002.

ZHAO, J. Data-flow-based unit testing of aspect-oriented programs. In: *Proceedings of the 27th Annual IEEE International Conference on Computer Software and Applications (COMPSAC'2003)*, Dallas, TX, USA: IEEE Computer Society, 2003, p. 188–197.

ZHU, H.; HALL, P. A. V.; MAY, J. H. R. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, v. 29, n. 4, p. 366–427, 1997.

APÊNDICE

A

Documento de Requisitos do Registro de Serviços de Ferramentas de Teste

Introdução

Propósito

Este documento descreve os requisitos de software para o sistema de registro de serviços de teste de software. O sistema tem como função dar apoio à publicação, busca e classificação de serviços relacionados a ferramentas de teste. O sistema oferece opções de classificação e busca por meio de uma ontologia de teste de software.

Visão Geral

O restante deste documento está organizado como segue: inicialmente, definem-se alguns termos importantes para entendimento do documento na Seção A. Em seguida, apresenta-se uma descrição geral do registro. Por último, são identificados os requisitos funcionais específicos e os requisitos de desempenho do sistema de registro.

Definições

- **Serviço**: Representa uma funcionalidade de negócio criada de maneira fracamente acoplada que é encapsulada como um serviço e disponibilizada pela internet.
- **Operação**: Representa uma ação específica executada pelo serviço. Uma operação pode ser comparada a um método público utilizado em componentes tradicionais.

- **Registro**: Sistema no qual são oferecidas opções de publicação, busca e classificação de serviços.
- **Ontologia**: Conjunto de conceitos e relações que descrevem certo domínio de conhecimento.
- **Palavras-chave**: Conjunto de palavras utilizadas para realizar uma busca por serviços catalogados.
- **Usuário**: Refere-se aos usuários do sistema podendo ser o provedor, administrador, pessoa ou programa que utiliza o sistema de registro para encontrar, publicar ou manter um determinado serviço Web.
- **Provedor**: Organização ou pessoa que disponibiliza algum serviço Web e o publica no registro de serviços.
- **Cliente**: Pessoa ou programa interessado em utilizar o serviço Web desenvolvido por um provedor.
- **Administrador**: Pessoa responsável por gerenciar o sistema. Ele pode incluir, editar, remover ou verificar a disponibilidade de um serviço Web.

Descrição Geral

O software de registro tem por objetivo ser um registro de serviços padronizado, no qual um provedor pode armazenar a especificação de seus serviços, tanto para uso próprio quanto para disponibilizar os serviços externamente. Ele permite aos consumidores (ou clientes) de serviços encontrarem serviços que melhor atendam suas necessidades de acordo com determinado critério de busca. O software deve permitir a busca e publicação de serviços, podendo-se buscar serviços com o uso de palavras-chave ou por uma interface de serviço. A publicação dos serviços pode ser feita manualmente por um humano ou automaticamente por um programa. O sistema permite a navegação por meio da ontologia para auxiliar a busca dos serviços que atendam aos critérios selecionados.

Requisitos Específicos

Requisitos Funcionais

R1 - O sistema deve permitir ao provedor do serviço publicar os serviços por ele desenvolvidos. A publicação é realizada por meio de uma interface de publicação de serviços, onde se deve fornecer os seguintes dados: nome, descrição, URL e WSDL do serviço. O WSDL contém as operações disponíveis, os parâmetros de entrada e saída de cada operação, os tipos de dados que dão suporte às definições do serviço e o formato das mensagens trocadas.

R2 - O sistema deve permitir, por meio da interface de publicação, que provedores de serviços registrem os detalhes da organização, tais como nome, descrição e URL dos serviços oferecidos. Também se deve permitir alterar ou excluir essas informações se houver necessidade.

R3 - O sistema deve permitir ao cliente encontrar serviços desejados por meio de uma interface de consulta. Além disso, pode-se consultar determinado serviço e seus detalhes de invocação por um programa utilizando uma interface de serviço. Durante a consulta, três tipos de buscas podem ser realizadas:

1. Por meio de uma listagem dos serviços oferecidos;
2. Por meio de palavras-chave;
3. Por último, navegando-se pela ontologia de teste.

R4 - O sistema deve permitir encontrar detalhes dos provedores do serviço e também informações de outros provedores que ofereçam serviços relacionados.

R5 - O sistema de registro deve possuir uma interface gráfica para facilitar o uso do registro pelos humanos, oferecendo opções para inserção, atualização ou busca de serviços.

R6 - Para cada serviço armazenado no registro além das operações, parâmetros e URL de acesso. Devem ser armazenadas informações sobre a ontologia de teste, de modo a permitir uma classificação ontológica dos serviços catalogados.

R7 - O sistema deve permitir consultar se um serviço está disponível ou não para uso, por meio de uma invocação de teste. Esta consulta deve ser feita diretamente pelo serviço de registro. Primeiramente, deve-se selecionar uma operação do serviço que se deseja invocar, entrar com os parâmetros e visualizar a mensagem SOAP da requisição. Por último, a mensagem pode ser enviada para o serviço a fim de verificar o resultado.

R8 - O sistema deve classificar os serviços catalogados de acordo com a ontologia de teste. Deve-se perguntar quais fases, técnicas, critérios, artefatos e linguagem de programação o serviço apoia. A classificação é realizada na etapa de publicação do serviço, após terem sido fornecidos as informações básicas do serviço.

R9 - O sistema deve possuir uma área administrativa para gerenciar (incluir, editar, remover) os serviços catalogados.

Requisitos de Desempenho

R10 - O sistema deve responder as requisições dos usuários em menor tempo possível. Mensagens de erro também devem ser exibidas no máximo 30 segundos após a ocorrência do erro.

Atributos

Disponibilidade

R11 - O sistema de registro deve ficar disponível vinte e quatro horas por dia.

Segurança

R12 - O sistema deve permitir uma forma de autenticação por meio de senhas de acesso e identificação para os tipos de usuários: provedor e administrador. A autenticação é requerida para as operações de publicação, atualização ou exclusão de serviços ou provedores do sistema.

Manutenção

R13 - Se houver necessidade de manutenção, ela deve ser anunciada com o mínimo de 12 horas de antecedência. Somente os administradores estão autorizados a realizar manutenção no sistema de registro.

Banco de Dados

R14 - As consultas efetuadas no sistema de registro devem ter todas as propriedades de transações em uma base de dados (atomicidade, consistência, isolamento e durabilidade). Além disso, as descrições dos serviços, ontologias e outros artefatos do registro devem ser armazenados permanentemente.