Estabelecimento de uma arquitetura de referência orientada a serviços para ferramentas de teste de software

Lucas Bueno Ruas de Oliveira

Estabelecimento de uma arquitetura de referência orientada a serviços para ferramentas de teste de software

Lucas Bueno Ruas de Oliveira

Orientadora: ***Profa. Dra. Elisa Yumi Nakagawa***

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

USP – São Carlos
Dezembro/2010

À professora da minha vida,
minha mãe, Rosa Maria Bueno.

Agradecimentos

A minha mãe, Rosa Maria Bueno, por tudo. Agradeço pelo exemplo, educação, dedicação e principalmente pelo carinho. Não tenho palavras suficientes para descrever o quanto sou grato a você, por ter dedicado sua vida a minha criação.

À orientadora e amiga Prof[a]. Dr[a]. Elisa Yumi Nakagawa, pelos ensinamentos, conselhos e por toda sua dedicação durante o desenvolvimento deste trabalho. Você é um exemplo de esmero, otimismo e empenho para seus orientandos. Muito obrigado.

A toda minha família. Em especial, a minha segunda mãe Ivone (Dida), pelo cuidado e afeto incondicionais. A minha irmã Michelle, por sempre apoiar e incentivar minhas decisões. E aos meus padrinhos Hideyo e Lourdes, pelo carinho.

Aos grandes e bons amigos de Pouso Alegre: Gustavo, Guima, Eloy e Marina. Vocês são, e sempre serão, os irmãos que eu tive o prazer de escolher.

À República Lahma, local onde tive a oportunidade de aprender muito sobre amizade, companheirismo e como curtir a vida.

A minha namorada Juliana, sempre muito paciente, compreensiva, companheira e carinhosa.

Aos meus colegas de república Felipe, Thiago e Capi, e às eternas "Vezênhas" Giu, Milena, Linão, Mary, Carol e Gabi. O meu primeiro ano em São Carlos não teria sido o mesmo sem a companhia de vocês.

Aos amigos do LabES, pelo companheirismo durante esses dois anos: Rafael Oliveira (Frota), Fabiano, André Endo, Marco Aurélio (Marcão), Vinícius, Gabriel (Ceará), Adriano (D'), David, Maria Adelina, Bruno, Jorge Cutigi (Piu), Paulo Nardi, Rodrigo, André Abe, Katia, Sandro (KLB), Paulo Henrique (Nerso), Rafael Messias, Vania, Rodolfo, André Domingues

(Andrezinho), Marcelo Eler, Rodrigo Gondim, Flávio Dusse (Sorim), Edson Oliveira, Erika Höhn, Otávio Lemos, Viviane Malheiros, Draylson, Daniel, Arineiza, Joice, Eduardo, Silvana, Thiago, Harry e Rodolfo (Resina).

Aos amigos Pablo, Cássio Prazeres, Danilo (Pimpa), Tácito e Maycon.

Aos bons professores que tive durante a minha graduação na Universidade Federal de Itajubá. Em especial, gostaria de agradecer ao amigo Prof. Dr. Laércio Baldochi e ao Prof. Dr. Enzo Seraphim, que me incentivaram e foram fundamentais para o meu ingresso no mestrado.

Aos professores do ICMC que contribuíram para a minha formação: Paulo C. Masiero, Graça Nunes, Rosana Braga, Simone e Márcio Delamaro.

Aos amigos que ajudaram na revisão desta dissertação: André Endo, Rafael (Frota), Paulo Nardi, Rodrigo e Marcão. Gostaria de agradecer, especialmente, ao amigo Endo, com quem aprendi muito durante o desenvolvimento deste trabalho.

A todos os funcionários do ICMC, pela excelência nos serviços prestados. Em especial, às funcionarias da secretaria da pós-graduação, sempre prestativas e atenciosas, e aos bem-humorados funcionários da segurança.

Obrigado a todas as pessoas que contribuíram de alguma forma para o desenvolvimento deste trabalho.

À Universidade de São Paulo.

Ao CNPq e à FAPESP pelo apoio financeiro.

Há homens que lutam um dia e são bons;
Há outros que lutam um ano e são melhores;
Há os que lutam muitos anos e são muito bons;
Porém, há os que lutam toda a vida,
Esses são os imprescindíveis.

Bertolt Brech

Resumo

O teste de software é reconhecido como uma importante atividade na garantia da qualidade de sistemas de software. Com o objetivo de dar apoio a essa atividade, uma diversidade de ferramentas de teste têm sido desenvolvida. Entretanto, grande parte dessas ferramentas é construída de forma isolada, possuindo arquiteturas e estruturas próprias, o que tem impactado negativamente a capacidade de integração e o reúso dessas ferramentas. Nesse contexto, esforços têm sido dedicados à disponibilização de ferramentas de teste orientadas a serviço, ou seja, ferramentas que são baseadas na SOA (*Service Oriented Architecture*). Em uma outra perspectiva, arquiteturas de referência têm desempenhado um importante papel no desenvolvimento de sistemas de software, uma vez que contém informações sobre como desenvolver sistemas para um determinado domínio de aplicação, buscando contribuir para o sucesso de sistemas desse domínio. Assim, o principal objetivo deste trabalho é o estabelecimento de uma arquitetura de referência orientada a serviço, denominada RefTEST-SOA (*Reference Architecture for Software Testing Tools based on SOA*), que agrega o conhecimento e a experiência de como organizar ferramentas de teste orientadas a serviço, visando também à integração, à escalabilidade e o reúso providos pela SOA. Para o estabelecimento dessa arquitetura, foi utilizado o ProSA-RA, um processo que sistematiza o projeto, representação e avaliação de arquiteturas de referência. Resultados alcançados no estudo de caso conduzido evidenciam que a RefTEST-SOA é uma arquitetura viável e reusável para o desenvolvimento de ferramentas de teste orientadas a serviço.

Abstract

Software testing is considered as an important activity to ensure the quality of software systems. To support such activity, a diversity of testing tools have been developed. However, most of them have been separately built and have usually their particular structures and architectures, which has hindered the integration and reuse of these tools. In this context, efforts have been employed in order to provide service-oriented testing tools, i.e., tools that are based on SOA (Service Oriented Architecture). In another perspective, reference architectures have played an important role in the development of software systems, since they contain information about how to develop systems for a particular application domain, contributing to the success of systems in that domain. Thereby, our main objective is to establish a service-oriented reference architecture, named RefTEST-SOA (Reference Architecture for Software Testing Tools based on SOA), which aggregates the knowledge and experience about how to organize service-oriented testing tools, also aiming at integration, scalability and reuse provided by SOA. To establish this architecture, we have used ProSA-RA, a process that provides guidelines to the design, representation and evaluation of reference architectures. Results achieved by a conducted case study indicate that RefTEST-SOA is a viable and reusable architecture for developing service-oriented testing tools.

Sumário

1 Introdução **1**
1.1 Contextualização . . . 1
1.2 Motivação . . . 3
1.3 Objetivo . . . 4
1.4 Organização . . . 5

2 Teste de Software: Uma Visão Geral **7**
2.1 Considerações Iniciais . . . 7
2.2 Terminologia e Conceitos Básicos . . . 8
2.3 Técnicas e Critérios de Teste . . . 9
2.3.1 Teste Funcional . . . 10
2.3.2 Teste Estrutural . . . 11
2.3.3 Teste Baseado em Defeitos . . . 13
2.4 Automatização do Teste de Software . . . 14
2.5 Processo de Teste de Software . . . 16
2.6 Considerações Finais . . . 17

3 Arquitetura de Software **19**
3.1 Considerações Iniciais . . . 19
3.2 Arquitetura de Software . . . 20
3.2.1 Terminologia e Conceitos Básicos . . . 21
3.2.2 Representação de Arquiteturas de Software . . . 23
3.2.3 Métodos de Avaliação Arquitetural . . . 24
3.3 Arquitetura de Referência . . . 25
3.3.1 Classificações de Arquiteturas de Referência . . . 26
3.3.2 Arquiteturas de Referência de Teste de Software . . . 29
3.3.3 Processo para o Estabelecimento de Arquiteturas de Referência . . . 31
3.4 Arquitetura Orientada a Serviço . . . 34
3.4.1 Conceitos Básicos . . . 34
3.4.2 Composição de Serviços . . . 36
3.4.3 *Enterprise Service Bus* . . . 38
3.4.4 Qualidade de Serviço . . . 40

3.4.5 Serviços Web . . . 41
3.4.6 Arquiteturas e Modelos de Referência Orientados a Serviço . . . 43
3.5 Considerações Finais . . . 44

4 Estabelecimento da RefTEST-SOA 47
4.1 Considerações Iniciais . . . 47
4.2 Passo RA-1: Investigação das Fontes de Informação . . . 48
4.2.1 Grupo 1: Arquiteturas e Ferramentas Orientadas a Serviço de Teste, de Verificação e de Análise de Software . . . 48
4.2.2 Grupo 2: Diretrizes para o Desenvolvimento de Sistemas Orientados a Serviço . . . 51
4.2.3 Grupo 3: Arquiteturas de Referência Orientadas a Serviço . . . 56
4.2.4 Grupo 4: Arquiteturas de Referência para o Domínio de Teste . . . 58
4.3 Passo RA-2: Estabelecimento dos Requisitos Arquiteturais . . . 66
4.3.1 Requisitos Arquiteturais do Domínio de Teste de Software . . . 66
4.3.2 Requisitos Arquiteturais do Contexto de Serviços . . . 68
4.4 Passo RA-3: Projeto Arquitetural . . . 70
4.4.1 Visão Geral . . . 70
4.4.2 Visão de Módulo . . . 75
4.4.3 Visão em Tempo de Execução . . . 77
4.4.4 Visão de Implantação . . . 79
4.5 Passo RA-4: Avaliação da Arquitetura de Referência . . . 80
4.6 Considerações Finais . . . 80

5 Estudo de Caso: Utilização da RefTEST-SOA 83
5.1 Considerações Iniciais . . . 83
5.2 Instanciação da RefTEST-SOA para o Critério Análise de Mutantes . . . 84
5.3 Descrição dos Serviços de Teste . . . 86
5.3.1 Serviço de Gerenciamento de Artefatos de Teste . . . 86
5.3.2 Serviço de Gerenciamento de Requisitos de Teste . . . 88
5.3.3 Serviço de Gerenciamento de Critérios de Teste . . . 91
5.3.4 Serviço de Gerenciamento de Casos de Teste . . . 93
5.4 Integrando Serviços de Teste . . . 94
5.5 Implementação e Tecnologias Utilizadas . . . 96
5.6 Exemplo de Utilização . . . 97
5.7 Análise Preliminar do Reúso em Serviços de Teste . . . 101
5.8 Considerações Finais . . . 103

6 Conclusão 105
6.1 Caracterização da Pesquisa Realizada . . . 105
6.2 Contribuições . . . 106
6.3 Dificuldades e Limitações . . . 107
6.4 Trabalhos Futuros . . . 107

Referências 123

Lista de Figuras

2.1 Processo de teste para detecção de defeitos 17

3.1 Relacionamento entre modelo de referência, padrão arquitetural, arquitetura de referência e arquitetura concreta 23
3.2 Interação de interessados e contextos entre arquiteturas concretas e de referência 26
3.3 Relacionamento entre atividades primárias, organizacionais e de apoio . . . 30
3.4 Visão geral da RefASSET 32
3.5 Representação do ProSA-RA 33
3.6 Papéis desempenhados por um serviço agregador 35
3.7 Estrutura genérica da interação entre serviços em SOA 36
3.8 Estrutura genérica de uma orquestração 37
3.9 Estrutura genérica de uma coreografia 38
3.10 Estrutura genérica de um ESB 39

4.1 Arquitetura da ferramenta *JaBUTi*Service 50
4.2 Arquitetura de referência S3 52
4.3 Arquitetura de referência SOAII 53
4.4 Arquitetura proposta por Eickelmann e Richardson (1996) 59
4.5 Mapeamento das funcionalidades específicas de ferramentas de teste presentes nas fontes de informação 60
4.6 Visão geral da RefTEST 62
4.7 Visão de módulos da RefTEST 63
4.8 Visão em tempo de execução da RefTEST 64
4.9 Visão de implantação da RefTEST 65
4.10 Mapeamento das funcionalidades identificadas nos grupos de fontes de informação 65
4.11 Visão geral da RefTEST-SOA 72
4.12 Visão de módulo da RefTEST-SOA 76
4.13 Visão em tempo de execução da RefTEST-SOA 78
4.14 Visão de implantação da RefTEST-SOA 79

5.1 Instanciação arquitetural da RefTEST-SOA para uma ferramenta de teste de mutação 85
5.2 Diagrama de classes simplificado do serviço MTPA 88
5.3 Diagrama de classes simplificado do serviço MTPR 90
5.4 Diagrama de classes simplificado do serviço MTC 92
5.5 Diagrama de classes simplificado do serviço TCM 94
5.6 Processo de negócio da ferramenta PascalMutants-Service 95
5.7 Menu de opções para a utilização da PascalMutants-Service 98
5.8 Mensagens apresentadas após a criação de um projeto de teste 99
5.9 Mensagens apresentadas após a inserção e execução de um caso de teste . . 100
5.10 Representação geral do reúso de serviços de teste segundo a RefTEST-SOA 102

Lista de Tabelas

4.1 Breve descrição de arquiteturas e ferramentas orientadas a serviço de teste, de verificação e de análise de software 49
4.2 Modelos e arquiteturas de referência orientados a serviço 51
4.3 Funcionalidades de sistemas orientados a serviço e seus conceitos relacionados 54
4.4 Breve descrição dos objetivos das arquiteturas de referência identificadas . 57
4.5 Conceitos relacionados a serviço abordados em arquiteturas de referência . 58
4.6 Funcionalidades específicas e conceitos do domínio de testes 61
4.7 Relação entre os conceitos de teste e requisitos arquiteturais da RefTEST-SOA 68
4.8 Relação entre os conceitos e requisitos arquiteturais referentes à SOA . . . 70

5.1 Equivalência entre as operações do serviço MTPA e as funcionalidades específicas de ferramentas de teste propostas pela RefTEST-SOA 87
5.2 Equivalência entre as operações do serviço MTPR e as funcionalidades específicas de ferramentas de teste propostas pela RefTEST-SOA 89
5.3 Operadores de mutação utilizados pelo serviço MTPR 90
5.4 Equivalência entre as operações do serviço MTC e as funcionalidades específicas de ferramentas de teste propostas pela RefTEST-SOA 91
5.5 Equivalência entre as operações do serviço TCM e as funcionalidades específicas de ferramentas de teste propostas pela RefTEST-SOA 93
5.6 Número de classes e LoCs utilizadas na implementação dos serviços 97
5.7 Informações sobre a execução dos casos de teste no programa Conversor . . 101
5.8 Análise da capacidade de reúso dos serviços de teste primários 101

Lista de Listagens

5.1 Código fonte do programa em teste 97

Lista de Abreviaturas e Siglas

ADL	*Architectural Description Language*
ATAM	*Architecture Trade-off Analysis Method*
ASSET	*A System to Select and Evaluate Test*
ERA	*E-contracting Reference Architecture*
ESB	*Enterprise Service Bus*
GFC	Grafo de Fluxo de Controle
GUI	*Graphical User Interface*
JaBUTi	*Java Bytecode Understanting and Testing*
MVC	*Model-View-Controller*
MTPA	*Mutation Test Pascal Artifact*
MTPR	*Mutation Test Pascal Requirement*
MTC	*Mutation Test Criteria*
OASIS	*Organization for the Advancement of Structured Information Standards*
POA	Programação Orientada a Aspecto
POKE-TOOL	*Potencial Uses Criteria Tool for Program Testing*
POO	Programação Orientada a Objeto
QoS	*Quality of Service*
RefASSET	*Reference Architecture for Software Engineering Environment*
RefTEST	*Reference Architecture for Testing Tools*
RefTEST-SOA	*Reference Architecture for Testing Tools based on SOA*
SAAM	*Software Architecture Analysis Method*
SLA	*Service Level Agreement*
SOA	*Service Oriented Architecture*
SOAP	*Simple Object Access Protocol*
TCM	*Test Case Management*
VV&T	Verificação, Validação e Teste
UDDI	*Universal, Discovery, Description and Integration*
UML	*Unified Modeling Language*
W3C	*World Wide Web Consortium*
WSBPEL	*Web Services Business Process Execution Language*
WSCDL	*Web Services Choreography Description Language*
WSDL	*Web Services Description Language*
XML	*Extensible Markup Language*

Capítulo

1

Introdução

1.1 Contextualização

Sistemas de software têm apresentado uma crescente importância nas últimas décadas, tendo desempenhado um papel fundamental em diversos ramos da sociedade. Em decorrência disso, a garantia da qualidade tem sido considerada como uma das principais atividades do processo de desenvolvimento de sistemas de software. Nesse contexto, o teste de software é uma das mais importantes atividades na garantia dessa qualidade e também, da confiabilidade dos sistemas sendo desenvolvidos (Harrold, 2000), tendo fundamental importância na identificação e eliminação de defeitos (Myers et al., 2004). Em virtude disso, diversas técnicas e critérios de teste vêm sendo propostos e utilizados (Budd, 1981; DeMillo et al., 1978; Maldonado, 1991; Rapps e Weyuker, 1982). Contudo, o teste de software sem o apoio automatizado de ferramentas constitui uma atividade onerosa, propensa a erros e que pode dispender muito tempo. A automatização dessa atividade é, portanto, bastante importante, contribuindo para melhorar a qualidade e a produtividade da atividade de teste, bem como aumentar a qualidade dos produtos de software (Harrold, 2000). Dessa forma, diversas ferramentas de teste têm sido desenvolvidas e utilizadas (Chaim, 1991; Delamaro, 1993; Gao et al., 1997; Hewlett-Packard, 2010; Vincenzi, 2004; Yano, 2004). Considerando a necessidade de se construir ferramentas de qualidade, esforços para o estabelecimento das arquiteturas de software dessas ferramentas são importantes.

Em uma outra perspectiva, arquiteturas de software têm recebido crescente atenção como um importante ramo da Engenharia de Software, assim como já apontado anteriormente por Garlan (2000). Arquiteturas de software desempenham um papel fundamental na determinação da qualidade de sistemas, uma vez que constituem a principal estrutura de qualquer sistema de software (Shaw e Clements, 2006; Wasserman, 1996). Decisões tomadas em nível arquitetural podem influenciar diretamente na realização dos objetivos de negócio, bem como no cumprimento dos requisitos funcionais e de qualidade (Angelov et al., 2008). A arquitetura de software é a estrutura (ou conjunto de estruturas) do sistema que compreende os elementos de software, as propriedades externamente visíveis desses elementos e as relações entre eles (Bass et al., 2003).

Um tipo especial de arquitetura, a arquitetura de referência, tem emergido como um elemento que agrega o conhecimento de um domínio específico por meio de módulos e suas relações (Bass et al., 2003). Arquiteturas de referência promovem o reúso de experiências de projeto, por meio do sólido e consolidado entendimento de um domínio específico. Essas arquiteturas podem ser consideradas como repositórios de conhecimento que visam apoiar o desenvolvimento de sistemas (Bass et al., 2003). Considerando sua relevância, arquiteturas de referência para diferentes domínios podem ser encontradas (Angelov e Grefen, 2008; Arsanjani et al., 2007; Costagliola et al., 2008; Eickelmann e Richardson, 1996; Fioravanti et al., 2010; Hemalatha et al., 2008; Nakagawa et al., 2007; Peristeras et al., 2009). Além disso, arquiteturas de referência para o domínio de teste de software também podem ser encontradas (Eickelmann e Richardson, 1996; Nakagawa et al., 2007). Em especial, a RefTEST (*Reference Architecture for Testing Tools*)(Nakagawa et al., 2007) tem contribuído para o desenvolvimento de ferramentas de teste de software (Ferrari et al., 2010; Nakagawa et al., 2009b).

Ainda no contexto de arquitetura de software, um estilo arquitetural que tem despertado bastante interesse é a arquitetura orientada a serviço (do inglês, *Service-Oriented Architecture* - SOA) (Josuttis, 2007; Papazoglou e Heuvel, 2007). Sistemas de software desenvolvidos a partir desse estilo arquitetural possuem como principais características a independência de linguagem de programação e a flexibilidade quanto à plataforma, uma vez que funcionalidades oferecidas por serviços são definidas por meio de linguagens de descrição padronizadas (Papazoglou et al., 2008). Dessa forma, sistemas orientados a serviços podem ser construídos pela composição de funcionalidade simples disponibilizadas por outros serviços, formando aplicações mais completas e, segundo Papazoglou e Heuvel (2007), de forma mais produtiva. Todavia, segundo Arsanjani et al. (2007), a construção de aplicações baseadas na SOA é uma tarefa complexa. Para que diferentes aplicações, muitas vezes projetadas por equipes distintas, possam cooperar de maneira satisfatória, é necessário que arquitetos desenvolvam abordagens consistentes de produção, utilizando

notações bem definidas e que ilustrem as capacidades de cada aplicação (Arsanjani et al., 2007). Essas abordagens de desenvolvimento têm sido representadas em diversos domínios por meio do uso de arquiteturas de referência orientadas a serviço (Costagliola et al., 2006; Hemalatha et al., 2008; Oliveira et al., 2010; Peristeras et al., 2009). No contexto deste trabalho, entende-se por arquiteturas de referência orientadas a serviço aquelas que têm como base a SOA. A utilização de arquiteturas de referência orientadas a serviço pode proporcionar benefícios quanto à interoperabilidade, compreensão do domínio, estabelecimento de vocabulário comum, reúso arquitetural, consistência de representação e diminuição do tempo gasto no desenvolvimento (*time-to-market*) dos sistemas daquele domínio (Costagliola et al., 2008; Ramanathan et al., 2008).

1.2 Motivação

Tendo em vista a importância da automatização da atividade de teste de software, ferramentas de teste têm sido desenvolvidas e utilizadas. Entretanto, muitas dessas ferramentas de teste são produzidas de maneira isolada, sobre arquiteturas e infraestruturas próprias (Nakagawa, 2006; Nakagawa et al., 2007). Como consequência, dificuldades quanto à integração, evolução, manutenção e reutilização dessas ferramentas são muito comuns (Nakagawa et al., 2007). Nesse contexto, a utilização do estilo arquitetural SOA na automatização da atividade de teste parece apresentar-se com uma iniciativa promissora, podendo viabilizar a utilização de diferentes ferramentas de teste de forma integrada. A integração de ferramentas de teste possui especial importância, uma vez que a utilização complementar de técnicas e critérios de teste de software é uma prática fortemente recomendada (Myers et al., 2004). Além disso, a execução de técnicas de teste com alto custo computacional ou que possuam grandes quantidades de dados de teste poderiam ser executadas por serviços distribuídos, reduzindo, de alguma forma, o tempo da execução do teste. O uso da SOA no domínio de teste poderia trazer vantagens quanto à disponibilidade e à facilidade de uso das ferramentas desenvolvidas, podendo essas serem ainda utilizadas por aplicações *desktop*, web ou por outros serviços, inclusive sem a necessidade da instalação dessas ferramentas em uma máquina local.

Em virtude disso, ferramentas de teste oferecidas na forma de serviços web já podem ser encontradas (Bartolini et al., 2008; Eler et al., 2010, 2009). Além disso, pode ser encontrado, inclusive, um registro de serviços projetado especificamente para a publicação e descoberta de serviços relacionados ao teste de software (Gondim, 2010). Em outras palavras, observa-se que a comunidade começa a entender a importância da SOA também para a disponibilização de ferramentas de teste. No entanto, o desenvolvimento de ferramentas de teste segundo o estilo arquitetural SOA ainda é recente. Dessa forma, esforços

destinados ao estudo e o estabelecimento de arquiteturas para essas ferramentas possuem grande importância.

Com o intuito de auxiliar o desenvolvimento de ferramentas de teste de software, esforços em nível arquitetural podem ser encontrados (Eickelmann e Richardson, 1996; Nakagawa et al., 2007; Yang et al., 2002, 1999). Em especial, arquiteturas de referência possuem grande importância, uma vez que podem ser utilizadas como guia para o projeto de arquiteturas concretas (Muller, 2008). Para o domínio de Teste de Software, duas são as arquiteturas de referência encontradas na literatura: a proposta por Eickelmann e Richardson (1996) e a RefTEST (Nakagawa et al., 2007). Eickelmann e Richardson (1996) propõem a primeira arquitetura de referência para o domínio de teste, que representa as funcionalidades de ferramentas de teste de software utilizando um alto nível de abstração. A arquitetura proposta por Nakagawa et al. (2007) representa um repositório de conhecimento do domínio de teste e foi desenvolvida com base em arquiteturas e ferramentas existentes, documentos, padrões e o acúmulo de experiência relacionada à atividade de teste. A RefTEST provê direções para o desenvolvimento de ferramentas de teste disponíveis na forma *stand-alone* ou por meio da plataforma web. Contudo, tanto a RefTEST quanto a arquitetura de referência proposta por Eickelmann e Richardson (1996) não são adequadas ao desenvolvimento de ferramentas de teste que tenham como base a SOA. Dessa forma, existe uma carência de arquiteturas de referência que apoiem o desenvolvimento de ferramentas de teste orientadas a serviço.

1.3 Objetivo

Motivado pela falta de arquiteturas de referência para o domínio de teste de software que tenham como base a SOA, o principal objetivo deste projeto é o estabelecimento de uma arquitetura de referência orientada a serviço, denominada RefTEST-SOA (*Reference Architecture for Testing Tools Based on SOA*). Tal arquitetura engloba a experiência e o conhecimento do domínio e os conceitos e particularidades da SOA, tendo como objetivo auxiliar o desenvolvimento de ferramentas de teste orientadas a serviço. Visa-se assim, contribuir para a área de Teste de Software com a disponibilização de uma arquitetura que facilite o desenvolvimento, o reúso, a manutenibilidade e a evolução de ferramentas de teste orientadas a serviço.

Com o objetivo de prover evidências da aplicabilidade da arquitetura de referência proposta, bem como indícios do aumento da capacidade de reúso e integração das ferramentas desenvolvidas, um estudo de caso é realizado. Com esse estudo de caso, visa-se ilustrar o processo de instanciação da RefTEST-SOA, bem como o projeto e implementação de uma ferramenta de teste de software orientada a serviço. Objetiva-se, com isso, ter

os primeiros indícios do aumento da capacidade de reúso, tanto das ferramentas de teste desenvolvidas e baseadas na RefTEST-SOA quanto do conhecimento por ela representado.

1.4 Organização

Neste capítulo foi apresentado o contexto no qual este projeto de mestrado se insere e a motivação que levou ao seu desenvolvimento. O restante deste trabalho está organizado da seguinte forma. No Capítulo 2 são apresentados conceitos relacionados à atividade de teste de software. No Capítulo 3 são discutidos conceitos e terminologias relacionadas à arquitetura de software, dando especial atenção às arquiteturas de referência e à SOA. Os detalhes acerca do estabelecimento da RefTEST-SOA são apresentados no Capítulo 4. No Capítulo 5 é apresentado um estudo de caso que ilustra em detalhes o desenvolvimento de uma ferramenta de teste orientada a serviço, com base na RefTEST-SOA. Por fim, no Capítulo 6 são discutidos os principais resultados deste trabalho, as contribuições, perspectivas de trabalhos futuros.

Capítulo

2

Teste de Software: Uma Visão Geral

2.1 Considerações Iniciais

O teste de software é considerado uma importante atividade do processo de desenvolvimento de sistemas de software, uma vez que essa contribui para o aumento da confiança nos produtos desenvolvidos, por meio da redução de defeitos (Harrold, 2000). Em virtude disso, uma diversidade de técnicas e critérios de teste vem sendo proposta e utilizada. Contudo, a aplicação efetiva dessas técnicas e critérios de teste é impraticável sem o apoio automatizado de ferramentas de teste de software (Delamaro et al., 2007). Ferramentas de teste aprimoram a qualidade e a produtividade da atividade de teste, sendo utilizadas tanto na academia quanto na indústria.

Na Seção 2.2 são apresentados conceitos básicos relacionados à atividade de teste. Na Seção 2.3 são apresentadas as principais técnicas e critérios de teste encontrados na literatura. A importância da automatização da atividade de teste, bem como as ferramentas de apoio que vêm sendo desenvolvidas e utilizadas são discutidas na Seção 2.4. Na Seção 2.5, processos de teste de software são apresentados. Por fim, na Seção 2.6, são discutidas as considerações finais deste capítulo.

2.2 Terminologia e Conceitos Básicos

A garantia da qualidade é considerada como uma das principais atividades do processo de desenvolvimento de sistemas de software. Para que um sistema atinja um nível aceitável de qualidade, atividades de verificação, validação e teste de software (VV&T) são fundamentais (Maldonado, 1991). A verificação consiste de um conjunto de atividades que possuem o objetivo de assegurar que o processo de desenvolvimento esteja sendo conduzido de maneira correta (Andriole, 1986). A validação, por sua vez, visa garantir que o produto desenvolvido atenda às suas especificações (Andriole, 1986). A atividade de teste objetiva a corretitude de um determinado programa, provendo evidências da confiabilidade do uso do mesmo (Harrold, 2000).

A atividade de teste, segundo Myers et al. (2004) e Pressman (2005), contribui para evidenciar que o software desenvolvido desempenha suas funções de acordo com o que lhe foi especificado. Apesar de ser impossível por meio de testes garantir que um software esteja correto, a atividade de teste aplicada de maneira criteriosa e formal contribui para o aumento da confiança, fornecendo indicativos de que o software desempenha suas funções de forma aceitável e com o mínimo de qualidade (Harrold, 2000).

Nesse contexto, torna-se importante distinguir os termos defeito, engano, erro e falha. Segundo o padrão IEEE 610.12 (IEEE, 1990), um *defeito* é uma divergência entre o software desenvolvido e o software supostamente correto, e é causado por algum erro humano, denominado *engano* (do inglês, *mistake*), durante a realização de algum processo, método ou atividade, como por exemplo, a codificação. O *erro*, por sua vez, ocorre quando um defeito faz com que o produto de software em questão entre em um estado inconsistente, diferente do estado correto. Já a *falha* é uma manifestação observável ocasionada em virtude da presença de um *erro*.

No teste de software, o conceito de *domínio de entrada* de um programa está relacionado ao conjunto de possíveis valores que podem ser utilizados em sua execução. Analogamente, pode-se definir como *domínio de saída* o conjunto de todos os resultados produzidos pela execução do programa em teste. Um *dado de teste*, por sua vez, consiste de um elemento do domínio de entrada de um programa. Já um *caso de teste* é um par formado por um dado de teste e o resultado esperado de sua execução. O conjunto de todos os casos de teste utilizados durante uma determinada atividade de teste é chamado de *conjunto de casos de teste*.

A atividade de teste, segundo Delamaro et al. (2007) e Pressman (2005), pode ser dividida em três fases, sendo essas: *teste de unidade*, *teste de integração* e *teste de sistema*.

Teste de Unidade: tem como objetivo encontrar defeitos de lógica e de implementação, inseridos na menor unidade de um software, buscando garantir que essa funcione adequadamente (Myers et al., 2004). Essa unidade pode ser, por exemplo, um método ou uma classe quando se utiliza o paradigma orientado a objetos, ou uma função ou procedimento quando o paradigma em questão é o procedural;

Teste de Integração: é o teste utilizado na integração dos módulos de um software e tem por objetivo revelar defeitos relacionados às suas interfaces (Maldonado, 1991). Nessa fase, é testada a integração da estrutura projetada por meio da composição dos módulos já testados isoladamente durante o teste de unidade. Uma vez que, no teste de integração, os testes podem envolver partes do software que ainda não estão completamente implementadas, *drivers* e *stubs* precisam ser utilizados. O *driver* consiste de um módulo responsável por emular chamadas para a unidade em teste. O *stub*, por sua vez, é um módulo que simula o comportamento de uma unidade invocada (Pressman, 2005); e

Teste de Sistema: após a integração das partes do sistema, é realizado o teste de sistema. O objetivo desse teste é avaliar o sistema como um todo, observando se as funcionalidades descritas na especificação foram corretamente implementadas. Características ligadas ao desempenho, segurança e demais requisitos não-funcionais são analisadas durante o teste de sistema (Pressman, 2005).

Além das três fases apresentadas, uma atividade denominada teste de regressão pode ser utilizada. O teste de regressão, ao contrário das demais atividades, não é utilizado durante o desenvolvimento do produto de software, mas sim durante a manutenção do software (Delamaro et al., 2007). Quando modificações são realizadas, novos defeitos podem ser introduzidos. Dessa forma, testes que evidenciem se as modificações realizadas foram corretamente implementadas são necessários (Delamaro et al., 2007). Independentemente da fase de teste, segundo Myers et al. (2004) e Pressman (2005), os testes são executados em quatro etapas bem definidas: planejamento de testes, projeto dos casos de teste, execução do teste e coleta e avaliação dos resultados. Dentre essas, Myers et al. (2004) define a atividade de projeto de casos de teste como a mais importante.

2.3 Técnicas e Critérios de Teste

O teste completo de um software é uma tarefa impraticável devido ao seu alto custo computacional e financeiro (Myers et al., 2004). Em virtude disso, a procura por formas de selecionar um subconjunto reduzido de casos de teste para execução de um programa, que possua alta probabilidade de revelar a presença de defeitos, constitui uma tarefa

de grande importância (Myers et al., 2004). Nesse contexto, existem duas maneiras de selecionar casos de teste para cada um dos subconjuntos de teste: o *teste randômico* e o *teste de subdomínios*.

O *teste randômico* tem seu conjunto de casos de teste composto por elementos aleatórios pertencentes ao domínio de entrada. Visto que os erros não são produzidos de forma homogênea com relação ao conjunto de entrada, ou seja, são mais frequentes para uma determinada faixa desses valores, o teste randômico normalmente não é eficiente se utilizado puramente. Desse modo, uma possível abordagem é considerar o *teste de subdomínios*, que consiste da procura de subdomínios a serem utilizados e os dados de entrada pertencentes a cada um deles. Para a seleção de subdomínios são estabelecidas "regras" denominadas critérios de teste. Por meio dos critérios de teste, são selecionados os casos de teste que devem pertencer ou não a cada subdomínio (Delamaro et al., 2007). De modo geral, são definidos requisitos de teste que, quando satisfeitos, determinam a inclusão de casos de teste em um subdomínio. Ou seja, se um caso de teste satisfaz um requisito pertencente a um critério de teste, então esse caso de teste pertence ao subdomínio de casos de teste adequados a esse critério.

De um modo geral, podem ser identificadas três técnicas de teste nos quais critérios de teste se inserem (Delamaro et al., 2007): a funcional, a estrutural e a baseada em defeitos. O que diferencia cada uma dessas técnicas é o tipo de informação utilizada para determinar cada um dos subdomínios. Além disso, cada técnica de teste possui vantagens, limitações e seu contexto de aplicação. As seções a seguir detalham brevemente as características de cada uma dessas técnicas.

2.3.1 Teste Funcional

No teste funcional, também chamado de teste caixa-preta, os critérios e requisitos de teste são construídos a partir da especificação do sistema, ou seja, o teste é avaliado segundo o ponto de vista do usuário (Delamaro et al., 2007). Assim, consideram-se apenas as entradas, saídas e estado do programa, não sendo necessário ao testador o acesso ao código fonte do software (Myers et al., 2004). O principal objetivo da aplicação do teste funcional é a localização de discrepâncias entre o comportamento real de parte de um sistema e sua especificação.

Em princípio, a ideia do teste funcional é detectar todos os defeitos, submetendo o programa a todas as entradas possíveis (Delamaro et al., 2007). Contudo, o domínio de entrada pode ser muito grande ou até infinito, tornando o teste inviável (Myers et al., 2004). Em virtude disso, critérios de teste são utilizados para a seleção de subconjuntos de casos de teste que possuam grande possibilidade de revelar defeitos. Dentre os critérios mais conhecidos, pode-se citar o Particionamento em Classes de Equivalência, a Análise

do Valor Limite e o teste por meio de Grafos Causa-efeito (Delamaro et al., 2007; Myers et al., 2004).

Particionamento em Classes de Equivalência: nesse critério, o domínio de entrada do programa é dividido em classes com características similares, chamadas de classes de equivalência, que são utilizadas para derivar os requisitos de teste (Pressman, 2005). A divisão das classes de equivalência deve ser feita de maneira que, defeitos revelados por um dado de teste pertencente a uma classe devem ser revelados por todos os demais dados de teste pertencentes à mesma classe. Essa divisão é feita particionando-se o domínio de entrada de um programa em classes de equivalência válidas e inválidas, de acordo com a especificação (Myers et al., 2004). Em seguida, um conjunto de casos de teste deve ser criado de forma a representar os elementos pertencentes a cada uma das classes de equivalência. Dessa forma, segundo Myers et al. (2004), é possível diminuir a quantidade de casos de teste sem afetar a qualidade do teste executado;

Análise do Valor Limite: é um critério complementar ao critério Particionamento em Classes de Equivalência. Nesse critério, os limites associados às condições de entrada são exercitados de maneira mais rigorosa, de forma que, ao invés de se selecionar qualquer elemento de uma classe, casos de teste são escolhidos nas fronteiras inferior e superior de cada classe (Myers et al., 2004; Pressman, 2005). Segundo Myers et al. (2004), casos de testes que exploram condições limites possuem maior probabilidade de encontrar defeitos; e

Grafo de Causa-efeito: tem por objetivo explorar as fraquezas dos critérios Particionamento em Classes de Equivalência e Análise de Valor Limite, uma vez que não exploram combinações entre os dados de entrada, não permitindo a cobertura das classes de equivalência de maneira mais eficiente (Delamaro et al., 2007). O critério Grafo de Causa-efeito é utilizado para auxiliar na definição de conjuntos de casos de teste que explorem ambiguidades e incompletudes nas especificações (Myers et al., 2004). No critério Grafo de Causa-efeito, um conjunto de causas e de possíveis efeitos é identificado e representado por meio de um grafo. Nesse grafo, as causas correspondem às entradas ou estímulos que provoquem reações do sistema. Os efeitos correspondem as saídas, mudanças de estado ou resultado observáveis (Myers et al., 2004).

2.3.2 Teste Estrutural

No teste estrutural, também conhecido como teste caixa-branca, os casos de teste são derivados a partir da estrutura interna do programa em teste, ou seja, do código fonte

(Myers et al., 2004; Pressman, 2005). Normalmente, essa técnica é aplicada com o auxílio de uma representação gráfica do programa, sendo essa representação denominada Grafo de Fluxo de Controle (GFC) (Myers et al., 2004). Nesses grafos, os vértices representam blocos de comando executados de forma indivisível e arestas representam os possíveis fluxos de controle entre esses blocos (Delamaro et al., 2007; Maldonado, 1991; Myers et al., 2004). Em cada bloco de comando, uma vez que um comando é executado, todos os demais comandos são executados sequencialmente, não existindo desvios de execução dentro do mesmo.

No teste estrutural, os critérios de teste mais conhecidos são os baseados em Fluxo de Controle (Zhu et al., 1997) e em Fluxo de Dados (Maldonado, 1991; Rapps e Weyuker, 1982). Nos critérios baseados em Fluxo de Controle são utilizadas apenas informações referentes à execução do programa, como por exemplo, comandos e desvios. Segundo Myers et al. (2004) e Zhu et al. (1997), os critérios baseados em Fluxo de Controle mais conhecidos são:

Todos-Nós: esse critério requer que a execução do programa passe, ao menos uma vez, em cada bloco de comando, ou seja, todos os vértices do CFG devem ser executados pelo menos uma vez;

Todas-Arestas: esse critério requer que cada desvio seja exercitado pelo menos uma vez, ou seja, a execução do programa deve passar por todos os desvios de fluxo controle; e

Todos-Caminhos: nesse critério é necessário que todos os caminhos possíveis do CFG sejam executados.

Os critérios baseados em Fluxo de Dados requerem que os caminhos que envolvam definições e usos de variáveis sejam testados. Esses critérios são baseados na investigação dos modos nos quais os valores são associados às variáveis do programa e como essas associações afetam sua execução (Zhu et al., 1997). Existem dois tipos de usos possíveis de variáveis (Rapps e Weyuker, 1982): predicativo e computacional. O uso predicativo, ou *p-uso*, ocorre quando uma variável é utilizada como parte de uma condição de desvio de fluxo de execução. Já o uso computacional, ou *c-uso*, ocorre quando uma variável é utilizada no cálculo do valor de outra variável (Rapps e Weyuker, 1982). Os critérios de teste baseados em Fluxo de Dados possuem como justificativa de existência o fato que, mesmo para programas pequenos, o teste baseado em Fluxo de Controle não é eficaz para detectar defeitos considerados triviais (Delamaro et al., 2007). A seguir, são apresentados exemplos de critérios de teste baseados em Fluxo de Dados:

Todas-Definições: esse critério requer que cada definição de variável seja exercitada ao menos uma vez, seja por um *c-uso* ou um *p-uso* (Rapps e Weyuker, 1982). Para isso, deve-se criar um conjunto de casos de teste tal que sejam exercitados caminhos entre cada variável e um dos seus usos, antes que essa variável seja redefinida;

Todos-Usos: requer que todas as associações entre a definição e o uso de uma variável sejam executadas por pelo menos um caminho no qual ela não foi redefinida, ou seja, um caminho livre de definição (Rapps e Weyuker, 1982). Tem-se: Todos-p-Usos, Todos-c-usos, Todos-c-Usos/Alguns-p-Usos como exemplos de variações do critério Todos-Usos (Rapps e Weyuker, 1982); e

Todos-Du-Caminhos: esse critério requer o exercício de todas as associações entre uma definição e seus *c-usos* ou *p-usos*, por todos os caminhos livre de definição e de laço que cubram essa associação (Rapps e Weyuker, 1982).

Todos-Potenciais-Usos: estabelece que os caminhos entre cada variável definida em cada nó do grafo e todos os nós alcançáveis desse grafo, antes que a variável seja redefinida, sejam exercitados (Maldonado, 1991).

2.3.3 Teste Baseado em Defeitos

A técnica de teste baseada em defeitos utiliza informações sobre os tipos de enganos frequentemente cometidos por programadores no processo de desenvolvimento de software. Dentre os critérios mais conhecidos, pode-se citar a Semeadura de Erros (do inglês, *Error Seeding*) (Budd, 1981) e a Análise de Mutantes (do inglês, *Mutant Analysis*)(Budd, 1981; DeMillo et al., 1978).

De forma simplificada, a ideia principal do critério de Análise de Mutantes é evidenciar que uma determinada parte de programa está correta demonstrando que suas possíveis variações não estão (DeMillo et al., 1987). Para isso, DeMillo et al. (1978) parte da "hipótese do programador competente", que assume que programadores experientes codificam programas corretos ou muito próximos do correto, sendo os defeitos introduzidos por pequenos desvios sintáticos. Para esse critério, é adotada também a hipótese do "efeito de acoplamento" (DeMillo et al., 1978), que assume que erros complexos são resultado da combinação de erros mais simples. Dessa maneira, se um conjunto de casos de teste é capaz de revelar um defeito simples, também pode identificar defeitos mais complexos.

No teste de mutação, a partir do programa em teste P, é criado um conjunto M de programas com pequenos desvios sintáticos, denominados mutantes (Agrawal et al., 1989). Em seguida, é criado um conjunto de casos de teste T que tem por objetivo mostrar que o comportamento de um programa mutante m, $(m \in M)$, difere do programa original para

um dado caso de teste, ou seja, $m(t) \neq P(t)$, $(t \in T)$. Caso a saída de um mutante seja diferente da saída esperada, o mutante é considerado como morto. No entanto, é preciso ressaltar que algumas modificações podem produzir programas equivalentes ao original, tal que não é possível encontrar um caso de teste que, para o conjunto de entradas desse programa, proporcione uma saída diferente da saída esperada (Delamaro et al., 2007). Em outras palavras, um programa mutante m é equivalente ao programa original P, se $\forall t \in T$: $P(t) = m(t)$. Nesse caso, a determinação de equivalência entre programas requer a intervenção do testador, pois é computacionalmente indecidível.

Embora o teste de mutação seja eficiente na detecção de defeitos em programas, sendo considerado um dos mais fortes critérios de teste presentes na literatura (Li et al., 2009), uma importante consideração a ser feita acerca desse critério é que, mesmo para programas pequenos, a quantidade de mutantes gerados pode ser muito grande ou até mesmo infinita (Morell, 1990), ocasionando um tempo de execução muito alto. Em virtude do grande número de mutantes produzidos, é de suma importância a utilização de ferramentas que automatizem a aplicação desse critério.

2.4 Automatização do Teste de Software

A aplicação prática de critérios de teste de software está fortemente relacionada à sua automatização. Dessa forma, a utilização de critérios de teste sem o apoio automatizado de ferramentas de teste de software é, possivelmente, uma atividade propensa a erros e restrita a programas de tamanho reduzido (Vincenzi, 2004). Segundo Harrold (2000), a utilização de ferramentas de teste auxilia no desenvolvimento de sistemas de maior qualidade. Ferramentas de teste podem ser classificadas de duas formas (Howden, 1987): ferramentas de teste de programas e ferramentas de teste de especificação. As ferramentas de teste de programa dão apoio à aplicação de critérios para o teste do código fonte da aplicação. Já as ferramentas de teste de especificação utilizam como entrada especificações de programas escritos em técnicas de modelagem, tais como Statecharts (Harel, 1987), Redes de Petri (Peterson, 1977) e Máquinas de Estados Finitas (Gill, 1962).

Em virtude dos benefícios proporcionados pela automatização da atividade de teste de software, diversas ferramentas vêm sendo desenvolvidas e utilizadas. Dentre essas, encontram-se: a SCORE (Chusho, 1987), que apoia o teste de arestas para programas escritos em Pascal; o conjunto de ferramentas xSud (Agrawal et al., 1998; Technologies, 1997), destinadas ao entendimento, análise e teste de programas desenvolvidos na linguagem C/C++; a ferramenta ASSET (*A System to Select and Evaluate Test*) (Frankl e Weyuker, 1988), uma das primeiras ferramentas de teste desenvolvidas e que apoia a apli-

cação de critérios baseados em Fluxo de Dados e a ferramenta HP QuickTest Professional (Hewlett-Packard, 2010), que oferece apoio ao teste funcional e de regressão.

Recentemente, ferramentas de teste para aplicações implementadas segundo o paradigma orientado a objeto têm sido também desenvolvidas. A ferramenta proposta por Ma et al. (2006), denominada MuJava, destina-se à automatização do critério de Análise de Mutantes para programas implementados na linguagem Java. Por sua vez, o *framework* JUnit (Gamma e Beck, 2010) apoia o teste de programas em nível unitário, por meio do uso de assertivas que verificam se o programa executou conforme o esperado. Apesar de não implementar nenhum critério de teste, o JUnit pode ser incorporado a outras ferramentas, uma vez que esse *framework* é disponibilizado na forma de um software livre.

Além dessas ferramentas, outras desenvolvidas pelo grupo de pesquisa em teste de software do ICMC/USP, em parceria com outros grupos de pesquisa, podem ser citadas, sendo essas[1]:

- **POKE-TOOL (*Potencial Uses Criteria Tool for Program Testing*):** Desenvolvida pela FEEC/UNICAMP, em parceria com o ICMC/USP, essa ferramenta apoia a aplicação de critérios baseados em Fluxo de Controle e em Fluxo de Dados. Inicialmente desenvolvida para o teste de programas escritos na linguagem C (Chaim, 1991), foi estendida para diversas linguagens, tais como COBOL (Leitão Jr., 1992), Fortran (Fonseca, 1993) e Clipper (Borges et al., 1995);

- **Proteum:** A família de ferramentas Proteum é destinada à automatização do critério Análise de Mutantes. Inicialmente, a ferramenta Proteum (Delamaro, 1993) foi desenvolvida para automatizar a Análise de Mutantes em nível unitário, para a linguagem C. A partir da Proteum, foram desenvolvidas outras ferramentas para o teste de programa, tais como: Proteum-IM (Delamaro, 1997), Proteum/IM 2.0 (Delamaro et al., 2000) e Proteum/SML (Yano, 2004). Além disso, também foram desenvolvidas as ferramentas Proteum-RS/FSM (Fabbri et al., 1994), Proteum-RS/ST (Fabbri et al., 1999; Sugeta et al., 2001) e a Proteum-RS/PN (Simão et al., 2000), voltadas ao teste de especificação; e

- ***JaBUTi*(*Java Bytecode Understanting and Testing*):** Essa ferramenta apoia a aplicação da técnica estrutural em programas codificados em Java. A *JaBUTi*, proposta por Vincenzi (2004), apoia diferentes critérios de teste, tais como: Todos-Nós, Todas-Arestas, Todos-Usos e Todos-Potenciais-Usos. Além disso, essa ferramenta provê um conjunto de métricas estáticas e funcionalidades para a análise de cobertura.

[1]Relações mais completas dos trabalhos desenvolvidos pelo grupo de pesquisa do ICMC/USP, e seus detalhes, podem ser encontrados em trabalhos como os de Nakagawa (2006), Ferrari (2005) e Vincenzi (2004).

É importante notar, entretanto, que grande parte das ferramentas de teste de software representam iniciativas isoladas de desenvolvimento, possuindo arquiteturas e estruturas próprias, projetadas sem considerar características como manutenibilidade e reúso (Nakagawa, 2006). Como consequência, essas ferramentas podem apresentar dificuldades quanto à integração e à evolução (Nakagawa et al., 2007). Nesse contexto, a atenção às arquiteturas de software dessas ferramentas pode influenciar positivamente no seu entendimento e evolução. Em virtude disso, estudos sobre arquiteturas de software para ferramentas de teste podem ser encontrados (Eickelmann e Richardson, 1996; Nakagawa et al., 2007; Richardson, 1994; Yang et al., 2002).

2.5 Processo de Teste de Software

Um processo de software é um conjunto de atividades e resultados associados que levam à confecção de um produto de software (Sommerville, 2003). De forma mais detalhada, segundo Fuggetta (2000), um processo de software consiste de um conjunto de políticas, estruturas organizacionais, tecnologias, procedimentos e artefatos necessários à concepção, desenvolvimento, entrega e manutenção de um produto de software. Um processo deve ser responsável por nortear pessoas quanto a divisão e coordenação de trabalhos, garantindo uma comunicação eficiente. Segundo Cugola e Ghezzi (1998), processos proporcionam uniformidade de comportamento ao longo de diferentes projetos, reduzindo custos e o tempo de entrega de produtos.

Em um contexto mais específico, um processo de teste de software representa uma estruturação de etapas, atividades, artefatos, papéis e responsabilidades que buscam a padronização e o controle de projetos de teste (Sommerville, 2003). No modelo de processo de teste definido por Sommerville (2003), apresentado na Figura 2.1, são ilustradas as relações entre as atividades de teste e os artefatos produzidos. As atividades consideradas durante o processo de teste software proposto são: projeto de casos de teste, preparação de casos de teste, execução de casos de teste e comparação de resultados. Dentre os artefatos, os casos de teste representam especificações de entrada para o teste e sua respectiva saída esperada. Os dados de teste são entradas que foram criadas para testar o sistema. Com base na execução dos dados de teste são produzidos resultados que, por sua vez, são comparados com os casos de teste.

Em virtude da importância desempenhada por processos na atividade de teste de software, esforços relacionados a essa área de pesquisa podem ser encontrados (Cangussu et al., 2002, 2003; Vincenzi et al., 2006). No contexto de teste baseado em defeitos, Vincenzi et al. (2006) propõe o Muta-Pro, um processo que visa sistematizar a aplicação do critério Análise de Mutantes. O processo proposto é similar ao de Sommerville (2003),

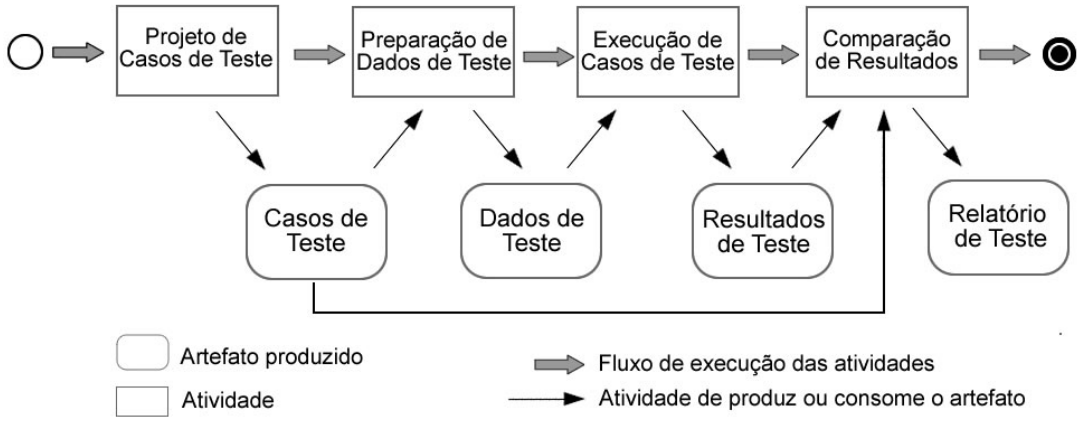

Figura 2.1: Processo de teste para detecção de defeitos (adaptado de Sommerville (2003))

porém, um pouco mais detalhado quanto a preparação e execução dos dados de teste. No Muta-Pro, a atividade preparação dos dados de teste subdivide-se em: tratamento do artefato de teste e na geração dos mutantes. Já a atividade de execução do teste é dividida na execução do programa original, seguida da execução dos mutantes. De modo geral, as ferramentas de teste para o critério Análise de Mutantes, como as de Delamaro (1993) e Simão et al. (2000), apresentam as funcionalidades descritas pelo processo de Vincenzi et al. (2006).

2.6 Considerações Finais

No desenvolvimento de sistemas de software, a garantia da qualidade é considerada parte importante do processo de software. As atividades de verificação, validação e teste contribuem significantemente para a garantia dessa qualidade. Em especial, a atividade de teste destaca-se por prover indicativos de confiabilidade dos sistemas. Em virtude disso, diversas técnicas e critérios de teste têm sido propostos e utilizados, sendo que cada um desses possui vantagens, limitações e seu contexto de aplicação. Para viabilizar o uso dessas técnicas e critérios, ferramentas de teste vêm sendo desenvolvidas. Contudo, grande parte das ferramentas encontradas na literatura foi desenvolvida de forma isolada, apresentando estruturas próprias, sem se preocupar com características como evolução, manutenibilidade e reúso. Para aprimorar o desenvolvimento de ferramentas de teste, arquiteturas de software e arquiteturas de referência para o domínio de teste vêm sendo propostas. No próximo capítulo são apresentados os conceitos relacionados à arquitetura de software, à arquitetura de referência e à SOA.

Capítulo

3

Arquitetura de Software

3.1 Considerações Iniciais

Em virtude da crescente complexidade dos sistemas de software, abordagens sistemáticas e eficientes de estruturação e desenvolvimento têm sido necessárias. Métodos e processos de desenvolvimento, convenções de nome, gerência de configuração e arquiteturas de software desempenham importante papel no desenvolvimento de sistemas e sua adequada estruturação (Bass et al., 2003; Land, 2002). Em especial, arquiteturas de software são consideradas parte fundamental do projeto de sistemas (Clements et al., 2002; Land, 2002). Segundo Bass et al. (2003), arquiteturas de software são uma das principais responsáveis pela determinação da qualidade de sistemas, uma vez que essas contemplam um conjunto de boas práticas de desenvolvimento, pelo uso de modelos, arquiteturas de referência, padrões de projeto e estilos arquiteturais.

Nesse contexto, arquiteturas de referência têm-se destacado como um tipo especial de arquitetura, provendo direções para a especificação de arquitetura concretas de sistemas de um dado domínio de aplicação. Arquitetura de referência consiste de um conjunto de abstrações de informações de um domínio, estruturadas de forma a auxiliar no desenvolvimento de um conjunto de sistemas (Bass et al., 2003).

Paralelo a isso, a arquitetura orientada a serviço vem se destacando como um estilo arquitetural que provê diversas vantagens quanto ao reúso, produtividade e evolução de aplicações (Josuttis, 2007; Papazoglou e Heuvel, 2007). Sistemas de software desenvolvidos

com base nesse estilo arquitetural possuem como principais características a independência de linguagem de programação e a flexibilidade quanto à plataforma, uma vez que as funcionalidades oferecidas por serviços são definidas por meio de linguagens de descrição padronizadas (Papazoglou et al., 2008). Além disso, sistemas orientados a serviços podem ser construídos pela composição de serviços mais simples, formando aplicações completas e, segundo Papazoglou e Heuvel (2007), de forma produtiva.

Na Seção 3.2 são apresentados conceitos e terminologias relacionados à arquitetura de software. Na Seção 3.3, arquiteturas de referência são apresentadas, dando especial atenção às desenvolvidas para o domínio de teste de software. Os conceitos relacionados à SOA, serviços web e arquitetura de referência orientada a serviço são apresentados na Seção 3.4. Por fim, na Seção 3.5 são apresentadas as considerações finais deste capítulo.

3.2 Arquitetura de Software

A área de Arquitetura de Software emergiu da crescente preocupação em compreender a organização de sistemas de software. A primeira referência ao termo "Arquitetura de Software" surgiu em 1969 (Kruchten et al., 2006); contudo, até o final dos anos 80, seu sentido era associado à arquitetura do sistema, compreendendo principalmente sua estrutura física ou, muitas vezes, possuindo um sentido restrito a um conjunto de instruções específicas de uma família de computadores (Kruchten et al., 2006). O conceito de Arquitetura de Software como disciplina distinta começou a emergir no início da década de 90, evidenciado pelo aumento do número de contribuições tanto da indústria quanto da academia (Boehm, 2006).

Arquiteturas de software são consideradas como a principal estrutura de um sistema (Clements et al., 2002). Segundo Bass et al. (2003), uma arquitetura compreende elementos de software, as propriedades externamente visíveis desses elementos e o relacionamento entre as partes que a compõe. Arquiteturas são abstrações de sistemas que descrevem o relacionamento entre os componentes, suprimindo detalhes não relacionados à maneira pelo qual componentes usam, são utilizados, estão relacionados ou interagem com os demais componentes (Bass et al., 2003). Os relacionamentos, ou conexões, descrevem como são feitas a transmissão e a comunicação da informação entre as partes do sistema, representadas pelos componentes.

Dentre os oito conceitos descritos por Wasserman (1996) como sendo a base da área de Engenharia de Software, a Arquitetura de Software é considerada como o principal elemento na determinação de aspectos de qualidade e manutenibilidade do software. Da mesma forma, segundo o *Software Engineering Institute* (SEI)[1], requisitos de qualidade

[1] `http://www.sei.cmu.edu/`

de sistemas de software estão fortemente relacionados às suas arquiteturas. Ou seja, arquiteturas de software desempenham um papel determinante na qualidade dos sistemas por elas descritos. Vale a pena ressaltar então que, por arquiteturas de software desempenharem um papel fundamental na determinação da qualidade, elas podem influenciar diretamente no sistema a ser concebido, de modo a habilitar, facilitar, dificultar ou interferir no alcance das metas referentes aos requisitos funcionais e não-funcionais (Nakagawa et al., 2009a).

A seguir, são apresentados conceitos e terminologias definidos no contexto de arquitetura de software, úteis ao entendimento dos conceitos abordados neste trabalho.

3.2.1 Terminologia e Conceitos Básicos

Na área de Arquitetura de Software, diversos conceitos e terminologias vêm sendo definidos a fim de estabelecer um vocabulário comum entre os pesquisadores. Dentre esses termos, podem-se citar: instância arquitetural, estilo arquitetural, padrão arquitetural, arquitetura concreta, modelo de referência e arquitetura de referência. Tais termos definem conceitos distintos pertencentes ao contexto de arquiteturas de software. Contudo, algumas vezes, alguns desses termos têm sido utilizados como sinônimos.

Uma **arquitetura concreta** (ou simplesmente arquitetura de software) consiste de um conjunto coerente de padrões e especificações, que guiam o desenvolvimento de cada parte pertencente a um determinado sistema de software (SEI, 2010). Por sua vez, o termo **instância arquitetural** é utilizado como sinônimo de arquitetura concreta, para um contexto de aplicação ou problema específico, que pode ser proveniente de utilização de uma arquitetura de referência (Nakagawa, 2006).

Um conceito mais abrangente, o de **estilo arquitetural** (ou **padrão arquitetural**), define um conjunto de restrições quanto à forma e à estrutura de uma família de arquiteturas concretas ou instâncias arquiteturais (Bass et al., 2003). Um estilo arquitetural permite aos engenheiros de software, desenvolvedores, arquitetos e projetistas determinarem a classe ao qual pertencem os sistemas de software (Bass et al., 2003; Mendes, 2002). Um exemplo de estilo arquitetural bastante conhecido é o *Cliente-Servidor*, que define a existência de dois elementos: o servidor e o cliente. Esse estilo descreve também que a comunicação entre esses dois elementos deve ser feita por meio de um protocolo. Contudo, não é uma preocupação do estilo arquitetural identificar como cada um dos clientes deve ser construído ou de qual maneira os protocolos de comunicação devem ser implementados (Bass et al., 2003). Outros exemplos de estilo arquitetural encontrados na literatura são (Shaw e Clements, 1997; Shaw e Garlan, 1996): *Pipe and Filters*, *Arquitetura em Camadas* e *Blackboard*. Podem ser encontradas também, combinações resultantes de mais de um estilo arquitetural.

Ainda nesse contexto, uma das mais importantes características de estilos arquiteturais é que esses possuem atributos de qualidade conhecidos, o que permite que arquitetos decidam sobre a utilização de um estilo em detrimento a outros. Alguns estilos apresentam soluções a problemas de desempenho, outros podem ser utilizados em sistemas no qual a segurança é um importante atributo de qualidade, sendo que a decisão sobre qual estilo utilizar é frequentemente considerada como a maior decisão tomada durante a fase inicial de um projeto (Bass et al., 2003).

O conceito de **modelo de referência**, segundo a definição proposta por Bass et al. (2003), consiste da divisão de funcionalidades apresentada de maneira conjunta e o fluxo de dados pelo qual suas partes se relacionam. Tais modelos são decomposições padronizadas do conhecimento de um domínio em partes que cooperam para solução de problemas a ele pertencentes. Um conhecido exemplo de modelo de referência é apresentado pela OASIS (2006), que define em um modelo abstrato a essência da arquitetura orientada a serviços, independente de conceitos tecnológicos.

Ao passo que um modelo de referência consiste de um conjunto minimal de conceitos unificados, axiomas e relacionamentos de um domínio particular, ilustrados de maneira independente de padrões específicos, tecnologias e demais conceitos concretos (OASIS, 2006), **arquiteturas de referência** são definidas como mapeamentos de modelos de referência em elementos de software que, de maneira cooperativa, implementam funcionalidades definidas por esse modelo (Angelov et al., 2009; Bass et al., 2003). Ou seja, uma arquitetura de referência consiste de componentes de software e os relacionamentos entre eles, que implementam funcionalidades relativas às partes definidas no modelo de referência. Segundo Bass et al. (2003), uma arquitetura de referência consiste de um repositório de conhecimento de domínio que promove o reúso arquitetural e apoia o desenvolvimento de sistemas.

Embora os propósitos de arquiteturas de referência e modelos de referência sejam similares, os dois conceitos não são considerados como sinônimos, possuindo definições distintas (Angelov et al., 2009; Bass et al., 2003; OASIS, 2006). Além disso, apesar de serem considerados como conceitos equivalentes (Mendes, 2002), arquiteturas de referência diferenciam-se também de estilos arquiteturais. Ao passo que arquiteturas de referência consistem de estruturas que proveem caracterizações de funcionalidades de sistemas de softwares em um determinado domínio de aplicação, os estilos arquiteturais são soluções conhecidas que podem ser aplicadas de forma independente do domínio. Na Figura 3.1 é ilustrado o relacionamento entre modelos de referência e padrões arquiteturais, utilizados no estabelecimento de arquiteturas de referência. A partir do estabelecimento da arquitetura de referência, são projetadas então arquiteturas concretas, destinadas à construção de sistemas.

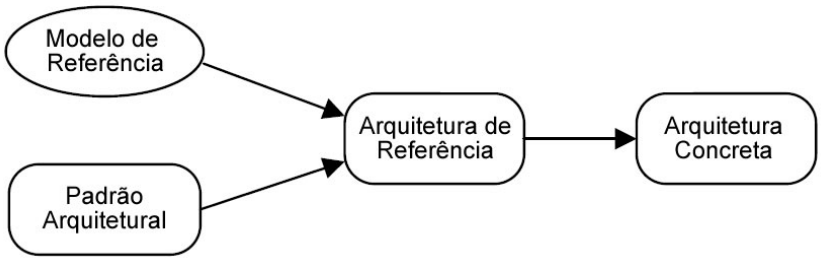

Figura 3.1: Relacionamento entre modelo de referência, padrão arquitetural, arquitetura de referência e arquitetura concreta (Bass et al., 2003)

3.2.2 Representação de Arquiteturas de Software

Em virtude da importância da arquitetura de software no desenvolvimento de sistemas, esforços destinados a sua adequada representação podem ser encontrados. Embora o projeto da arquitetura seja fundamental para o sucesso do desenvolvimento, não existe um consenso acerca da maneira mais adequada para a sua representação. Na grande maioria dos casos, arquiteturas de software são representadas de maneira informal, como modelos contendo componentes e conectores (*box-and-lines*). Componentes consistem de representações das grandes funcionalidades do sistema e dos dados nele armazenados. Os conectores, por sua vez, representam a comunicação entre os componentes, definindo as interações entre eles. Embora essa forma de representação arquitetural seja útil para prover uma visão geral do sistema a ser desenvolvido, é importante salientar que, se utilizada isoladamente, problemas relacionados à ambiguidade quanto ao seu entendimento podem acorrer (Land, 2002).

Face à necessidade de se prover representações formais para arquiteturas de software, que propusessem uma sintaxe e uma estrutura conceitual que permitissem caracterizar uma arquitetura de forma não ambígua, linguagens de descrição arquitetural (do inglês, *Architectural Description Language* – ADL) têm sido propostas (Medvidovic e Taylor, 2000). Essas linguagens fornecem abstrações adequadas à modelagem de grandes sistemas, garantindo detalhes suficientes para o estabelecimento das propriedades desejadas. Nesse contexto, cada ADL envolve uma abordagem particular de especificação e evolução de uma arquitetura, lidando com questões de avaliação específicas, relacionadas à modelagem, abordando aspectos do sistema de forma aprofundada. Contudo, em virtude da especificidade inerente às ADLs, a integração de modelos projetados por meio dessas linguagens a outros artefatos de desenvolvimento pode ser dificultada (Medvidovic et al., 2002).

Além do uso de linguagens de descrição arquitetural, a utilização de diferentes visões na representação de arquiteturas de software vem sendo largamente investigada (ANSI/IEEE, 2000; Buschmann et al., 1996; ISO/IEC, 2007; Kruchten, 1995; Medvidovic et al.,

2002). Segundo Buschmann et al. (1996), cada visão arquitetural representa um aspecto parcial da arquitetura, que reflete em propriedades específicas do sistema de software. Em outras palavras, por meio de visões arquiteturais, é possível visualizar diferentes propriedades de um sistema, de acordo com o ponto de vista no qual a arquitetura é observada (Land, 2002). Apesar de não existir um consenso entre os pesquisadores sobre quais as visões arquiteturais e métodos de representação são mais adequados ao projeto de arquiteturas de software (ANSI/IEEE, 2000), as visões estrutural e comportamental têm sido as mais consideradas (ANSI/IEEE, 2000; Kruchten, 1995; Medvidovic et al., 2002). Merson (2005) propõe um conjunto mais amplo de visões para a representação de arquiteturas de software, que inclui: visão de módulos (*module view*), visão em tempo de execução (*runtime view*), visão de implantação (*deployment view*), visão de dados (*data view*) e visão de implementação (*implementation view*).

Recentemente, a UML (*Unified Modeling Language*) tem emergido como uma importante linguagem para representação de arquiteturas de software, sendo amplamente adotada tanto na acadêmia quanto na indústria (Medvidovic et al., 2002). A UML possui um grande conjunto de construções pré-definidas e é apoiada por um número considerável de ferramentas de modelagem. Além disso, a UML é considerada uma linguagem semi-formal e possui uma representação mais próxima da atividade de implementação, sendo facilmente compreendida pelos desenvolvedores do sistema.

3.2.3 Métodos de Avaliação Arquitetural

Considerando-se a relevância de arquiteturas de software como base sobre as quais todos os sistemas são construídos, métodos de avaliação de arquiteturas têm sido utilizados de forma a aumentar a qualidade dos sistemas desenvolvidos (Barcelos e Travassos, 2006a; Bengtsson et al., 2004; Kazman et al., 1994, 1998). A condução de avaliações em arquiteturas de software permite a descoberta e a resolução de potenciais problemas em sistemas a serem ainda desenvolvidos, sendo, segundo Clements et al. (2002), considerada uma maneira rápida e barata de se evitar o posterior insucesso dos sistemas de software. Dessa forma, uma arquitetura que passa por um processo de avaliação pode apresentar melhores perspectivas quanto ao desenvolvimento de um sistema com qualidade.

Diversos estudos têm sido conduzidos no sentido de estabelecer métodos e critérios para avaliação de arquiteturas de software, sendo que os exemplos mais evidentes encontrados na literatura são o SAAM (*Software Architecture Analysis Method*) (Kazman et al., 1994) e o ATAM (*Architecture Trade-off Analysis Method*) (Kazman et al., 1998). Além disso, adaptações desses métodos para a avaliação especificamente de arquiteturas de referência também podem ser encontradas, com os trabalhos de Angelov et al. (2008), Graaf et al. (2005) e Gallagher (2000). Esses trabalhos constituem importantes iniciativas na

avaliação de arquiteturas de referência, uma vez que métodos normalmente utilizados na avaliação de arquiteturas concretas não podem ser diretamente aplicados em arquiteturas de referência, principalmente porque existem diferenças entre essas arquiteturas, tais como o nível de abstração (Angelov et al., 2008).

Observa-se que a utilização de métodos de avaliação permite a análise de riscos relacionados à evolução de software, apresentando potenciais estratégias para lidar com esses riscos (Slyngstad et al., 2008). Há, entretanto, pouco consenso sobre quais questões deveriam ser tratadas pelo método adotado e sobre qual o método mais adequado para uma particular situação. Como forma de auxiliar o entendimento e a escolha de métodos de avaliação de arquiteturas adequados a cada contexto, classificações e comparações sobre os métodos podem ser encontradas (Babar e Gorton, 2004; Barcelos e Travassos, 2006b; Dobrica e Niemel, 2002; Ionita et al., 2002; Oliveira e Nakagawa, 2009).

3.3 Arquitetura de Referência

Arquiteturas de referência têm se destacado também como uma importante área de pesquisa em Arquitetura de Software, provendo direções mais precisas para a especificação de arquitetura concretas pertencentes a um dado domínio de aplicação (Angelov et al., 2008). Por conseguinte, arquiteturas de referência influenciam diretamente na qualidade e no projeto de todo um conjunto de arquiteturas concretas e no conjunto de sistemas derivados dessa arquitetura (Angelov et al., 2009).

Segundo Muller (2008), arquiteturas de referência podem ser utilizadas como guia para projeto de arquiteturas concretas ou como ferramenta de padronização, provendo interoperabilidade entre sistemas ou componentes de sistemas. Dessa forma, uma arquitetura de referência pode ser utilizada em diversos contextos, resultando no desenvolvimento de instâncias arquiteturais distintas, dependentes dos interessados (*stakeholders*) e metas pertencentes a cada instituição, como ilustrado na Figura 3.2. Arquiteturas de referência têm por objetivo facilitar o entendimento de um determinado domínio, provendo vocabulário comum, compreensão das partes que o compõem e do relacionamento entre essas partes.

Diversas arquiteturas de referência para os mais diferentes domínios de aplicação podem ser encontradas. Os trabalhos de Arsanjani et al. (2007); Choi et al. (2009); Costagliola et al. (2006); Eickelmann e Richardson (1996); Nakagawa (2006); Nakagawa et al. (2007); Peristeras et al. (2009) são exemplos de arquiteturas de referência propostas. Contudo, poucas arquiteturas de referência são encontradas para o domínio de engenharia de software, ou seja, que visam apoiar o desenvolvimento de ferramentas que permitam a automatização de atividades como a inspeção, verificação e validação de software. A

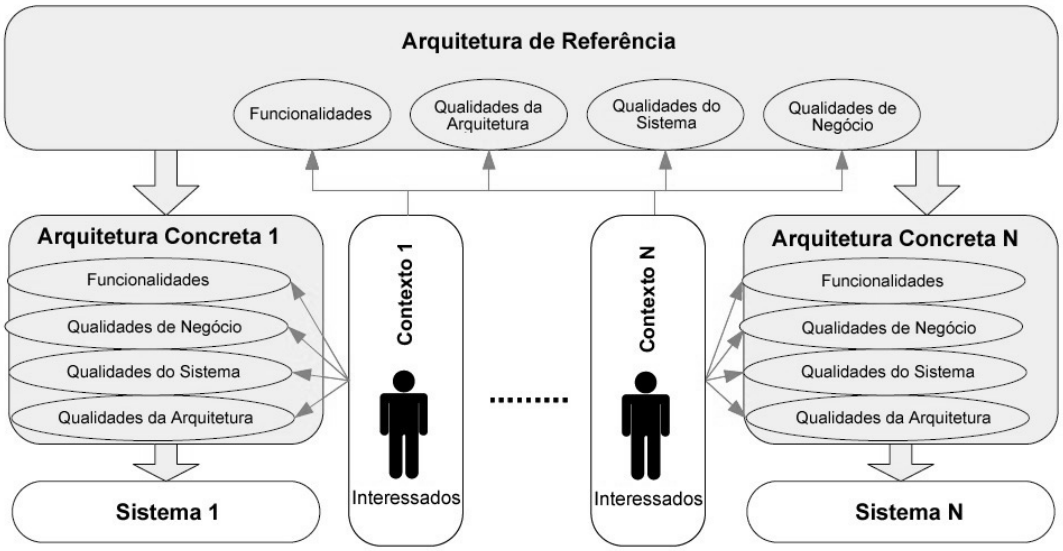

Figura 3.2: Interação de interessados e contextos entre arquiteturas concretas e de referência (Angelov et al., 2008)

arquitetura de referência proposta por Nakagawa (2006) tem por objetivo auxiliar no desenvolvimento de ferramentas e ambientes de engenharia de software. Além disso, duas arquiteturas de referência para ferramentas de teste de software também podem ser encontradas (Eickelmann e Richardson, 1996; Nakagawa et al., 2007), sendo essas apresentadas em mais detalhes na Seção 3.3.2.

Em virtude da diversidade de áreas de aplicação e dos interesses envolvidos em sua utilização, arquiteturas de referência podem ser classificadas quanto a diversos aspectos. A seguir, são discutidos aspectos relacionados aos tipos de arquiteturas de referência existentes, seus propósitos de utilização e princípios de desenvolvimento.

3.3.1 Classificações de Arquiteturas de Referência

Arquiteturas de referência vêm sendo desenvolvidas por pesquisadores e pela indústria em diferentes contextos e com diferentes objetivos (Angelov et al., 2009). O que se espera de uma arquitetura de referência é que essa possa facilitar o desenvolvimento de múltiplos projetos em uma instituição ou o desenvolvimento cooperativo de produtos, envolvendo diferentes instituições inseridas em um domínio comum (Angelov et al., 2009). Entretanto, para que arquiteturas de referência efetivas sejam desenvolvidas, é necessário que sejam considerados não somente os sistemas a serem produzidos, mas também os contextos para os quais elas são desenvolvidas e sob quais objetivos. Arquiteturas de referência podem variar de diversas maneiras, seja quanto ao contexto de produção, metas aos quais são desenvolvidas ou como são descritas. Assim, segundo Angelov et al. (2009), uma arquitetura de referência pode ser classificada quanto: (i) ao contexto para o qual foi

desenvolvida; (ii) seu propósito de uso; e (iii) os insumos utilizados em sua construção. Essa classificação é descrita em mais detalhes a seguir:

Contexto de aplicação: Arquiteturas de referência podem ser projetadas para o contexto de uma *única organização*, que deseja construir uma família de produtos similares, ou por *múltiplas organizações* que compartilhem propriedades em comum, como sua localização geográfica ou domínio de mercado;

Propósito de uso: Com relação ao propósito de uma arquitetura de referência, dois possíveis usos pretendidos podem ser identificados (Angelov et al., 2009, 2008; Muller, 2008): a padronização e a facilitação. A *padronização* de arquiteturas concretas tem por objetivo prover interoperabilidade entre sistemas, por meio da definição de consenso arquitetural. Já a *facilitação* tem como intenção oferecer linhas guia, na forma de boas práticas e padrões, para o projeto de arquiteturas concretas; e

Insumos utilizados: Uma arquitetura de referência pode ser projetada antes de existência de qualquer sistema, por meio do conhecimento de domínio, sendo denominada como arquitetura de referência *preliminar*, ou por meio do acúmulo de experiência proveniente de um conjunto de sistemas já implementados anteriormente, sendo definida como uma arquitetura de referência *clássica* (Angelov et al., 2008).

Com base nas possíveis características e na forma nas quais podem ser classificadas, Angelov et al. (2009) definem seis tipos de arquiteturas de referência, detalhados a seguir:

Tipo 1 (Múltiplas Organizações, Clássica, Padronização): Essas arquiteturas são utilizadas na obtenção de consenso arquitetural entre organizações e, em virtude disso, são projetadas por meio de descrições de componentes e suas interfaces, em altos níveis de abstração. Nesse tipo de arquitetura, o detalhamento dos componentes é desnecessário, pois cada organização deve ser livre para escolher quais detalhes concretos são pertinentes às suas metas;

Tipo 2 (Única Organização, Clássica, Padronização): Arquiteturas pertencentes a essa classificação têm por objetivo auxiliar na construção de uma família de sistemas de software com características semelhantes. Tais arquiteturas possuem características mais concretas, podendo especificar detalhes, tais como tecnologias, aplicações existentes e padrões, de acordo com as metas desejadas pela organização;

Tipo 3 (Múltiplas Organizações, Clássica, Facilitação): Arquiteturas de referência pertencentes a essa classificação são utilizadas para fomentar a construção de produtos de software, por meio de uma descrição mais concreta de componentes e

interfaces, provendo direções ao desenvolvimento. Esse tipo de arquitetura de referência tem como exemplo a *Microsoft Application Architecture for .Net* (Microsoft, 2002), que fornece direções para o projeto de arquiteturas e aplicativos baseados no *framework* .NET;

Tipo 4 (Única Organização, Clássica, Facilitação): Arquiteturas pertencentes a esse grupo possuem características similares ao do Tipo 2; contudo, são projetadas para facilitar o desenvolvimento e, em virtude disso, são descritas de maneira mais informal;

Tipo 5 (Múltiplas Organizações, Preliminares, Facilitação): Arquiteturas de referência pertencentes a esse tipo são projetadas a partir do conhecimento de domínio, antes da existência de qualquer sistema concreto e, por isso, são referenciadas como preliminares. Essas arquiteturas definem componentes requeridos à sua implementação, podendo indicar algoritmos e protocolos que facilitem a interação entre esses componentes; e

Tipo 6 (Única Organização, Preliminares, Facilitação): Apesar de plausível, a definição de arquiteturas desse tipo requer substancial esforço, sendo sua produção restrita às organizações líderes de mercado.

No desenvolvimento dessas arquiteturas, diferentes tipos de interessados podem estar envolvidos. O estabelecimento de uma arquitetura de referência pode ser realizado por organizações de software, organizações de usuários, centros de pesquisa, projetistas de software, gerentes de projeto, organizações de padronização ou por composições desses grupos. Como cada conjunto de arquiteturas de referência possui suas particularidades, diferentes tipos de instituições costumam concentrar esforços na construção dessas arquiteturas (Angelov et al., 2009). No projeto de arquiteturas de referência construídas da maneira clássica, geralmente estão envolvidas organizações de software, organização de usuários, projetistas, gerentes de projeto e organizações de padronização. Já as arquiteturas preliminares geralmente são desenvolvidas por projetistas de software e centros de pesquisa.

Em especial, arquiteturas preliminares desenvolvidas por centros de pesquisa, com o objetivo de facilitar o desenvolvimento de sistemas ainda não existentes, são consideradas um tipo particular de arquitetura. O projeto dessas arquiteturas de referência concentra-se não em requisitos do domínio dos interessados, mas sim em elementos de inovação da arquitetura. Por esse motivo, tais arquiteturas não são consideradas implementações de sistema na prática e sim contribuições para projetos futuros (Angelov et al., 2009). A ERA (*E-contracting Reference Architecture*), proposta por Angelov e Grefen (2008), é

destinada ao domínio de contratos eletrônicos e pode ser citada com exemplo pertencente a esse conjunto especial de arquiteturas.

Além disso, a estimativa dos resultados provenientes de uma arquitetura de referência também são dependentes do tipo ao qual ela pertence (Angelov et al., 2009, 2008). Arquiteturas desenvolvidas da maneira clássica, por meio da observação de sistemas já implementados, têm seus resultados estimados mais facilmente, ao passo que as preliminares possuem resultados difíceis de estimar (Angelov et al., 2008).

3.3.2 Arquiteturas de Referência de Teste de Software

Como discutido anteriormente, a atividade de teste é uma das mais importantes na garantia da qualidade de sistemas de software (Myers et al., 2004). Em virtude disso, ferramentas de teste vêm sendo propostas e utilizadas. Entretanto, tais ferramentas, na maioria dos casos, são implementadas individualmente e de maneira independente, com arquiteturas e estruturas internas próprias, podendo apresentar dificuldades quanto à integração e ao reúso (Nakagawa et al., 2007). Em virtude disso, arquiteturas de referência para o domínio de teste de software podem ser encontradas (Eickelmann e Richardson, 1996; Nakagawa et al., 2007).

A primeira arquitetura de referência para o domínio de testes foi proposta por Eickelmann e Richardson (1996). De uma maneira simplificada, essa arquitetura é dividida em seis módulos, que representam funcionalidades de uma ferramenta de teste de software, sendo esses[2]: execução do teste, desenvolvimento do teste, análise de falhas, mensuração de resultados do teste, gerenciamento dos dados de teste e planejamento do teste. Contudo, segundo os próprios autores, essa arquitetura é introdutória e com alto nível de abstração. Apesar de sua relevante contribuição, sendo a primeira desenvolvida para o domínio de teste, essa arquitetura apresenta limitações, não informando detalhes que auxiliem de fato sua utilização (Nakagawa, 2006). Essa arquitetura não discute detalhes como comunicação entre seus componentes, interfaces, fluxo de dados ou maiores informações sobre cada uma de suas funcionalidades.

Com o propósito de mitigar os problemas apresentados pela proposta de Eickelmann e Richardson (1996) e apoiar o desenvolvimento de ferramentas de teste, foi desenvolvida a RefTEST (*Reference Architecture for Testing Tools*) (Nakagawa et al., 2007). A RefTEST tem como base uma arquitetura mais genérica, denominada RefASSET (*Reference Architecture for Software Engineering Environment*) (Nakagawa, 2006), que é destinada ao desenvolvimento de ferramentas e ambientes de engenharia de software. Em virtude

[2]A estrutura da arquitetura de referência e o detalhamento de suas camadas são melhor analisados no Capítulo 4.

disso, a RefASSET é brevemente discutida de modo a facilitar o entendimento da RefTEST.

A RefASSET tem como princípio a separação de interesses (do inglês, *Separation of Concerns*) (Nakagawa e Maldonado, 2008), que possibilita a identificação de elementos participantes de um sistema, ocultando a complexidade por meio de abstrações, promovendo a modularização e o reúso em sistemas de software (Dijkstra, 1976). Nesse contexto, interesses podem ser considerados funcionais ou não-funcionais. Na RefASSET, cada atividade de Engenharia de Software é considerada um interesse. Atividades como a especificação de requisitos, teste de software e gerência de configuração são exemplos de interesses. Para identificar todos os interesses em ambientes de engenharia de software, Nakagawa (2006) utiliza como base o padrão ISO/IEC 12207 (ISO, 1995), que agrupa as atividades de engenharia de software em três categorias — primária, organizacional e de apoio — dependendo das características de cada atividade. As atividades organizacionais e de apoio são consideradas interesses transversais (Nakagawa e Maldonado, 2007), como ilustrado na Figura 3.3, e permeiam todo o processo de desenvolvimento de software. Um exemplo de interesse transversal é a documentação, uma atividade de apoio que é realizada desde a especificação de requisitos até a manutenção.

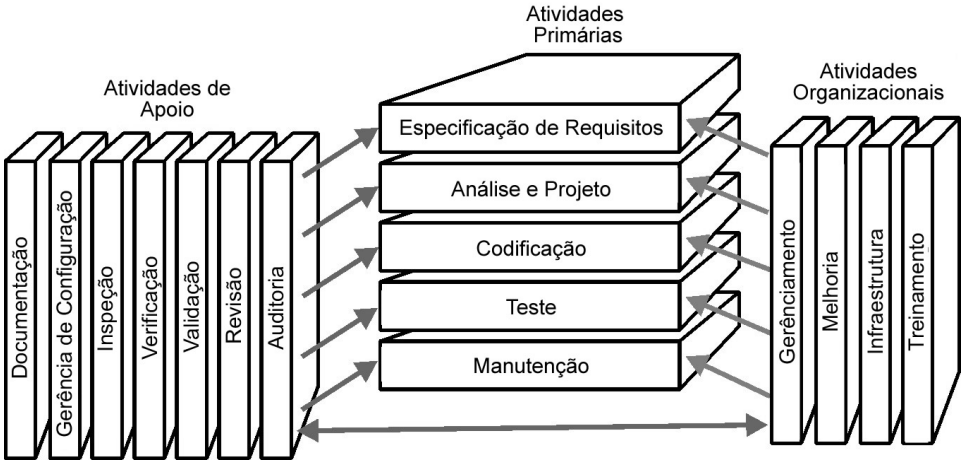

Figura 3.3: Relacionamento entre atividades primárias, organizacionais e de apoio (Nakagawa et al., 2007)

Nesse contexto, entende-se por Programação Orientada a Aspectos (POA) (Kiczales et al., 1997) uma abordagem de desenvolvimento de software que dá apoio a uma melhor separação de interesses e mais adequadamente reflete a forma como os desenvolvedores pensam sobre o sistema (Kiczales et al., 1997). Essencialmente, a POA introduz uma unidade de implementação modular — o aspecto — que tem sido tipicamente utilizada para encapsular um interesse transversal (isto é, um interesse que se encontra espalhado ou entrelaçado com outros interesses do sistema). Modularidade, manutenibilidade e facilidade de implementação podem ser alcançados com a POA (Laddad, 2003).

A RefASSET é também baseada no padrão arquitetural MVC (*Model-View-Controller*) (Buschmann et al., 1996) e na arquitetura em três camadas (*three tier architecture*) (Eckerson, 1995), uma vez que são consolidadas como arquiteturas que proveem uma adequada separação de interesses em sistemas interativos. A Figura 3.4 apresenta uma representação geral da RefASSET. A Camada de Apresentação refere-se aos módulos responsáveis pela interface com o usuário. Essa camada é composta pelo Controlador responsável por processar os eventos vindos do usuário e invocar as funcionalidades contidas no Modelo. A Visão contém a interface do usuário propriamente dita. Outra importante parte da arquitetura é a Camada de Aplicação, que contém o Modelo que agrega o núcleo das funcionalidades das ferramentas de engenharia de software. A Camada de Persistência é a terceira camada e corresponde à base de dados e funcionalidades para manipulação da persistência dos dados.

A RefTEST, obtida a partir da especialização arquitetural[3] da RefASSET para o domínio de teste de software, foi projetada por meio da investigação de informações de elementos do domínio, tais como arquiteturas de software, ferramentas de teste e documentos relacionados. Nakagawa et al. (2007) consideraram também processos de teste de software, uma ontologia de teste (Barbosa et al., 2006) e estudos de ferramentas de teste. Uma vez que a RefTEST é base para a arquitetura proposta neste trabalho, no Capítulo 4 são apresentados maiores detalhes acerca de seus requisitos e sua estrutura.

3.3.3 Processo para o Estabelecimento de Arquiteturas de Referência

A sistematização do processo para o estabelecimento de arquiteturas de referência constitui uma atividade de grande importância (Nakagawa et al., 2009a). O trabalho de Nakagawa et al. (2009a) introduz um processo denominado ProSA-RA, que tem por objetivo sistematizar um conjunto de passos para a construção de arquiteturas de referência orientadas a aspecto. O ProSA-RA foi proposto como resultado do estabelecimento da RefASSET e da RefTEST, arquiteturas de referência que apresentam aspectos arquiteturais (isto é, aspectos em nível arquitetural) (Nakagawa et al., 2009a).

O ProSA-RA, ilustrado na Figura 3.5, define os seguintes passos para o estabelecimento de uma arquitetura de referência:

- **Passo 1: Investigação das Fontes de Informação:** São selecionadas as principais fontes de informação do domínio. Tais fontes devem prover informações sobre o processo, atividades e tarefas a serem automatizadas. Fontes de informação podem

[3]Entende-se por especialização arquitetural a construção de uma arquitetura de referência mais específica, a partir de uma mais genérica de um mesmo domínio (Nakagawa et al., 2007).

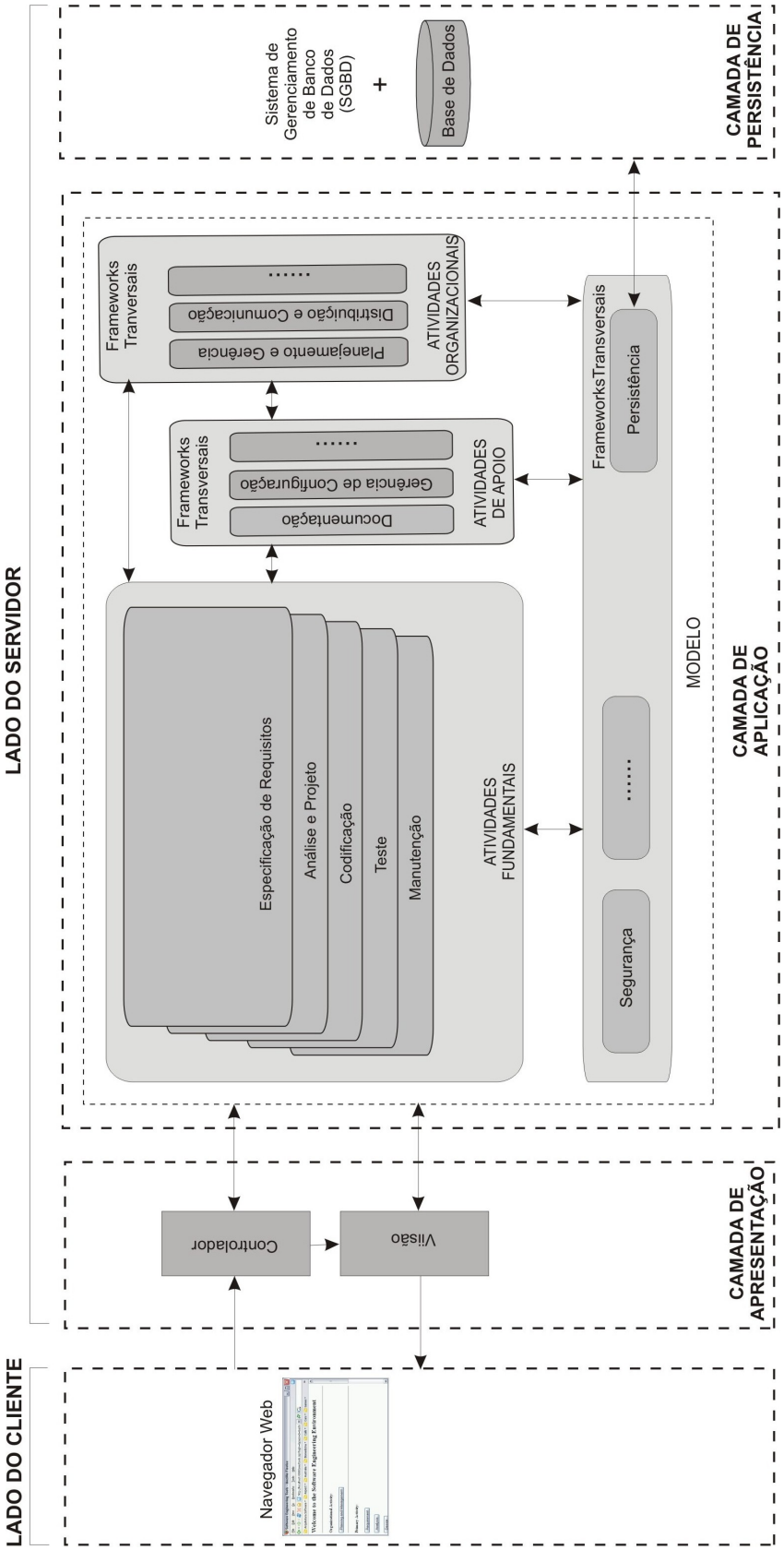

Figura 3.4: Visão geral da RefASSET (Nakagawa et al., 2007)

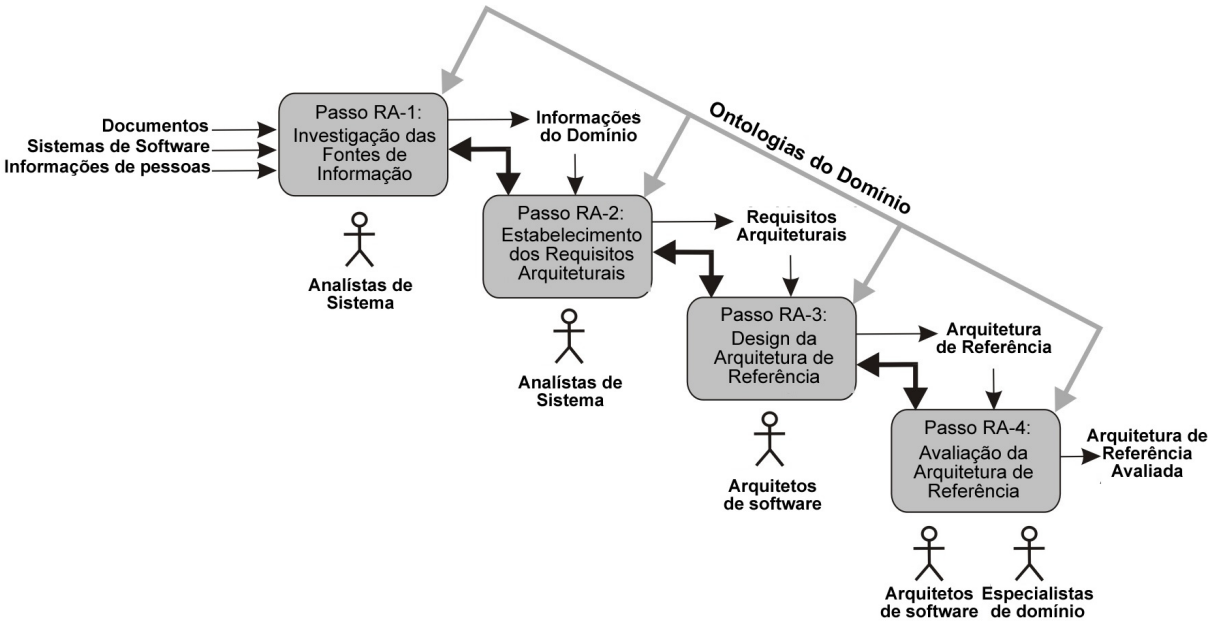

Figura 3.5: Representação do ProSA-RA (Nakagawa et al., 2009a)

ser provenientes de pessoas inseridas no domínio (usuários, pesquisadores, especialistas, entre outros), sistemas desenvolvidos (quando disponíveis), publicações, documentos relacionados e ontologias. Como resultado desse passo, um grupo de fontes de informação é selecionado;

- **Passo 2: Estabelecimento de Requisitos Arquiteturais:** Com base nas fontes de pesquisa selecionadas, informações do domínio são elicitadas, resultando em um conjunto de requisitos da arquitetura de referência. Para isso, requisitos dos sistemas de software são identificados e mapeados em conceitos do domínio. Por fim, esses conceitos são classificados em transversais e não-transversais, com o objetivo de identificar aspectos arquiteturais. Em geral, conceitos relacionados a muitos requisitos ou requisitos não-funcionais possuem característica transversal;

- **Passo 3: Projeto da Arquitetura de Referência:** É o passo no qual é realizado o projeto da arquitetura de referência. Durante o projeto são utilizadas diferentes visões arquiteturais, a UML e suas extensões para representação adequada de aspectos. As visões arquiteturais consideradas pelo ProSA-RA são: visão de módulo, visão em tempo de execução, visão de implantação e visão conceitual. Nesse passo, os aspectos arquiteturais devem ser considerados de uma maneira especial, sendo representados por uma notação específica e adequada (Nakagawa e Maldonado, 2007).

- **Passo 4: Avaliação da Arquitetura de Referência:** Por fim, inspeções por meio de *checklist* são utilizadas de forma a avaliar a qualidade da arquitetura de referência. *Checklists* são questionários que guiam revisores na detecção de defeitos em

documentos. Nesse caso especial, *checklists* são utilizados na inspeção da documentação relacionada à arquitetura de referência estabelecida. Por meio dessa técnica, características de qualidade, tais como manutenibilidade, desempenho, segurança, usabilidade e reúso são avaliadas. Além disso, defeitos de omissão, ambiguidade, inconsistência e de informações incorretas são identificados.

O ProSA-RA encontra-se atualmente em uso no estabelecimento de arquiteturas de referência, tais como para ferramentas de visualização (Nakagawa et al., 2009a) e para ambientes educacionais (Fioravanti et al., 2010).

3.4 Arquitetura Orientada a Serviço

No contexto de arquitetura de software, a Arquitetura Orientada a Serviço (do inglês, *Service Oriented Architecture* – SOA) tem emergido como um estilo arquitetural que visa prover diversas vantagens quanto ao reúso, produtividade e evolução de aplicações (Josuttis, 2007; Papazoglou e Heuvel, 2007). Sistemas de software desenvolvidos a partir desse estilo arquitetural possuem como principais características a independência de linguagem de programação e a flexibilidade quanto à plataforma, uma vez que funcionalidades oferecidas por serviços são definidas por meio de linguagens de descrição padronizadas (Papazoglou et al., 2008). Além disso, sistemas orientados a serviços podem ser construídos pela composição de funcionalidades mais simples, formando aplicações completas e, segundo Papazoglou e Heuvel (2007), de forma produtiva. Nesta seção, são discutidos os conceitos referentes à SOA, os protocolos e linguagens utilizados por serviços web, bem como arquiteturas de referência orientadas a serviço, uma vez que esses são temas relacionados a este trabalho.

3.4.1 Conceitos Básicos

A SOA é um estilo arquitetural que provê baixo acoplamento na construção de aplicações distribuídas, mesmo em ambientes heterogêneos (Papazoglou e Heuvel, 2007). Sistema orientados a serviço são independentes de linguagem de programação, de sistema operacional e permitem que organizações exponham funcionalidades em interfaces autodescritivas (Papazoglou et al., 2008). Dessa forma, SOA introduz o conceito de composição de aplicações, por meio da descoberta de interfaces invocáveis para a realização de tarefas mais completas (Papazoglou et al., 2007). Para isso, é introduzido o conceito de serviço de negócio como unidade fundamental de projeto, construção e composição.

Serviços são módulos bem definidos e autocontidos que fornecem funcionalidades de negócio e são independentes do estado ou contexto de outros serviços (Erl, 2005; Papazo-

glou e Heuvel, 2007). Um serviço deve ocultar informações relacionadas a seus detalhes de implementação, expondo apenas informações relativas às suas interfaces (Erl, 2005). Além disso, serviços devem disponibilizar descrições necessárias à sua descoberta, para que outros serviços possam compreender e requisitar suas funcionalidades. Descrições de serviços são disponibilizadas na forma de detalhamentos formais que estabelecem regras para garantia da interoperabilidade entre serviços.

Em SOA, um mesmo serviço pode assumir diferentes papéis, dependendo do contexto no qual é observado (Erl, 2005). Segundo Papazoglou e Heuvel (2007), serviços podem ocupar dois papéis principais: consumidor e provedor. Um serviço, quando desempenha o papel de provedor, expõe sua interface, publicando sua descrição para ser posteriormente invocado. Já um serviço consumidor utiliza funcionalidades disponibilizadas por um provedor de acordo com suas necessidades de negócio (Leymann et al., 2002). Uma abordagem alternativa ocorre quando um serviço provê funcionalidades compostas, sendo rotulado como agregador de serviços. Um agregador desempenha um duplo papel, atuando tanto como provedor quanto como consumidor. Ao mesmo tempo em que um agregador desempenha o papel de provedor para uma aplicação, também requisita serviços de outros provedores de modo a compor funcionalidades mais completas, como ilustrado na Figura 3.6.

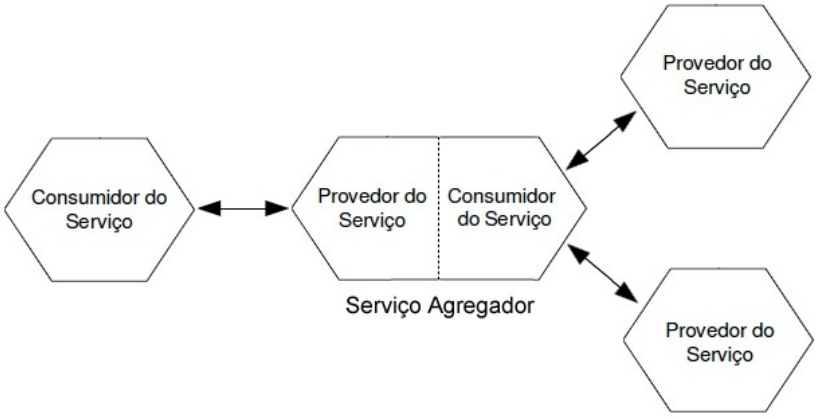

Figura 3.6: Papéis desempenhados por um serviço agregador (adaptado de Papazoglou e Heuvel (2007))

Para que requisições aos serviços sejam realizadas, é necessário que o endereço do provedor seja inicialmente fornecido ao seu consumidor. A comunicação entre serviços pode ser feita de duas maneiras: direta e por meio de serviços intermediários. Na comunicação direta, o consumidor possui conhecimento prévio do endereço do serviço ao qual deseja utilizar. Já a comunicação por meio de serviço intermediário, o serviço consumidor informa a um registro (*broker*) sobre qual funcionalidade deseja utilizar (Papazoglou e Heuvel, 2007). O registro, por sua vez, consulta um conjunto de serviços nele publica-

dos, informando ao consumidor o endereço de um serviço que atenda às especificações informadas. No segundo caso, a interação entre provedor e consumidor é feita de maneira indireta, tornando a escolha de um serviço transparente para quem o solicita. Na Figura 3.7 é ilustrado o relacionamento entre as partes que compõem a estrutura de comunicação em SOA. Segundo Josuttis (2007), a ideia utilizada na interação entre serviços em SOA baseia-se nos três principais papéis do mercado de trabalho: provedores de serviços, consumidores de serviços e corretores que relacionam provedores e consumidores por meio da publicação e localização de serviços.

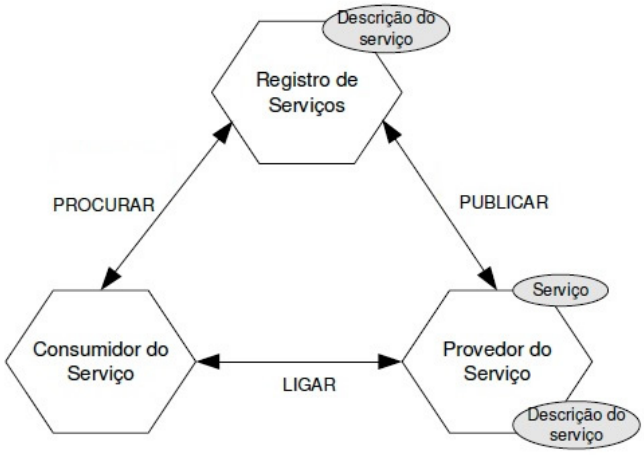

Figura 3.7: Estrutura genérica da interação entre serviços em SOA (Papazoglou e Heuvel, 2007)

3.4.2 Composição de Serviços

Serviços podem representar tanto funcionalidades construídas especificamente para novas aplicações quanto abstrações de componentes de software existentes, provenientes de aplicações legadas. Serviços podem ser oferecidos como funcionalidades simples ou compostas (Josuttis, 2007). Os serviços simples proveem funcionalidades de negócio básicas, que não fazem sentido quando divididas em múltiplos serviços. Segundo Josuttis (2007), há dois tipos de serviços básicos: serviços envolvendo dados e serviços envolvendo lógica. Serviços envolvendo dados geralmente realizam leitura ou escrita de informações, como por exemplo, a criação de um cliente ou a adição de um endereço. Serviços envolvendo lógica representam regras de negócio fundamentais e, usualmente, recebem uma entrada e retornam o resultado correspondente. Por exemplo, um serviço que recebe o valor de um ano e retorna informando se o ano é bissexto ou não é um serviço simples envolvendo lógica.

Um serviço composto constitui um processo que agrega informações sobre diversos serviços, provendo funcionalidades mais completas, construídas a partir de funcionalidades oferecidas por serviços mais simples (Kazhamiakin et al., 2006). Serviços compostos agregam valor às aplicações por meio da interação e combinação de serviços existentes. Segundo Kazhamiakin et al. (2006), a composição de serviços constitui uma das mais promissoras características relacionadas à SOA. Serviços compostos viabilizam o reaproveitamento de componentes, acelerando o desenvolvimento de aplicações. Combinar serviços para criar aplicações requer a definição de uma sequência de composição, denominada processo de negócio. Para a construção de aplicações de alto nível, por meio de serviços pertencentes a diferentes empresas, padrões de interação que modelem os processos de negócio são necessários (Peltz, 2003). Os conceitos de orquestração e coreografia, que descrevem aspectos da criação de processos de composição, são descritos a seguir:

Orquestração de Serviços: estabelece que a coordenação das interações entre os serviços participantes de uma composição deve ser realizada por meio de um coordenador central. Para isso, a orquestração de serviços faz uso de um processo de negócio executável que interage com os serviços, por meio da transmissão de mensagens (Peltz, 2003). Esse processo inclui lógicas de negócio e ordens de execução de tarefas, podendo interligar aplicações e organizações na definição de modelos de processo transacionais e duradouros (Peltz, 2003). Na orquestração, o processo é sempre controlado a partir do ponto de vista de uma das partes do negócio (Erl, 2005). Na Figura 3.8 é apresentada uma representação genérica de orquestração, na qual serviços clientes solicitam funcionalidades a um processo de negócio, o Coordenador Central, responsável por invocar funcionalidades de outros serviços existentes.

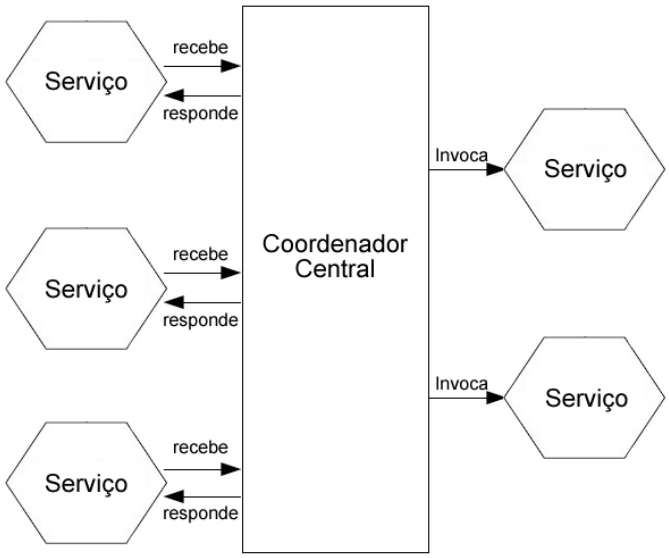

Figura 3.8: Estrutura genérica de uma orquestração (adaptado de Peltz (2003))

Coreografia de Serviços: é uma técnica que permite que partes envolvidas em um processo descrevam seus próprios aspectos de interação, caracterizando uma interação mais colaborativa (Papazoglou et al., 2008). A coreografia oferece um meio pelo qual regras de participação e colaboração podem ser claramente definidas de maneira conjunta. A coreografia define uma sequência de mensagens no qual podem estar envolvidos múltiplos consumidores, provedores e parceiros, interagindo individualmente e de maneira descentralizada (Papazoglou et al., 2008). Na Figura 3.9 é apresentada uma estrutura genérica de comunicação por meio de coreografia, na qual serviços colaboram sem a presença de um coordenador central.

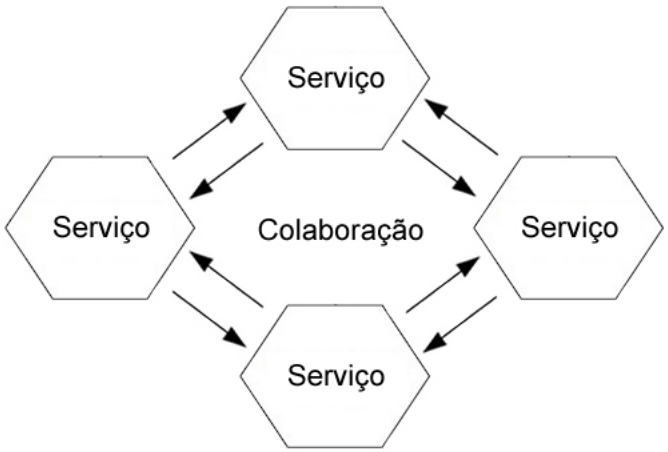

Figura 3.9: Estrutura genérica de uma coreografia (adaptado de Peltz (2003))

3.4.3 Enterprise Service Bus

Para que serviços desenvolvidos por tecnologias com características distintas possam ser integrados de maneira efetiva, soluções que apoiem a infraestrutura da SOA são necessárias. O barramento *Enterprise Service Bus* (ESB) é uma estrutura baseada em mensagens, desenvolvida para habilitar a implementação, implantação e o gerenciamento de sistemas de software baseados em SOA (Papazoglou e Heuvel, 2007). Um ESB desempenha o papel de intermediário entre os serviços, por meio da criação de uma camada de abstração para a comunicação entre aplicações, conforme ilustrado na Figura 3.10. Um barramento ESB formaliza a publicação, provendo um registro dos serviços disponíveis e consumidores que se conectarão a eles (Schmidt et al., 2005). Dessa forma, um ESB é responsável pelo controle de registros e publicações de serviços, comunicação, conversão de dados e protocolos, entre outras atividades. Segundo Papazoglou e Heuvel (2007), um ESB constitui um conjunto de funcionalidades estruturais disponibilizadas na forma de uma camada intermediária (*middleware*) que possibilita a implementação da SOA e

atenua incompatibilidades entre serviços executados sobre diferentes plataformas e tipos de dados.

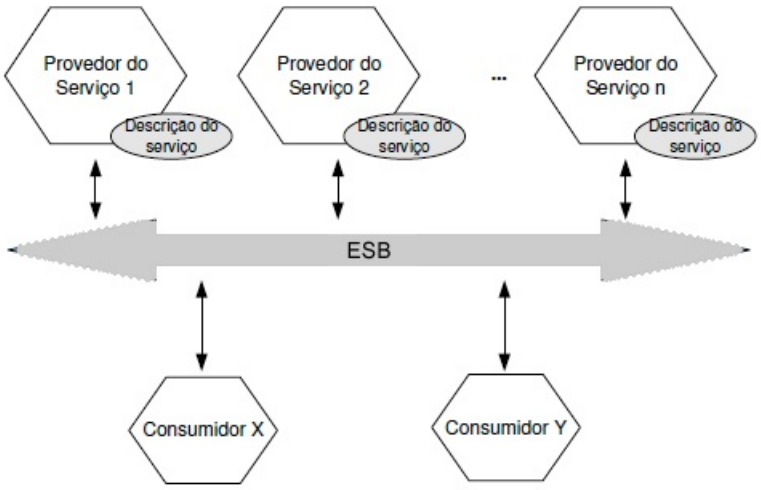

Figura 3.10: Estrutura genérica de um ESB (adaptado de Papazoglou e Heuvel (2007))

Segundo Josuttis (2007), um ESB possui, mas não se limita, as seguintes características:

Mapeamento de dados: Na manipulação de muitos serviços, implementados em diferentes linguagens de programação, é necessário que características próprias de cada serviço sejam consideradas, de forma que haja compatibilidade entre os dados transmitidos entre eles. Por esse motivo, o barramento deve ser capaz de mapear tipos de dados de uma aplicação para os correspondentes tipos de outra;

Roteamento inteligente: Em SOA podem existir diferentes versões de um mesmo serviço. O ESB deve então ser capaz de gerenciar o roteamento de requisições para os serviços corretos. Além disso, o barramento deve proporcionar balanceamento de carga e também controle de redundância;

Segurança: Por meio do ESB são transmitidas informações de diversos aplicativos. Algumas informações podem possuir conteúdo confidencial; por isso, o ESB deve garantir segurança na transmissão e acesso às informações trafegadas;

Garantia de entrega: O ESB deve garantir a entrega de mensagens. Caso não seja possível entregar alguma mensagem, o barramento deve informar que um erro ocorreu, para que o mesmo possa ser solucionado; e

Monitoramento e *logging*: O barramento deve monitorar todas as transações que nele ocorrem, criando *logs* para que possíveis falhas possam ser detectadas e corrigidas mais facilmente.

3.4.4 Qualidade de Serviço

Em sistemas orientados a serviço, frequentemente, funcionalidades solicitadas por serviços clientes podem ser oferecidas por diferentes serviços provedores, que possuem características de funcionamento distintas. Em outras palavras, isso significa que, assim como nos negócios, para uma dada necessidade, existem diversos provedores de serviço que oferecem a mesma funcionalidade. Nesse caso, a escolha sobre qual serviço deve ser invocado pode ser realizada com base em um conjunto de requisitos não-funcionais, ou seja, atributos de qualidade (Canfora et al., 2008). Dessa forma, um serviço cliente, ao pesquisar sobre uma funcionalidade desejada, poderá também especificar os requisitos de qualidade de serviço (do inglês, *Quality of Service – QoS*) relacionados ao serviço que deseja utilizar (Canfora et al., 2008). De acordo com o padrão ISO 8402 (ISO, 2002), QoS pode ser definida por meio de atributos como: preço, tempo de resposta, disponibilidade e reputação. Além disso, podem ser analisados requisitos de qualidade específicos de um domínio como, por exemplo, a qualidade da imagem e o número de cores em um serviço de processamento de imagens. Pelo uso da QoS, é possível que um usuário especifique restrições sobre determinados valores relacionados aos atributos de qualidade como, por exemplo, o preço de um serviço que não pode ser superior a um determinado valor (Canfora et al., 2005). Em contrapartida, o prestador do serviço pode também estimar intervalos de valores para os seus atributos de QoS, como parte do um contrato a ser firmado com os potenciais utilizadores.

Contratos em nível de serviço (do inglês, *Service Level Agreement* – SLA) podem ser utilizados como maneiras formais para descrever o acordo entre um serviço e seus clientes, expressando seus direitos e obrigações (Dan et al., 2004). SLAs especificam o que serviços clientes esperam dos serviços provedores e o que os provedores esperam deles. Segundo Dai et al. (2007), um bom contrato implica em obrigações, bem como em benefícios, para ambas as partes. Ou seja, um cliente só pode utilizar uma funcionalidade se cumprir todas as suas pré-condições, o que caracteriza uma obrigação para o serviço cliente e um benefício para o serviço provedor. Feito isso, após o início da interação, existe o compromisso de que determinados requisitos sejam garantidos pelo serviço provedor, representando um benefício para o serviço cliente.

Para que serviços possam operar de maneira satisfatória, inspeções relacionadas ao cumprimento de requisitos de qualidade podem ser realizadas. Para isso, interações entre os serviços devem ser monitoradas, de forma que não conformidades relacionadas aos requisitos de qualidade possam ser sinalizadas. Nesse contexto, a atividade de monitoramento de QoS pode ser realizada tanto pelas partes envolvidas na interação quanto por serviços certificadores, que são serviços terceirizados responsáveis por inspecionar e indicar a qualidade dos serviços disponíveis. A garantia da QoS possui grande importância, uma

vez que, no contexto da SOA, aplicações podem ser desenvolvidas por diferentes empresas, sendo esse desenvolvimento realizado sob infraestruturas heterogêneas com características próprias (Arsanjani et al., 2007).

3.4.5 Serviços Web

Para que os conceitos da SOA possam ser implementados, tecnologias de desenvolvimento vêm sendo desenvolvidas e utilizadas. O surgimento de serviço web (do inglês, *Web Service*) e padrões de apoio à integração de negócios automatizados tem impulsionado grandes avanços tecnológicos na integração de sistemas de software (Papazoglou e Heuvel, 2007). Segundo Papazoglou et al. (2008) e Erl (2005), serviço web é a tecnologia mais utilizada para implementar sistemas orientados a serviço, proporcionando difusão de funcionalidades a diversas aplicações por meio da internet. Serviços web são aplicações com baixo grau de acoplamento, fornecidas na forma de funcionalidades que podem ser facilmente compostas por meio da criação de processos de negócio (Papazoglou e Heuvel, 2007).

Essa tecnologia provê a base para o desenvolvimento e execução de processos de negócio distribuídos por meio da internet, fazendo uso de padrões abertos de comunicação. Os padrões utilizados na comunicação entre serviços web são baseados na linguagem XML (*eXtensible Markup Language*) (W3C, 2010a). A escolha da XML proporciona independência de plataforma e de linguagem de programação, promovendo maior interoperabilidade entre aplicações que operam na forma de serviço sob a web (Papazoglou et al., 2008). Basicamente, serviços web utilizam três padrões para viabilizar a comunicação, publicação e descoberta de serviços. Esses padrões são apresentados a seguir:

SOAP (W3C, 2010b): O *Simple Object Access Protocol* (SOAP) é o protocolo mais utilizado por serviços web, sendo a estrutura em XML na qual todas a mensagens são construídas. Por meio do SOAP é especificado o formato das mensagens trocadas entre o serviço consumidor e o serviço provedor. Em particular, o SOAP descreve como chamadas de procedimentos remotos (do inglês, *Remote Procedure Call*) podem ser realizadas utilizando-se o protocolo HTTP (*HyperText Transfer Protocol*);

WSDL (W3C, 2010c): Todo serviço deve possuir uma descrição que forneça ao seu potencial consumidor informações de seu funcionamento e acesso. O protocolo *Web Service Description Language* (WSDL) provê todas as informações necessárias à invocação de um serviço. Essas descrições incluem detalhes como: definição de tipos de dados, operações apoiadas pelo serviço e formatos das mensagens de entrada e saída. Para que a comunicação seja realizada, é necessária a transformação de especificações WSDL em protocolos concretos como, por exemplo, o SOAP. De-

senvolvedores podem criar, assim, serviços web consumidores sem necessariamente conhecerem os serviços provedores em questão; e

UDDI (OASIS, 2010a): O *Universal, Discovery, Description and Integration* (UDDI) é um protocolo pelo qual consumidores podem encontrar serviços desejados em um registro público. Além de descrever os serviços web, ou seja, especificar quais são as interfaces utilizadas e tipos de valores que um serviço possui, o UDDI permite também que sejam descritas suas finalidades. De uma maneira intuitiva, UDDI funciona como uma espécie de "páginas amarelas" nas quais serviços são oferecidos e pesquisados por interessados.

Esses três padrões implementam a estrutura da SOA, tornando possível a interação indireta entre provedores de serviço e seus consumidores. No contexto de serviços web, a solicitação de um serviço é feita requisitando-se uma funcionalidade ao registro de serviços, tendo como parâmetros informações contidas no protocolo UDDI. De posse dessas informações, o registro de serviços recorre à sua base de dados contendo descrições dos provedores e retorna ao cliente o endereço (*endpoint*) do serviço encontrado, para que eles possam se comunicar. Após estabelecer a conexão, o serviço cliente solicita informações referentes à descrição daquele serviço para que possam interagir corretamente. O provedor, por meio de uma resposta escrita em WSDL, fornece as direções necessárias para que eles possam interagir corretamente. Em seguida, como base nas informações contidas no WSDL, o consumidor de um serviço solicita funcionalidades diretamente ao serviço provedor. Para que serviços estejam disponíveis no repositório do registro, é necessário que os provedores publiquem descrições referentes às funcionalidades e condições de uso por meio do seu UDDI e WSDL.

Além dos padrões utilizados na interação entre serviços, chamados de padrões de primeira geração, serviços web também proveem um conjunto de padrões denominados como padrões de segunda geração (Erl, 2005; Weerawarana et al., 2005). Em especial, dentre os padrões de segunda geração, duas linguagens são dedicadas à descrição das colaborações entre os participantes de uma composição de serviço. As linguagens detalhadas a seguir viabilizam, respectivamente, a aplicação dos conceitos de orquestração e de coreografia de serviço:

WS-BPEL (OASIS, 2010b): Conceitualmente, a *Web Services Business Process Execution Language* (WS-BPEL) define uma linguagem para a composição de serviços na forma de processos de negócio (Curbera et al., 2003). A WS-BPEL oferece suporte tanto para processos abstratos quanto para processos executáveis (Erl, 2005; Peltz, 2003). Um processo abstrato, ou protocolo de negócio, especifica a troca de mensagens públicas entre os serviços pertencentes a um processo. Processos executáveis,

por sua vez, modelam o comportamento dos serviços participantes em interações de negócio específicas. A WS-BPEL possui elementos de linguagem que lhe permitem especificar chamadas de serviços, responder a processos, manipular variáveis e estruturas, tratar dados, gerenciar eventos e exceções (Erl, 2005). Um processo de negócio que utiliza WS-BPEL também é considerado um serviço e, em virtude disso, necessita de um arquivo WSDL que descreva sua interface; e

WS-CDL (W3C, 2010d): A *Web Services Choreography Description Language* (WS-CDL) é uma linguagem baseada em XML que descreve a colaboração par-a-par entre serviços. Assim como no conceito de coreografia, essa linguagem define um ponto de vista global (não centralizado), os comportamentos comuns e complementares dos serviços participantes, ordenando resultados de trocas de mensagens de forma a alcançar um objetivo comum de negócio. A WS-CDL oferece uma ponte de comunicação par-a-par entre ambientes de desenvolvimento heterogêneos.

3.4.6 Arquiteturas e Modelos de Referência Orientados a Serviço

Como já foi discutido anteriormente, a utilização da SOA pode possibilitar o desenvolvimento de sistemas fracamente acoplados, reutilizáveis e extensíveis. A utilização desse estilo arquitetural pode viabilizar a rápida produção e integração entre organizações e aplicações isoladas. Todavia, segundo Arsanjani et al. (2007), a construção de sistemas orientados a serviço é uma tarefa complexa. Por isso, ainda existem grandes desafios quanto à criação desses sistemas de forma adequada e efetiva (Arsanjani et al., 2007). Para que diferentes aplicações, de empresas distintas, possam cooperar de maneira satisfatória, é necessário que arquitetos desenvolvam abordagens consistentes de produção, utilizando notações bem definidas e organizando soluções arquiteturais que ilustrem as capacidades de cada sistema e o relacionamento entre as partes que os compõem (Arsanjani et al., 2007).

Com a carência de padronização e de direções ao desenvolvimento de sistemas baseados na SOA, arquiteturas e modelos de referência orientados a serviço vêm sendo propostos e utilizados (Arsanjani et al., 2007; Costagliola et al., 2006; Hemalatha et al., 2008; OASIS, 2006, 2008; Peristeras et al., 2009). Iniciativas como as arquiteturas de referência S3 (Arsanjani et al., 2007), desenvolvida pela IBM, e a arquitetura de referência proposta pela OASIS (2008) visam guiar o projeto de arquiteturas concretas e sistemas baseados na SOA. A arquitetura de referência S3 provê uma descrição arquitetural da SOA por meio de uma estrutura dividida em camadas, que apresentam conceitos relacionados a serviço, mapeados em aspectos tecnológicos. Já a arquitetura de referência proposta pela OASIS objetiva definir vocabulário e entendimento comuns a respeito dos elementos e

interações em SOA, de forma independente de tecnologias, protocolos e produtos relacionados à implementação. Além dessas arquiteturas mais genéricas, que possuem alto nível de abstração e podem ser aplicadas em diferentes domínios, diversas outras arquiteturas de referência destinadas a domínios específicos podem ser encontradas (Costagliola et al., 2006; Fioravanti et al., 2007; Murakami et al., 2007; Peristeras et al., 2009). Essas arquiteturas relacionam informações referentes ao domínio de aplicação como, por exemplo, *e-learning* (Costagliola et al., 2006) e *e-working* (Peristeras et al., 2009), aos conceitos de serviço, fornecendo direções mais precisas ao desenvolvimento de sistemas. Com o objetivo de investigar de maneira criteriosa trabalhos relacionados aos modelos e arquiteturas de referência orientados a serviço, durante o desenvolvimento deste projeto, foi conduzida uma revisão sistemática (Oliveira et al., 2010; Oliveira e Nakagawa, 2010).

Revisões sistemáticas têm como objetivo apresentar uma avaliação justa a respeito de um tópico de pesquisa, fazendo uso de uma metodologia de revisão que seja confiável, rigorosa e que permita auditagem (Kitchenham, 2004). A revisão sistemática é uma abordagem sistêmica de revisão da literatura. Assim, para sua condução, é utilizado um conjunto de passos bem definidos e planejados de acordo com um protocolo previamente estabelecido (Biolchini et al., 2005). O planejamento de uma revisão sistemática define e descreve, entre outras informações, critérios de inclusão e exclusão, a metodologia para a extração de dados, *strings* de busca, critérios de seleção e questões que se pretende responder após sua condução. Por meio da revisão sistemática realizada foram identificados quais são os domínios de aplicação nos quais modelos e arquiteturas de referência têm sido utilizados, bem como as características da SOA que vêm sendo consideradas nesse nível de abstração. Além disso, foram identificados sobre quais insumos arquiteturas e modelos de referência vêm sendo desenvolvidos. É importante destacar que, entre os trabalhos disponíveis na literatura, não foram encontrados modelos e arquiteturas de referência que oferecessem suporte ao desenvolvimento de ferramentas de teste que tenham como base o estilo arquitetural SOA. Uma vez que as informações obtidas na revisão sistemática fazem parte da investigação das fontes de informação utilizadas no estabelecimento da arquitetura proposta neste trabalho, maiores detalhes sobre as arquiteturas e modelos de referência analisados são apresentados na Seção 4.2.

3.5 Considerações Finais

Neste capítulo foram apresentados importantes tópicos pertencentes à área de Arquitetura de Software, úteis no contexto deste trabalho. Em particular, foram apresentados conceitos, classificações e exemplos de arquiteturas de referência. Ademais, foram discutidos conceitos, características e tecnologias relacionadas ao estilo arquitetural SOA. Especial

atenção foi direcionada às arquiteturas de referência orientadas a serviço. É importante destacar que não foram encontradas na literatura arquiteturas de referência destinadas a apoiar o desenvolvimento de ferramentas de teste de software que tenham a SOA como base. Dessa forma, visando contribuir para o desenvolvimento dessas ferramentas, no próximo capítulo são apresentados detalhes acerca do estabelecimento de uma arquitetura de referência orientada a serviço, chamada de RefTEST-SOA.

Capítulo

4

Estabelecimento da RefTEST-SOA

4.1 Considerações Iniciais

Neste capítulo é apresentada uma arquitetura de referência orientada a serviço para ferramentas de teste de software, denominada RefTEST-SOA (*Reference Architecture for Testing Tools Based on SOA*). A arquitetura de referência proposta tem por objetivo apoiar a fase de projeto arquitetural no processo de desenvolvimento de ferramentas de teste de software que tenham como base o estilo arquitetural SOA.

Para o estabelecimento da RefTEST-SOA foi utilizado o ProSA-RA, um processo que provê um conjunto de quatro passos que guiam o desenvolvimento de arquiteturas de referência, desde a investigação das fontes de informação do domínio, utilizadas na definição dos requisitos arquiteturais, até a fase de avaliação arquitetural. Visando melhor descrever a arquitetura sendo proposta são utilizadas três visões arquiteturais: estrutural (ou visão de módulos); comportamental (ou visão em tempo de execução); e física (ou visão de implantação). Além das visões arquiteturais é apresentada também uma visão geral da RefTEST-SOA, que oferece uma representação de alto nível dos elementos presentes na arquitetura de referência.

Este capítulo está organizado de acordo com os passos definidos pelo ProSA-RA. Na Seção 4.2 são descritas as fontes de informações utilizadas na identificação dos requisitos arquiteturais da RefTEST-SOA. Na Seção 4.3 é apresentado o conjunto de requisitos arquiteturais identificados por meio da análise das fontes de informação. O projeto da

RefTEST-SOA, descrito por uma representação geral e três visões arquiteturais, é apresentado na Seção 4.4. Na Seção 4.5 são discutidas informações referentes à avaliação da arquitetura proposta. Por fim, na Seção 4.6 são apresentadas as considerações finais deste capítulo.

4.2 Passo RA-1: Investigação das Fontes de Informação

Durante o primeiro passo do ProSA-RA, denominado "Investigação das Fontes de Informação", foram identificadas diferentes fontes para obtenção de informações sobre o domínio de teste de software e também conceitos referentes à SOA. Para essa atividade, foram considerados os seguintes grupos de fontes de informação: (i) arquiteturas concretas e ferramentas orientadas a serviço de teste, de verificação e de análise de software; (ii) diretrizes para o desenvolvimento de sistemas orientados a serviço; (iii) arquiteturas de referência orientadas a serviço projetadas para diferentes domínios; e (iv) arquiteturas de referência para o domínio de teste disponíveis na literatura. Entre as arquiteturas de referência para o domínio de teste, especial atenção é destinada à RefTEST, uma vez que essa constitui a base para o desenvolvimento deste trabalho. Além disso, vale salientar que, até onde se tem conhecimento, a literatura dispõe de duas arquiteturas de referência de ferramentas de teste de software (Eickelmann e Richardson, 1996; Nakagawa et al., 2007), sendo a RefTEST, proposta por Nakagawa et al. (2007), mais atual e melhor documentada. A seguir, são discutidos os detalhes a respeito das fontes de informação utilizadas no estabelecimento da arquitetura de referência proposta. É importante ressaltar que as fontes de informação são descritas detalhadamente por constituírem um parte fundamental do processo de estabelecimento da RefTEST-SOA.

4.2.1 Grupo 1: Arquiteturas e Ferramentas Orientadas a Serviço de Teste, de Verificação e de Análise de Software

Uma importante fonte de informação é representada pelo conjunto de ferramentas de teste, de verificação e de análise de software disponíveis como serviços. Foram considerados também, trabalhos relacionados à integração, comunicação e descoberta de serviços de tais ferramentas. Os trabalhos pertencentes a esse grupo de fontes de informação são apresentados na Tabela 4.1.

Bartolini et al. (2008) propõem uma ferramenta, denominada TCov, destinada ao teste estrutural de serviços web. Tal ferramenta, disponível como serviço, faz uso de uma abordagem de análise de cobertura dos testes denominada SOCT (*Service-Oriented Coverage Testing*). Nessa abordagem, o desenvolvedor deve instrumentar manualmente

Tabela 4.1: Breve descrição de arquiteturas e ferramentas orientadas a serviço de teste, de verificação e de análise de software

Breve descrição	Trabalho
Ferramenta de teste estrutural	Bartolini et al. (2008)
Ferramenta de teste estrutural	Eler et al. (2009)
Registro de serviços para ferramentas de teste de software	Gondim (2010)
Plataforma para a disponibilização de ferramentas de análise de software como serviços	Ghezzi e Gall (2008)
Ambiente de integração para serviços de verificação	Baldamusa et al. (2004)
Abordagem para descoberta, integração e invocação de serviços de engenharia de software	Yap et al. (2005)

serviços ou composições de serviços (definidas em linguagem WS-BPEL) a serem testados, incluindo chamadas para o serviço TCov, de modo que esse possa registrar os caminhos executados pelo programa em teste. Essa tarefa não pode ser automatizada, uma vez que a ferramenta TCOV é destinada ao teste estrutural de serviços web e não possui acesso direto ao código fonte. Além da instrumentação manual, o cliente ou integrador deve coletar manualmente os dados obtidos das chamadas ao serviço ou implementar uma ferramenta para realizar tal coleta. Apesar de oferecer informações sobre o funcionamento de uma ferramenta de teste disponível como serviço, não foram discutidos no trabalho de Bartolini et al. (2008) detalhes sobre a arquitetura da ferramenta proposta.

Ao contrário da abordagem de Bartolini et al. (2008), que tem por objetivo testar composições de serviços definidas na linguagem WS-BPEL, a ferramenta de teste proposta por Eler et al. (2009) apoia o teste estrutural de programas desenvolvidos na linguagem Java ou em AspectJ, incluindo serviços web. Essa ferramenta, denominada *JaBUTi*Service, é um único serviço baseado na ferramenta de teste *JaBUTi* (Vincenzi, 2004), a qual pode ser utilizada apenas de forma *stand-alone*. Para que o *JaBUTi*Service pudesse ser utilizado ou integrado a um *workflow* de teste, as interfaces gráficas com o usuário (do inglês, *Graphical User Interface* – GUI) foram substituídas por uma interface de serviço padrão, descrita em WSDL. Na Figura 4.1 é ilustrada a arquitetura do *JaBUTi*Service. Segundo sua arquitetura, o serviço é composto por quatro principais elementos, a saber: uma *engine* para envio e recebimento de mensagens entre serviços clientes e provedores (*Axis2 engine*(Apache Foundation, 2010)); um controlador para o tratamento de eventos; um banco de dados para o armazenamento de projetos de teste; e o núcleo de funcionalidades provenientes da *JaBUTi*, responsável pela instrumentação e análise de cobertura do programa em teste.

A disponibilização da ferramenta *JaBUTi* na forma de um serviço web tem como uma de suas principais vantagens o aumento do reúso, uma vez que suas funcionalidades podem ser utilizadas por diferentes composições, formando funcionalidades mais completas, por meio de um *workflow* de teste (Eler et al., 2009). Contudo, é importante salientar

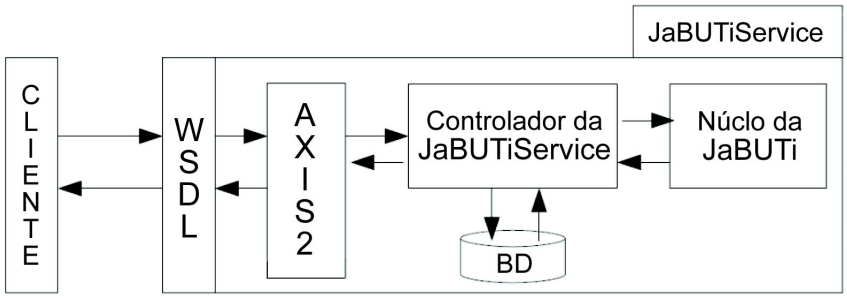

Figura 4.1: Arquitetura da ferramenta *JaBUTi*Service (Eler et al., 2009)

que apesar da arquitetura proposta viabilizar o reúso da ferramenta como um todo, sua estrutura não permite que funcionalidades providas por seus subsistemas — como por exemplo, o de gerenciamento de casos de teste — possam ser reutilizadas de maneira independente por outras ferramentas de teste.

Visando apoiar a publicação, descoberta e classificação de ferramentas de teste de software disponíveis segundo o estilo arquitetural SOA, o trabalho de Gondim (2010) propõe um repositório de serviços específico para o domínio de teste. Esse repositório faz uso de ontologias de teste (Bai et al., 2008; Barbosa et al., 2006) com o objetivo de aprimorar a identificação dos serviços adequados a cada consulta, aumentado assim a chance de que um serviço atenda às necessidades do cliente. No trabalho de Gondim (2010), além do portal destinado ao gerenciamento de serviços de teste, é apresentada também a especificação e a arquitetura referentes ao repositório proposto.

Em uma abordagem similar à proposta por Gondim (2010), na qual é desenvolvido um repositório que visa apoiar a publicação, descoberta e classificação de serviços de um domínio específico, Ghezzi e Gall (2008) propõem uma plataforma para a disponibilização de ferramentas de análise de software segundo o estilo arquitetural SOA. Nessa abordagem, a classificação das funcionalidades disponíveis é realizada com o auxílio de taxonomias e as entradas e saídas de cada funcionalidade implementada são padronizadas por meio de ontologias. Tal plataforma provê mecanismos para a seleção e composição de serviços em *workflows*.

Baldamusa et al. (2004) propõem a integração de ferramentas de verificação de software disponíveis como serviços. Nessa abordagem, usuários são capazes de especificar, por meio de um mecanismo de verificação de *scripts*, quais subtarefas de verificação desejam executar. Além disso, são utilizados repositórios para a publicação e descoberta de serviços de verificação. Na abordagem proposta por Baldamusa et al. (2004), serviços compostos são considerados como novos serviços ou como *workflows*, que realizam chamadas para serviços de verificação.

Em um contexto mais geral, Yap et al. (2005) propõem uma abordagem para a descoberta, integração e invocação dinâmica de ferramentas de engenharia de software utili-

zando serviços. Essa abordagem faz uso da ferramenta JEdit[1], disponível como software livre, para apoiar a interação entre diferentes ferramentas providas como serviços. Nessa abordagem, a composição de serviços não resulta em um *workflow* e sim em uma ferramenta *desktop*.

A análise das arquiteturas concretas e ferramentas de teste, de verificação e de análise de software é de grande importância, pois evidencia quais abordagens e tecnologias vêm sendo utilizadas durante o desenvolvimento desses tipos de sistema. Vale ressaltar que, em virtude da utilização da SOA no desenvolvimento de ferramentas de engenharia de software ser recente, poucos trabalhos relacionados ao desenvolvimento de ferramentas de teste disponíveis como serviços podem ser encontrados na literatura. Há também uma carência de arquiteturas de referência para o domínio de teste de software que sejam adequadas ao desenvolvimento de ferramentas segundo a SOA.

4.2.2 Grupo 2: Diretrizes para o Desenvolvimento de Sistemas Orientados a Serviço

Com o objetivo de nortear o desenvolvimento de sistemas baseados na SOA e prover entendimento e vocabulário comuns, modelos e arquiteturas de referência orientados a serviço independentes de domínios específicos podem ser encontrados. Na Tabela 4.2 são apresentados os trabalhos (T) encontrados na literatura que propõem modelos e arquiteturas de referência, visando prover direções ao desenvolvimento de sistemas orientados a serviço.

Tabela 4.2: Modelos e arquiteturas de referência orientados a serviço

T	Autor/Ano	Tipo
T1	Arsanjani et al. (2007)	Arquitetura de Referência
T2	Dillon et al. (2007)	Arquitetura de Referência
T3	Lan et al. (2008)	Arquitetura de Referência
T4	OASIS (2006)	Modelo de Referência
T5	OASIS (2008)	Arquitetura de Referência
T6	Zimmermann et al. (2009)	Arquitetura de Referência

A arquitetura de referência S3 (Arsanjani et al., 2007), proposta pela IBM, provê uma descrição arquitetural da SOA em uma estrutura composta por nove camadas, conforme ilustrada na Figura 4.2. Nessa arquitetura de referência, a camada de *sistemas operacionais* (Camada 1) contempla toda a infraestrutura que da apoio às atividades de negócio, tais como: pacotes, sistemas existentes e bancos de dados. A camada de *componente de serviço* (Camada 2) contém componentes de software, que representam realizações de

[1] `http://www.jedit.org/`

serviços ou operações. Na camada de *serviço* (Camada 3) são representados todos os serviços disponíveis. Por sua vez, na camada de *processo de negócio* (Camada 4) são realizadas as composições dos serviços disponíveis na camada de serviço, por meio do uso de coreografia e/ou orquestração. Na camada *consumidor* (Camada 5) são gerenciadas as interações com os usuários e com outros serviços. A camada de *integração* (Camada 6) une as camadas de 2 a 4, mediando, roteando e transportando requisições de serviço entre provedores e consumidores. As capacidades dessa camada são similares as de um ESB. A camada de *qualidade de serviço* (Camada 7) é responsável por inspecionar o cumprimento de acordos relacionados às características de qualidade, notificando problemas às camadas básicas (camadas de 2 a 4). A garantia de que a organização inclua considerações-chave referentes aos dados e informações arquiteturais é realizada na camada de *arquitetura de informação* (Camada 8). Por fim, a camada de *políticas & governança* (Camada 9) apoia todos os aspectos relacionados ao gerenciamento do ciclo de vida de operações de negócio, provendo direções e políticas para o gerenciamento de contratos em nível de serviço, considerando características como segurança e desempenho.

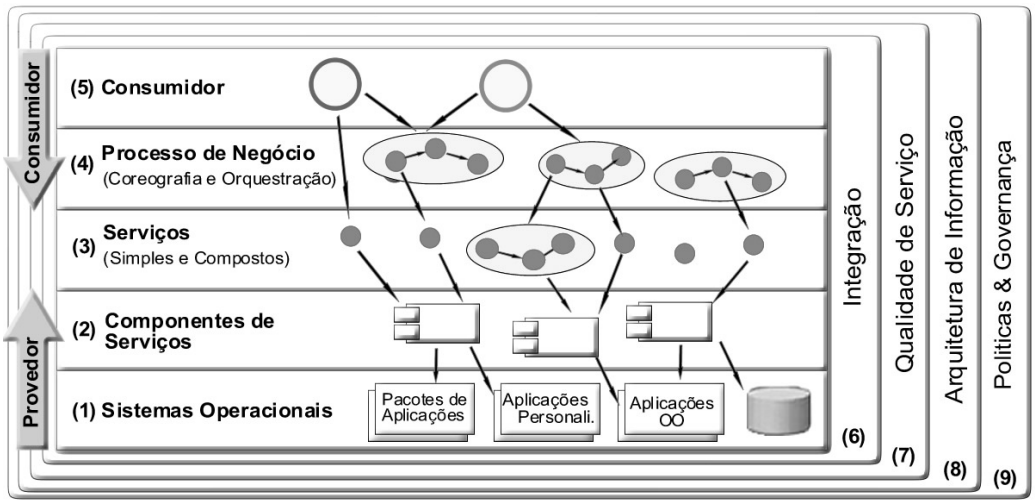

Figura 4.2: Arquitetura de referência S3 (adaptado de Arsanjani et al. (2007))

Outra arquitetura proposta pela IBM é a SOAI RA (Zimmermann et al., 2009), que é destinada a facilitar o desenvolvimento de projetos de linhas de produto implementadas por meio de serviços. A arquitetura de referência é representada segundo três pontos de vista (*viewpoint*): Lógico (*Logical Viewpoint*), Físico (*Physical Viewpoint*) e de cenário (*Scenario Viewpoint*). O ponto de vista lógico é descrito pelo modelo de componentes lógicos (*Logical Component Model*). O ponto de vista físico é descrito pelo modelo operacional físico (*Physical Operational Model*). Já a visão de cenário concentra-se em modelos de casos de uso (*Use Case Model*), requisitos não-funcionais e diagramas de contexto do sistema. Na arquitetura proposta, os três pontos de vista utilizados são unidos por decisões arquiteturais. É importante lembrar que a SOAI RA faz uso da arquitetura de

referência S3 como parte de seus diagramas, o que fornece indícios da aplicabilidade de tal arquitetura.

Lan et al. (2008) introduzem uma arquitetura de referência com uma estrutura similar à proposta na S3. Contudo, um contexto mais abrangente denominado como Modelo de Solução (do inglês, *Solution Model*) é utilizado. Um Modelo de Solução para a SOA envolve a definição de uma arquitetura de referência (denominada SOAII, ilustrada na Figura 4.3), padrões, metodologias e boas práticas de desenvolvimento. A arquitetura de referência proposta é composta por seis módulos, apoiados sobre implementações e artefatos de sistemas já desenvolvidos. Os módulos considerados na representação da SOAII são: Acesso a Dados e Aplicações; Orquestração de Processo; Composição de Aplicações; Barramento de Serviços; Ferramentas de Desenvolvimento; e Governança de Serviços e Monitoramento.

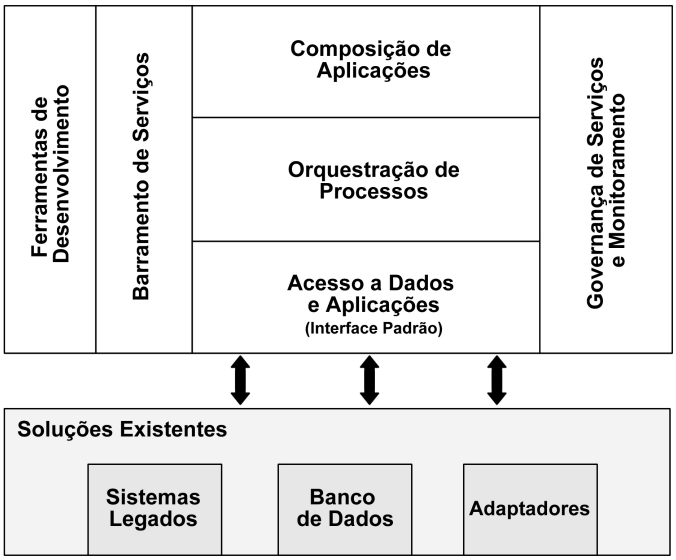

Figura 4.3: Arquitetura de referência SOAII (adaptado de Lan et al. (2008))

O modelo de referência proposto pela OASIS (OASIS, 2006) define a essência da SOA fornecendo vocabulário e entendimento comuns sobre esse estilo arquitetural. Nesse modelo de referência são definidos os relacionamentos entre conceitos pertencentes ao estilo arquitetural SOA, tais como: visibilidade, descrição de serviços, políticas, contratos, formas de interação e contextos de execução. Baseada nos conceitos definidos em seu modelo de referência e no estilo arquitetural SOA, a OASIS propõe também uma arquitetura de referência (OASIS, 2008). Essa arquitetura de referência visa prover direções ao desenvolvimento de sistemas baseados na SOA de forma independente de tecnologia, protocolos e produtos relacionados à implementação. Para isso, o detalhamento da arquitetura é feito por meio do uso da UML, segundo três diferentes pontos de vista: Negócios por meio de Serviços (*Business via Services*); Concretizando Arquiteturas Orientadas a Serviço

(*Realizing Service-Oriented Architectures*); e Posse em Arquiteturas Orientadas a Serviço (*Owning Service-Oriented Architecture*). Uma vez que essa arquitetura não possui uma visão geral, sendo detalhada por um grande conjunto de diagramas, maiores informações relacionadas a sua estrutura podem ser encontrados em (OASIS, 2008).

O trabalho de Dillon et al. (2007) apresenta um estudo sobre diferentes implementações do estilo arquitetural SOA. Com base nessas informações, é proposta uma arquitetura de referência orientada a serviço. Apesar do trabalho apresentar uma arquitetura de referência para a construção de sistemas implementados segundo a SOA, não existem maiores detalhes acerca de sua utilização, tampouco são apresentadas adequada documentação ou detalhamento por diferentes visões arquiteturais. Em virtude disso, poucas informações provenientes dessa arquitetura podem ser utilizadas na elicitação de requisitos arquiteturais da arquitetura de referência proposta neste trabalho.

Como resultado da investigação das arquiteturas e modelos de referência, diretrizes para o desenvolvimento de serviços foram identificadas e são apresentadas na Tabela 4.3. Na Coluna 1 (Funcionalidade identificada) são listadas as funcionalidades identificadas nos trabalhos analisados e na Coluna 2 (Fonte) são indicadas quais fontes de informação contribuíram para essa identificação[2]. Na Coluna 3 (Conceito Relacionado – CR) são apresentados os conceitos da SOA relacionados a essas funcionalidades. Os conceitos identificados foram: Descrição de Serviço (DS), Publicação de Serviço (PS), Interação entre Serviços (IS), Composição de Serviços (CS), Políticas (PO), Governança (GO) e Qualidade de Serviço (QoS).

Tabela 4.3: Funcionalidades de sistemas orientados a serviço e seus conceitos relacionados

Funcionalidade identificada	Fonte	CR
Armazenar e tornar disponíveis informações de como interagir com um serviço, suas estruturas e tipos de dados	T1, T4, T5	DS
Armazenar e tornar disponíveis modelos semânticos destinados a categorização e classificação de um serviço	T4, T5	DS
Armazenar e tornar disponíveis políticas e condições de uso de um serviço	T1, T4, T5	DS
Armazenar e tornar disponíveis métricas que descrevam as características operacionais de um serviço	T1, T4, T5	DS
Criar as descrições de serviços de forma padronizada e estruturada	T1, T4, T5	PS
Publicar descrições de serviços diretamente aos consumidores	T4, T5	PS

continua na próxima página

[2]Para indicar as fontes de informação, foram utilizados os identificadores de T1 a T6 referentes aos modelos e arquiteturas de referência listados na Tabela 4.2.

Tabela4.3 (continuação)

Funcionalidade identificada	Fonte	CR
Publicar descrições de serviços por meio de mediadores	T1, T2, T4, T5	PS
Descobrir descrições de serviços utilizando linguagens padronizadas	T1, T2, T4, T5	PS
Notificar atualizações ou alterações nas descrições de um serviço	T4, T5	PS
Verificar por meio da inspeção de funcionalidades e capacidades se um serviço provedor atende aos requisitos de um serviço cliente	T1, T4, T5	PS
Verificar por meio da inspeção de políticas e contratos se um serviço provedor atende aos requisitos de um serviço cliente	T4, T5	PS
Interagir com outros serviços diretamente	T2, T4, T5	IS
Interagir com outros serviços por meio de um barramento de serviços	T1, T3	IS
Solicitar serviços atômicos e compostos de maneira uniforme	T1, T4, T5	IS
Processar e executar a solicitação de uma funcionalidade feita por outro serviço ou aplicação	T1, T2, T3, T4, T5	IS
Responder, por meio de mensagens padronizadas, a uma solicitação de funcionalidade	T1, T2, T4, T5	IS
Solicitar funcionalidades de um serviço provedor por meio de trocas de mensagem padronizadas	T1, T2, T4, T5	IS
Organizar a interação entre serviços de uma composição por meio de processos de negócio	T1, T2, T3, T4, T5	CS
Compor, hierarquicamente, serviços em processos de negócio por meio de orquestração	T1, T3, T4, T5	CS
Coordenar processos de negócio por meio de orquestração	T1, T3, T4, T5	CS
Organizar a colaboração e a troca de mensagens entre serviços por meio de coreografia	T1, T4, T5	CS
Expressar restrições por meio de assertivas	T1, T5, T6	PO
Expressar permissões e obrigações por meio de políticas	T1, T5, T6	PO
Compor políticas para representar restrições impostas por um ou mais serviços	T5, T6	PO
Pesquisar políticas que melhor atendam aos requisitos do serviço consumidor	T5, T6	PO
Auditar métricas provenientes de restrições e obrigações impostas por contratos	T1, T2, T3, T4, T5	PO
Armazenar políticas representadas por meio de contratos	T3, T5, T6	PO
Disponibilizar políticas por meio de contratos	T3, T5, T6	PO
Disponibilizar informações sobre aspectos de governança de um serviço	T1, T3, T5, T6	GO

continua na próxima página

Tabela4.3 (continuação)

Funcionalidade identificada	Fonte	CR
Informar aos serviços clientes sobre eventos de governança como, alterações em políticas, regras e regulamentos	T5, T6	GO
Acessar automaticamente processos de governança	T5, T6	GO
Acessar regras e regulamentos e suas informações	T1, T5, T6	GO
Habilitar regras e regulamentos de forma processável	T1, T5, T6	GO
Descobrir e pesquisar regras e regulamentos especificados por um serviço	T5, T6	GO
Capturar, recuperar e armazenar *logs* de interação e contextos de execução	T1, T2, T5, T6	QoS
Definir e aferir métricas de conformidade	T2, T5, T6	QoS
Tornar acessíveis informações utilizadas por métricas de qualidade	T5, T6	QoS
Capturar, monitorar, registrar e sinalizar o não cumprimento de requisitos de qualidade	T1, T5, T6	QoS

Os conceitos apresentados representam as direções normativas para a construção de sistemas orientados a serviço e, por conseguinte, servem como base para estabelecimento de arquiteturas de referência destinadas aos domínios nos quais tais sistemas são desenvolvidos. Dessa forma, grande parte das funcionalidades identificadas e seus respectivos conceitos foram considerados durante a definição dos requisitos arquiteturais da arquitetura de referência sendo proposta. Na próxima seção são analisados quais dos conceitos identificados vêm sendo efetivamente considerados em arquiteturas de referência destinadas a domínios específicos. Essa análise visa identificar quais desses conceitos podem vir a ser considerados durante o projeto arquitetural da RefTEST-SOA.

4.2.3 Grupo 3: Arquiteturas de Referência Orientadas a Serviço

Além das diretrizes ao desenvolvimento de sistemas orientados a serviço, apresentadas na seção anterior, arquiteturas de referência baseadas na SOA destinadas a domínios específicos podem ser encontradas. Como forma de investigar em quais domínios arquiteturas de referência orientadas a serviço vêm sendo utilizadas, bem como identificar quais os conceitos relacionados à SOA – discutidos na Seção 4.2.2 – têm sido considerados efetivamente por essas arquiteturas, uma revisão sistemática sobre o tema foi realizada (Oliveira e Nakagawa, 2010).

Na Tabela 4.4 são apresentadas as arquiteturas de referência identificadas na revisão sistemática e uma breve descrição do objetivo de cada um desses trabalhos. Maiores informações relacionadas a essa revisão sistemática, tais como o protocolo, os detalhes da

condução e a sumarização dos resultados podem ser encontrados em (Oliveira e Nakagawa, 2010).

Tabela 4.4: Breve descrição dos objetivos das arquiteturas de referência identificadas

Artigo	Breve descrição
Brehm e Gómez (2007)	Estruturação dos subsistemas de um sistema ERP na forma de serviços
Choi et al. (2009)	Integração de informações sobre projetos de pesquisa e desenvolvimento (P&D) em um portal do NTIS[1]
Duro et al. (2005)	Interoperabilidade entre sistemas baseados na SOA para o monitoramento de solos via satélite
Fioravanti et al. (2007)	Discussão sobre a evolução de uma arquitetura de referência do domínio de multimídia para o contexto de serviços
Hemalatha et al. (2008)	Apoiar a disponibilização de ferramentas de processamento digital de imagens segundo a SOA
Leppaniemi et al. (2009)	Apoiar a interoperabilidade e o baixo acoplamento em sistemas de apoio à previsão de desastres naturais
Murakami et al. (2007)	Integração entre diferentes sistemas e recursos relacionados ao domínio de agricultura de precisão
Peristeras et al. (2009)	Apoiar a interoperabilidade em ambientes de trabalho colaborativo (*e-working*)
Ramanathan et al. (2008)	Reduzir custos operacionais em projetos de sistemas de telecomunicação
Reiff-Marganiec et al. (2008)	Dar apoio à interoperabilidade em ambientes de trabalho colaborativo (*e-working*)
Zheng et al. (2008)	Aumentar a interoperabilidade entre sistemas de gerenciamento de ensino e objetos de ensino por meio da SOA

[1] *National Technical Information Service*

Baseado na análise dos trabalhos obtidos pela revisão sistemática, foram identificados quais os conceitos relacionados à SOA vêm sendo considerados em arquiteturas de referência orientadas a serviço destinadas a domínios específicos. Na Tabela 4.5 é ilustrada a relação entre os trabalhos analisados e os conceitos por eles abordados.

Como resultado da análise das arquiteturas de referência, foi observado que os conceitos como Descrição e Interação de Serviços – DS e IS – são inerentes à SOA e, por isso, estão presentes em todas as arquiteturas de referência para domínios específicos, mesmo quando não ilustrados explicitamente. É possível observar também que a preocupação com a representação do conceito de Publicação de Serviços (PS) é considerada pela maioria dos trabalhos encontrados na literatura, sendo abordado por dez das 11 arquiteturas de referência. Além disso, a representação explícita do conceito de Composição de Serviços (CS) é abordada por grande parte das arquiteturas de referência. Já os conceitos de qualidade de serviço (QoS), políticas (PO) e governança (GO) foram considerados por menos de 40% das arquiteturas de referência.

Tabela 4.5: Conceitos relacionados a serviço abordados em arquiteturas de referência

Artigo	DS	PS	IS	CS	PO	GO	QoS
Brehm e Gómez (2007)	✓	✓	✓	–	✓	✓	✓
Choi et al. (2009)	✓	✓	✓	✓	–	–	–
Duro et al. (2005)	✓	✓	✓	✓	–	–	–
Fioravanti et al. (2007)	✓	✓	✓	–	–	–	–
Hemalatha et al. (2008)	✓	✓	✓	✓	–	–	–
Leppaniemi et al. (2009)	✓	–	✓	✓	–	–	–
Murakami et al. (2007)	✓	✓	✓	–	–	–	–
Peristeras et al. (2009)	✓	✓	✓	✓	–	–	✓
Ramanathan et al. (2008)	✓	✓	✓	✓	–	–	✓
Reiff-Marganiec et al. (2008)	✓	✓	✓	✓	✓	✓	✓
Zheng et al. (2008)	✓	✓	✓	✓	✓	✓	–
TOTAL	11	10	11	8	3	3	4

As informações obtidas pela análise desses trabalhos fornecem evidências de que os conceitos de Descrição de Serviço (DS), Interação entre Serviços (IS), Publicação de Serviço (PS) e Composição de Serviços (CS) devem ser considerados na evolução da RefTEST para o contexto da SOA, uma vez que esses já vêm sendo considerados em arquitetura de referência para outros domínios de aplicação. Entretanto, a pertinência da utilização dos demais conceitos relacionados à SOA deve ser avaliada de acordo com os requisitos estabelecidos no próximo passo do processo, podendo ou não serem considerados durante o projeto da arquitetura.

A condução de uma revisão sistemática possibilitou o estabelecimento de uma visão geral sobre a utilização de arquiteturas de referência orientadas a serviço e serviu como forma de apoio à obtenção das fontes de informação utilizadas na construção da arquitetura de referência proposta neste trabalho. Vale ressaltar que a aplicação de revisões sistemáticas como forma de auxílio à obtenção de fontes de informação, úteis à elicitação de requisitos, é uma prática que tem despertado interesse da comunidade científica (Dieste et al., 2007, 2008; Nakagawa e Oliveira, 2011).

4.2.4 Grupo 4: Arquiteturas de Referência para o Domínio de Teste

Entre as arquiteturas de referência encontradas na literatura, duas são destinadas ao domínio de teste: a proposta por Eickelmann e Richardson (1996) e a proposta por Nakagawa (2006). A arquitetura de referência proposta por Eickelmann e Richardson (1996) é constituída de seis funcionalidades básicas, identificadas com base na análise do processo de teste de software, na automatização desse processo e nas ferramentas de teste. Com base nessas funcionalidades, foi proposta a arquitetura de referência ilustrada na Figura 4.4.

Figura 4.4: Arquitetura proposta por Eickelmann e Richardson (1996)

Cada uma das funcionalidades representadas nas camadas da arquitetura é descrita a seguir (Eickelmann e Richardson, 1996):

Planejamento: é responsável pela avaliação de riscos, a necessidade de treinamento, os recursos necessários e disponíveis, as estratégias de teste, a alocação de responsabilidades e planejamento de cronogramas. Essa funcionalidade está relacionada ao desenvolvimento de um plano de teste e às características do sistema a ser testado;

Gerenciamento: oferece apoio à atividade de teste por meio do gerenciamento e armazenamento dos artefatos de teste, bem como do estado de execução do teste;

Medição: responsável pela análise e medição da cobertura do teste. Em geral, códigos fonte são instrumentados de forma a viabilizar a coleta do *trace* da execução. Artefatos resultantes dessa funcionalidade incluem medidas de cobertura e de falhas do teste;

Análise de Falha: é composta pela documentação, pela verificação do comportamento, bem como da análise estatística do teste. Essa funcionalidade é responsável por produzir relatórios de análise de falhas;

Desenvolvimento: responsável por auxiliar no desenvolvimento de artefatos, tais como os oráculos, casos de teste e critérios de teste. Essa funcionalidade tem por objetivo o desenvolvimento da configuração do teste, que define: como ele será conduzido, as entradas relacionadas aos artefatos de teste e à documentação relacionada a essa atividade; e

Execução: é responsável pela execução do programa em teste e o armazenamento dos resultados dessa execução. Como artefatos gerenciados nessa funcionalidade, estão incluídos os resultados dos testes, *trace* da execução e o andamento do teste.

Por outro lado, a arquitetura de referência RefTEST, proposta por Nakagawa et al. (2007), baseia-se na separação de interesses. Na RefTEST, os conceitos do domínio de teste e seus relacionamentos representam as funcionalidades desempenhadas pelas ferramentas de teste de software. Para que essas funcionalidades pudessem ser elicitadas foram investigadas informações provenientes de diferentes fontes, tais como: arquiteturas de software, processos de teste, ferramentas de teste e uma ontologia (Barbosa et al., 2006). A obtenção dos conceitos do domínio de teste foi realizada conforme ilustrado na Figura 4.5.

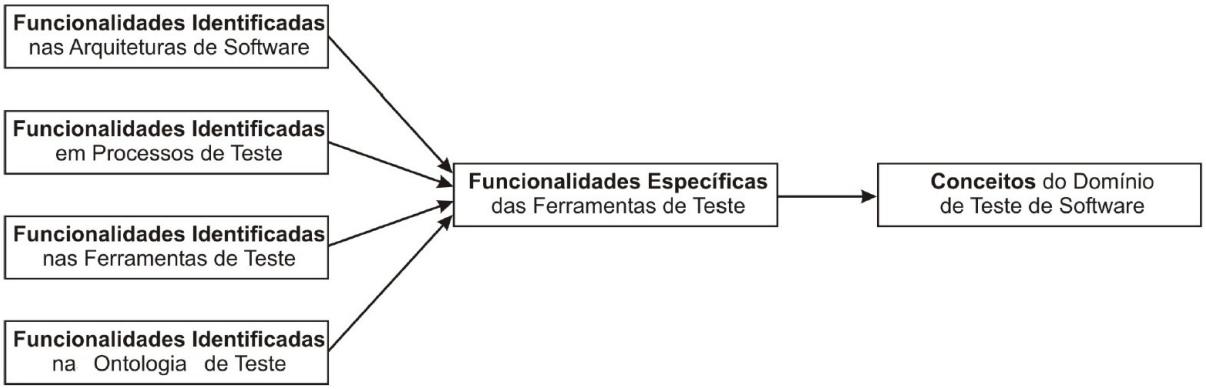

Figura 4.5: Mapeamento das funcionalidades específicas de ferramentas de teste presentes nas fontes de informação (Nakagawa, 2006)

Nessa abordagem, as funcionalidades específicas das ferramentas de teste são utilizadas na identificação dos conceitos do domínio, conforme apresentado na Tabela 4.6. Na primeira coluna da tabela são listadas as funcionalidades específicas, identificadas por meio da investigação das fontes de informação. Na segunda coluna são apresentados os conceitos relacionados a essas funcionalidades.

A arquitetura de referência RefTEST é descrita por diferentes visões arquiteturais, a saber (Nakagawa et al., 2007): visão de módulo (*Module View*), visão em tempo de execução (*Runtime View*), visão de implantação (*Deployment View*) e visão geral. Na Figura 4.6 é ilustrada a visão geral da RefTEST. Essa arquitetura de referência é baseada no padrão MVC e é dividida em três camadas: Apresentação, Aplicação e Persistência. A Camada de Apresentação é constituída dos componentes Visão, responsável por apresentar as informações aos usuários, e Controlador, que processa as requisições feitas por esses usuários. Na Camada de Aplicação são representadas as regras de negócio (*core*) das ferramentas de teste, de apoio, organizacionais e gerais, de uma maneira similar a

Tabela 4.6: Funcionalidades específicas e conceitos do domínio de testes (Nakagawa, 2006)

Funcionalidade Específica	Conceito
Adquire artefato a ser testado	Artefato em Teste
Trata o artefato a ser testado	Artefato em Teste
Executa o artefato a ser testado com casos de teste	Artefato em Teste
Fornece relatório de execução do artefato a ser testado	Artefato em Teste
Permite a visualização de artefato a ser testado	Artefato em Teste
Exclui artefato a ser testado	Artefato em Teste
Inclui *drivers* e *stubs*	Artefato em Teste
Permite visualizado de *drivers* e *stubs*	Artefato em Teste
Importa casos de teste	Casos de Teste
Exclui casos de teste manualmente	Casos de Teste
Gera casos de teste automaticamente	Casos de Teste
Minimiza conjunto de casos de teste	Casos de Teste
Habilita casos de teste	Casos de Teste
Desabilita casos de teste	Casos de Teste
Exporta casos de teste	Casos de Teste
Fornece relatórios de casos de teste	Casos de Teste
Remove casos de teste	Casos de Teste
Permite a visualização de casos de teste	Casos de Teste
Importa requisitos de teste	Requisito de teste
Gera requisitos de teste	Requisito de teste
Seleciona requisitos de teste	Requisito de teste
Marca requisito de teste como não-executável	Requisito de teste
Desmarca requisitos de teste	Requisito de teste
Executa requisitos de teste com casos de teste	Requisito de teste
Coleta *trace* de execução dos requisitos	Requisito de teste
Fornece relatório de requisitos de teste	Requisito de teste
Permite a visualização requisitos de teste	Requisito de teste
Exclui requisitos de teste	Requisito de teste
Estabelece critério de adequação do teste	Critério de teste
Calcula cobertura do teste	Critério de teste
Fornece relatório de falhas do teste	Critério de teste
Permite a análise estática da execução do teste pelo testador	Critério de teste

apresentada na Figura 3.4. Os dados resultantes da utilização das ferramentas de teste são armazenados na Camada de Persistência.

Na Figura 4.7 é apresentada a visão de módulos da RefTEST. De modo geral, o modelo objetiva especificar que a comunicação entre os módulos que implementam atividades de apoio (*supporting_crosscutting_frameworks*) e organizacionais (*organizational_crosscutting_frameworks*) seja realizada por meio do uso de aspectos, integrando-os ao módulo da ferramenta de teste (*testing_tool*) da arquitetura. De maneira análoga, os *frameworks* de propósito geral também se relacionam com os demais módulos da ferramenta.

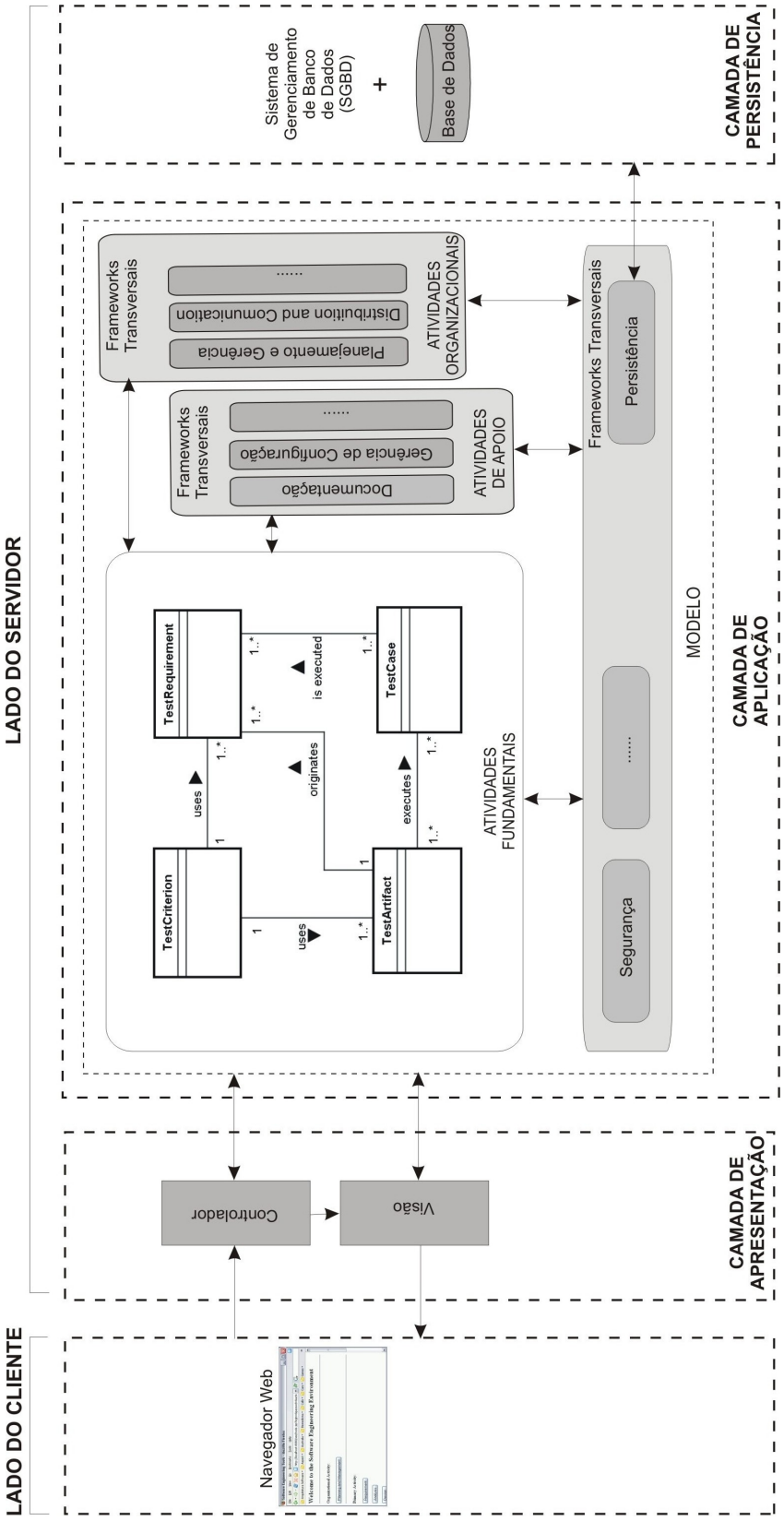

Figura 4.6: Visão Geral da RefTEST (Nakagawa, 2006)

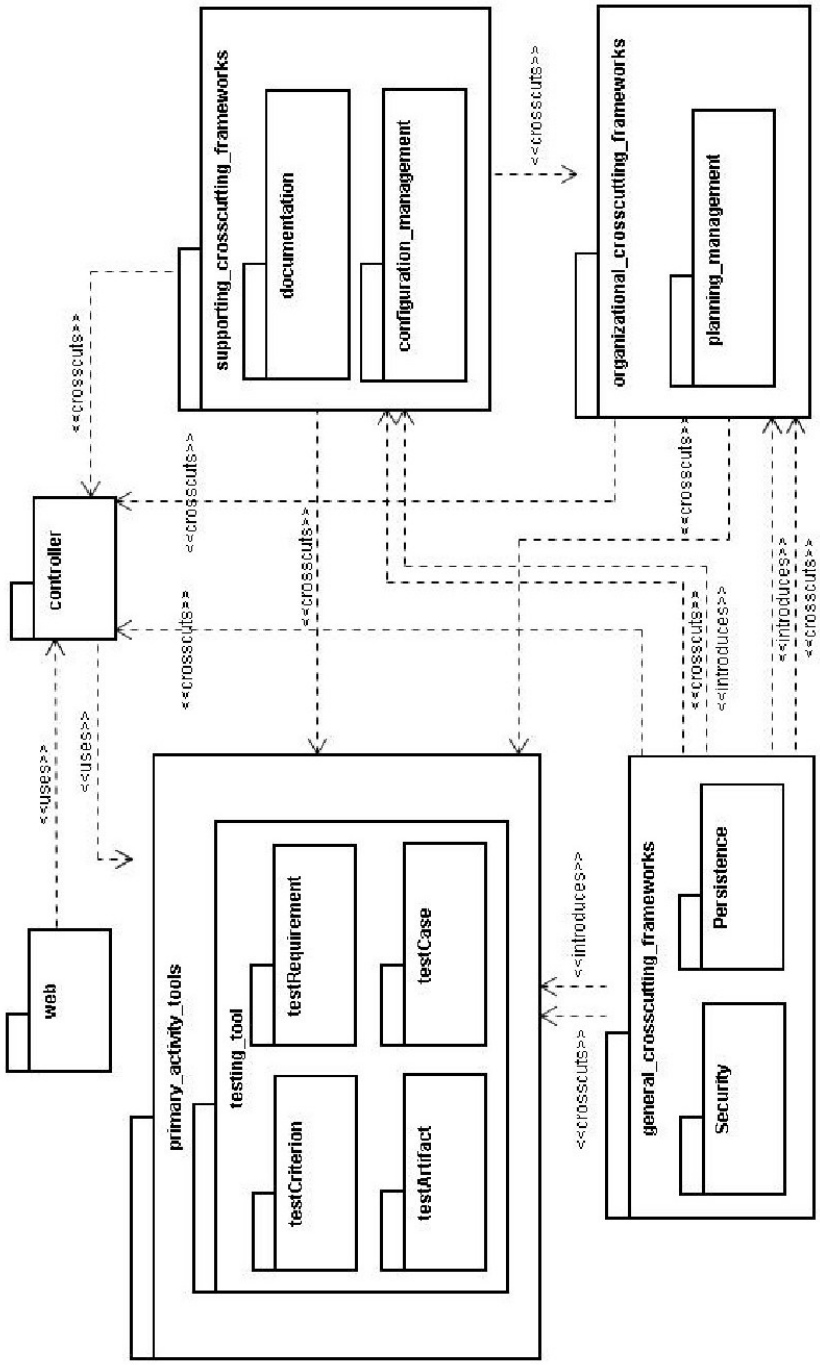

Figura 4.7: Visão de Módulos da RefTEST (Nakagawa et al., 2007)

Na Figura 4.8 é apresentada a visão em tempo de execução da RefTEST, que ilustra as interações entre os módulos de persistência (*Persistence*), o módulo de segurança (*Security*), o módulo de documentação (*Documentation*), de gerência de configuração (*Configuration_management*) e do módulo base (*Testing*). Nesse diagrama é utilizado o conceito de interface transversal, na qual um conjunto de características transversais definidas dentro de aspectos interagem com outros componentes por meio de entrecortes. Como a UML não apoia a representação de interfaces transversais, nesse diagrama, é

utilizada uma notação estendida, na forma de uma "interface preenchida", como as apresentadas pelo componente *Documentation*. Essa notação indica que tal interface interage com outros componentes por meio de entrecortes.

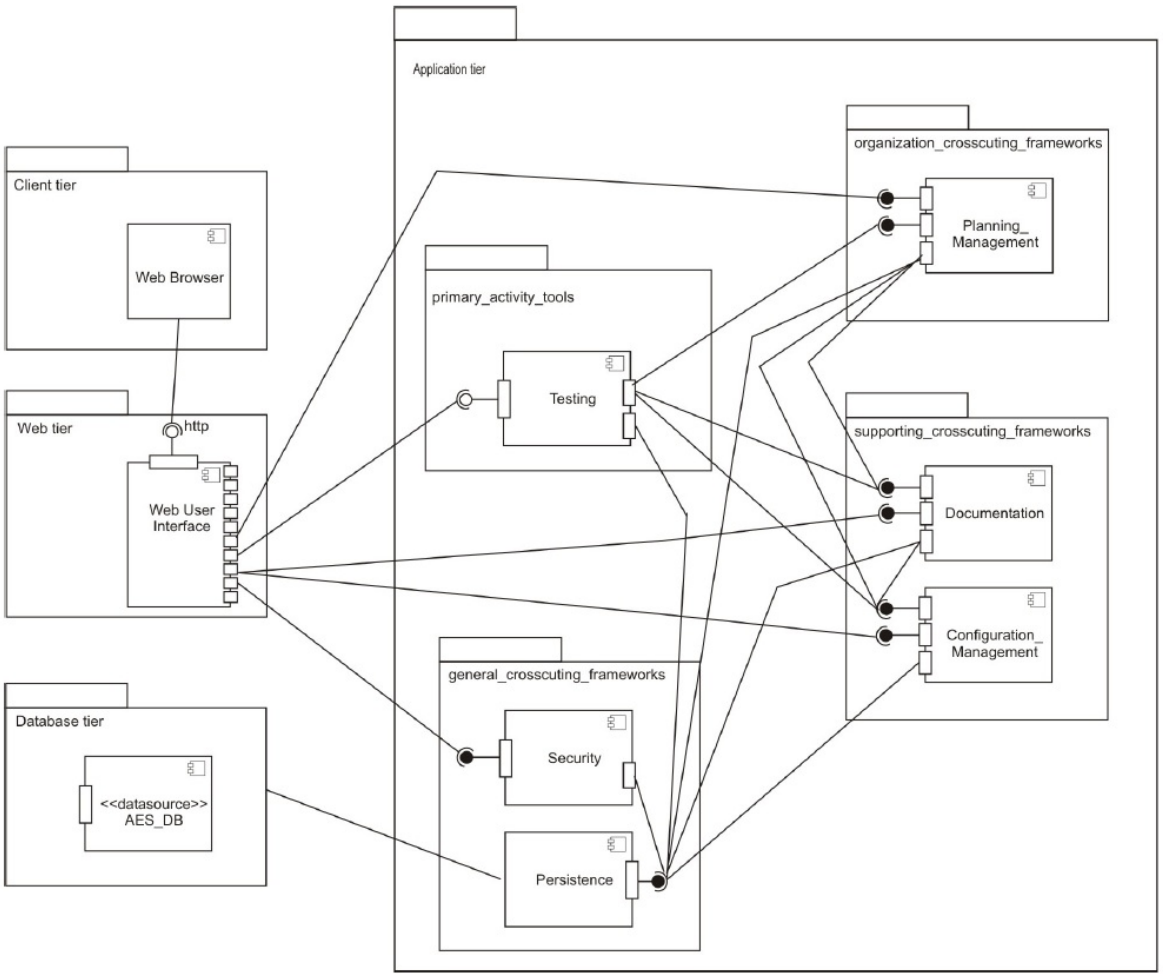

Figura 4.8: Visão em Tempo de Execução da RefTEST (Nakagawa et al., 2007)

Uma vez que a RefTEST dá base para o desenvolvimento de ferramentas de teste disponibilizadas tanto por meio da plataforma web quanto como uma ferramenta *stand--alone*, na Figura 4.9 é ilustrada uma possível visão de implantação para ferramentas desenvolvidas a partir dessa arquitetura. Nessa abordagem, clientes (*Internet User* e *Intranet Admin*) fazem uso de um navegador para ter acesso e utilizar funcionalidades das ferramentas de teste disponibilizadas por um servidor web (*web server*). Essas funcionalidades, por sua vez, são executadas pelo servidor de aplicação (*Application server*), que é responsável pelo armazenamento das ferramentas de teste. Por fim, no servidor de base de dados (*Database server*) é realizada a persistência dos dados referentes à atividade de teste.

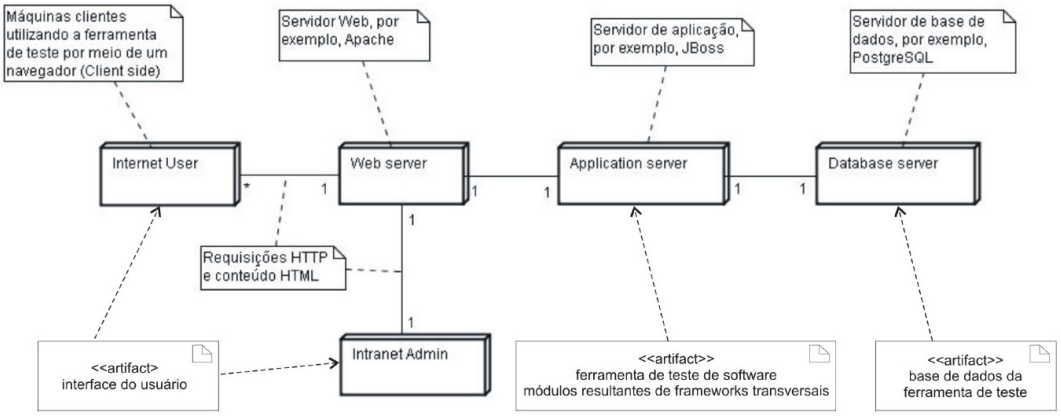

Figura 4.9: Visão de Implantação da RefTEST (Nakagawa, 2006)

Uma vez que a RefTEST será utilizada como base para o desenvolvimento da arquitetura que está sendo proposta neste trabalho, as funcionalidades específicas das ferramentas de teste de software, definidas nesse grupo de fontes de informação (Grupo 4), serão reutilizadas. Contudo, os requisitos referentes ao estilo arquitetural SOA necessitaram ser elicitados. O estabelecimento da arquitetura de referência RefTEST-SOA utiliza as informações de cada uma das fontes (descritas nos Grupos 1 a 4) para a definição das informações específicas, visando identificar os conceitos e funcionalidades pertencentes às ferramentas de teste de software orientadas a serviço. Na Figura 4.10 é ilustrada a abordagem utilizada para o estabelecimento dos requisitos arquiteturais da RefTEST-SOA.

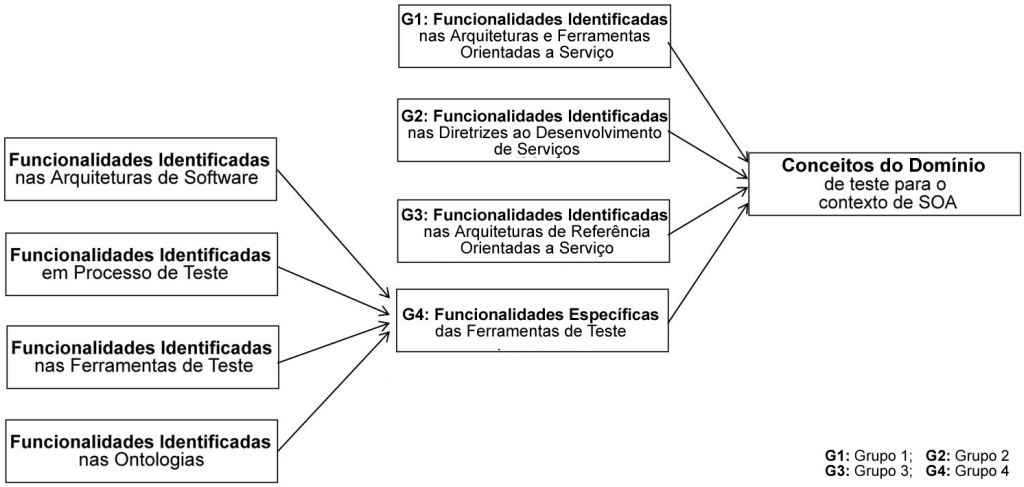

Figura 4.10: Mapeamento das funcionalidades identificadas nos grupos de fontes de informação

4.3 Passo RA-2: Estabelecimento dos Requisitos Arquiteturais

Durante o passo RA-2 do ProSA-RA, chamado "Estabelecimento de Requisitos Arquiteturais", as informações e conceitos do domínio obtidos como resultado da investigação das fontes de informação foram utilizados na elicitação dos requisitos da arquitetura de referência sendo proposta. Uma vez que a RefTEST-SOA é destinada ao contexto de serviço, as atividades relacionadas ao uso e identificação de aspectos arquiteturais[3], definidas pelo processo adotado, não foram aplicadas. Os requisitos arquiteturais da RefTEST-SOA foram divididos em dois conjuntos: requisitos arquiteturais do domínio de teste e requisitos arquiteturais relacionados à SOA.

4.3.1 Requisitos Arquiteturais do Domínio de Teste de Software

Os requisitos arquiteturais do domínio de teste apresentados nesta seção foram obtidos por meio da análise das fontes de informação pertencentes ao Grupo 4, discutido na seção anterior. A seguir, são listados os 19 Requisitos Arquiteturais relacionados ao domínio de Teste de software (RA-T) que foram elicitados para a RefTEST-SOA.

RA-T [1]: A arquitetura de referência deve viabilizar o desenvolvimento de ferramentas de teste que automatizem o tratamento, manipulação e visualização de artefatos a serem testados;

RA-T [2]: A arquitetura de referência deve viabilizar o desenvolvimento de ferramentas de teste que automatizem a execução dos artefatos a serem testados utilizando os casos de teste;

RA-T [3]: A arquitetura de referência deve viabilizar o desenvolvimento de ferramentas de teste que forneçam relatórios sobre os testes executados;

RA-T [4]: A arquitetura de referência deve viabilizar o desenvolvimento de ferramentas de teste que possibilitem a inclusão, utilização e visualização de *drivers* e *stubs*;

RA-T [5]: A arquitetura de referência deve apoiar o desenvolvimento de ferramentas de teste que viabilizem a exclusão de artefatos de teste;

[3]Segundo Nakagawa et al. (2009a), um aspecto arquitetural refere-se a um elemento que entrecorta outro elemento arquitetural como, por exemplo, pacotes e componentes, ou ainda outros aspectos arquiteturais.

RA-T [6]: A arquitetura de referência deve apoiar o desenvolvimento de ferramentas de teste que viabilizem a importação e exportação de casos de teste;

RA-T [7]: A arquitetura de referência deve apoiar o desenvolvimento de ferramentas de teste que viabilizem a geração automática de casos de teste;

RA-T [8]: A arquitetura de referência deve apoiar o desenvolvimento de ferramentas de teste que viabilizem a minimização de conjuntos de casos de teste;

RA-T [9]: A arquitetura de referência deve viabilizar o desenvolvimento de ferramentas de teste que automatizem a manipulação de casos de teste, de modo que esses possam ser habilitados e desabilitados;

RA-T [10]: A arquitetura de referência deve apoiar o desenvolvimento de ferramentas de teste que permitam a visualização e criação de relatórios de casos de teste;

RA-T [11]: A arquitetura de referência deve apoiar o desenvolvimento de ferramentas de teste que possibilitem a exclusão de casos de teste;

RA-T [12]: A arquitetura de referência deve viabilizar o desenvolvimento de ferramentas de teste que automatizem a execução de requisitos de teste utilizando os casos de teste e também armazenem o *trace* referente à sua execução;

RA-T [13]: A arquitetura de referência deve viabilizar o desenvolvimento de ferramentas de teste que automatizem a manipulação de requisitos de teste de modo que esses possam ser selecionados, desmarcados ou indicados como não-executáveis;

RA-T [14]: A arquitetura de referência deve apoiar o desenvolvimento de ferramentas de teste que viabilizem a adição de requisitos de teste, seja por meio de geração automática ou por importação;

RA-T [15]: A arquitetura de referência deve viabilizar o desenvolvimento de ferramentas de teste que permitam a visualização e criação de relatórios de requisitos de teste;

RA-T [16]: A arquitetura de referência deve apoiar o desenvolvimento de ferramentas de teste que possibilitem a exclusão de requisitos de teste;

RA-T [17]: A arquitetura de referência deve apoiar o desenvolvimento de ferramentas de teste que automatizem o estabelecimento de critérios de adequação do teste e o cálculo da análise de cobertura;

RA-T [18]: A arquitetura de referência deve apoiar o desenvolvimento de ferramentas de teste que viabilizem a análise estática da execução do teste pelos testadores; e

RA-T [19]: A arquitetura de referência deve apoiar o desenvolvimento de ferramentas de teste que viabilizem a geração de relatórios sobre teste executado.

Esses requisitos estão diretamente relacionados às funcionalidades e aos conceitos apresentados na Tabela 4.6, e representam as funcionalidades que as ferramentas de teste baseadas na arquitetura de referência sendo proposta devem possuir. Na Tabela 4.7 é apresentada uma relação entre os conceitos do domínio e os requisitos arquiteturais para o projeto da RefTEST-SOA.

Tabela 4.7: Relação entre os conceitos de teste e requisitos arquiteturais da RefTEST-SOA

Conceito	Requisito associado
Artefato de Teste	RA-T [1], RA-T [2], RA-T [3], RA-T [4], RA-T [5]
Caso de Teste	RA-T [6], RA-T [7], RA-T [8], RA-T [9], RA-T [10], RA-T [11]
Requisito de Teste	RA-T [12], RA-T [13], RA-T [14], RA-T [15], RA-T [16]
Critério de Teste	RA-T [17], RA-T [18], RA-T [19]

4.3.2 Requisitos Arquiteturais do Contexto de Serviços

Além dos requisitos arquiteturais referentes ao domínio de teste de software, foram elicitados requisitos específicos de sistemas orientados a serviço. Tais requisitos foram obtidos com base nas funcionalidades e nos conceitos identificados na análise dos Grupos 1, 2 e 3, apresentados na Seção 4.2. A seguir, são listados os 14 Requisitos Arquiteturais relacionados à SOA (RA-S) que foram elicitados para a RefTEST-SOA.

RA-S [1]: A arquitetura de referência deve possibilitar que ferramentas de teste desenvolvidas para automatizar diferentes técnicas e critérios de teste possam ser facilmente integradas;

RA-S [2]: A arquitetura de referência deve possibilitar que ferramentas de teste implementadas em linguagens de programação distintas e sob diferentes plataformas possam ser facilmente integradas;

RA-S [3]: A arquitetura de referência deve prover mecanismos para que ferramentas de teste na forma de serviços possam ser publicadas e posteriormente descobertas por aplicações cliente;

RA-S [4]: A arquitetura de referência deve prover mecanismos para que ferramentas de teste orientadas a serviço possam ser compostas por processos de negócio ou utilizadas por aplicações cliente;

RA-S [5]: A arquitetura de referência deve viabilizar o desenvolvimento de ferramentas de teste que disponibilizem informações sobre suas características e direções normativas de uso, por meio de descrições padronizadas;

RA-S [6]: A arquitetura de referência deve viabilizar o desenvolvimento de ferramentas de teste que disponibilizem descrições semânticas, permitindo assim sua classificação nos repositórios de serviço;

RA-S [7]: A arquitetura de referência deve viabilizar o desenvolvimento de ferramentas de teste que tenham à disposição informações e documentos relacionados às suas características de qualidade;

RA-S [8]: A arquitetura de referência deve prover mecanismos para a captura, monitoramento, registro e sinalização do não cumprimento de requisitos de qualidade estabelecidos entre serviços provedores e serviços clientes;

RA-S [9]: A arquitetura de referência deve viabilizar o desenvolvimento de ferramentas de teste escaláveis, capazes de evoluir de maneira incremental, por meio da composição de novas funcionalidades disponíveis na forma de serviços;

RA-S [10]: A arquitetura de referência deve possibilitar que serviços de teste e composições desses serviços sejam tratados uniformemente, ou seja, possam ser publicados, localizados e utilizados da mesma forma;

RA-S [11]: A arquitetura de referência deve possibilitar que serviços de teste possam interagir diretamente ou por meio do uso de barramentos de serviço;

RA-S [12]: A arquitetura de referência deve possibilitar que ferramentas de teste de software disponíveis como serviços, desenvolvidas sem o uso dessa arquitetura, possam ser integradas às ferramentas desenvolvidas segundo a arquitetura de referência;

RA-S [13]: A arquitetura de referência deve possibilitar sua instanciação parcial, ou seja, ferramentas desenvolvidas a partir dessa arquitetura podem ser construídas sem a necessidade de implementação de todos os módulos propostos; e

RA-S [14]: A arquitetura de referência deve permitir que ferramentas de teste possam evoluir ao longo de diferentes versões de maneira simples, ou seja, mudanças de versão das ferramentas de teste não devem afetar a interação com os serviços que a utilizam.

Para que esses requisitos fossem elicitados, foram consideradas as informações relacionadas aos conceitos de Descrição de Serviço (DS), Publicação de Serviço (PS), Interação

entre Serviços (IS), Composição de Serviços (CS) e Qualidade de Serviço (QoS), discutidos na Tabela 4.3. Tais conceitos foram considerados durante a elicitação dos requisitos por serem os mais utilizados entre as arquiteturas de referência orientadas a serviços destinadas a domínios específicos. Além disso, requisitos mais gerais, elicitados a partir da análise das arquiteturas e ferramentas orientadas a serviço de teste, verificação e análise de software (abordadas na Seção 4.2.1) também foram considerados. Na Tabela 4.8 é apresentada a relação entre os conceitos referentes ao contexto da SOA e os requisitos elicitados para a arquitetura de referência sendo proposta.

Tabela 4.8: Relação entre os conceitos e requisitos arquiteturais referentes à SOA

Conceito	Requisito associado
Descrição de Serviço (DS)	RA-S [5], RA-S [6], RA-S [7], RA-S [10]
Publicação de Serviço (PS)	RA-S [3], RA-S [6], RA-S [10]
Interação entre Serviços (IS)	RA-S [4], RA-S [10], RA-S [11], RA-S [14]
Composição de Serviços (CS)	RA-S [4], RA-S [9], RA-S [10]
Qualidade de Serviço (QoS)	RA-S [7], RA-S [8]
Requisito Geral	RA-S [1], RA-S [2], RA-S [12], RA-S [13]

4.4 Passo RA-3: Projeto Arquitetural

Com base nos requisitos arquiteturais elicitados na seção anterior, no Passo RA-3 do ProSA-RA, descrito nesta seção, é realizado o projeto da arquitetura de referência. Como forma de melhor documentar a arquitetura sendo proposta, são apresentadas a visão geral e as três visões arquiteturais da RefTEST-SOA, conforme proposto pelo processo adotado. As visões desenvolvidas para a RefTEST-SOA são: visão geral (Seção 4.4.1), visão de módulo (Seção 4.4.2), visão em tempo de execução (Seção 4.4.3) e visão de implantação (Seção 4.4.4). É importante destacar que os conceitos representados nas visões arquiteturais são independentes da abordagem de desenvolvimento, podendo ser implementados em diferentes linguagens de programação e tecnologias.

4.4.1 Visão Geral

Na Figura 4.11 é apresentada a visão geral da RefTEST-SOA. Na arquitetura sendo proposta, cada um dos elementos do módulo base da camada de aplicação implementa um conjunto de funcionalidades referente a um determinado conceito do domínio de teste, discutido previamente na Tabela 4.6. Tais elementos, no nível de projeto e implementação, devem ser desenvolvidos como serviços distintos e independentes entre si. É importante salientar que, cada um desses serviços pode ser desenvolvido pela linguagem de programação que lhe for mais adequada, utilizando as tecnologias e subsistemas que forem mais

convenientes a sua implementação. Além disso, como forma de evoluir a RefTEST para o contexto da SOA, foram introduzidos conceitos específicos de serviços, identificados durante os passos RA-1 e RA-2 do ProSA-RA. A estrutura geral da RefTEST-SOA, bem como as características de suas seis camadas são apresentadas a seguir:

Camada de aplicação: contempla os conceitos diretamente relacionados ao domínio de teste de software. A camada de aplicação é composta por quatro conjuntos de serviços, denominados serviços de teste: (i) `Serviços de Teste Primários` que se referem aos quatro principais serviços de uma ferramenta de teste (ou seja, serviços que oferecem apoio ao gerenciamento de casos de teste, critérios de teste, artefatos de teste e requisitos de teste). Tais serviços formam a base da arquitetura de referência sendo proposta; (ii) `Serviços Ortogonais de Suporte` que apoiam as atividades de engenharia de software consideradas como atividades de suporte pela ISO/IEC 12207 e que podem ser aplicadas ao domínio de teste; (iii) `Serviços Ortogonais Organizacionais` que apoiam as atividades de engenharia de software consideradas como atividades organizacionais pela ISO/IEC 12207 e que também podem ser aplicadas ao domínio de teste; e (iv) `Serviços Ortogonais Gerais` que se referem aos serviços de propósito geral, tais como a `Persistência` e a `Segurança`. É importante salientar que esses três últimos grupos de serviços[4] são considerados ortogonais, uma vez que cada um deles pode ser utilizado por diversos outros serviços durante a atividade de teste. Por exemplo, o serviço de `Documentação`, responsável por documentar a atividade de teste, pode ser utilizado por outros serviços, tais como o `Caso de Teste` e o `Critério de Teste`. Além disso, na RefTEST-SOA, os serviços implementados na camada de aplicação devem ser desenvolvidos visando à modularidade e à coesão, para que possam ser utilizados por diferentes ferramentas de teste, de forma a aprimorar a capacidade de reúso. É importante observar que a RefTEST-SOA tem o objetivo de ser uma arquitetura de referência dirigida à pesquisa[5] (do inglês, *research-driven architecture*), visando tornar possível o desenvolvimento de ferramentas de teste que forneçam esses novos serviços;

Camada de persistência: é responsável por armazenar os dados produzidos pelos serviços que compõem as ferramentas de teste. Em particular, o serviço `Persistência` atua diretamente nessa camada. Para que os dados possam ser persistidos na base

[4]Os conjuntos de serviços em (ii), (iii) e (iv) são baseados e possuem as mesmas características dos módulos transversais propostos pela RefTEST.

[5]Segundo Angelov et al. (2008), uma arquitetura de referência dirigida à pesquisa provê uma visão futurística de ferramentas que deverão tornar-se importantes no futuro, mas que ainda não possuem todos os seus detalhes de projeto claramente definidos.

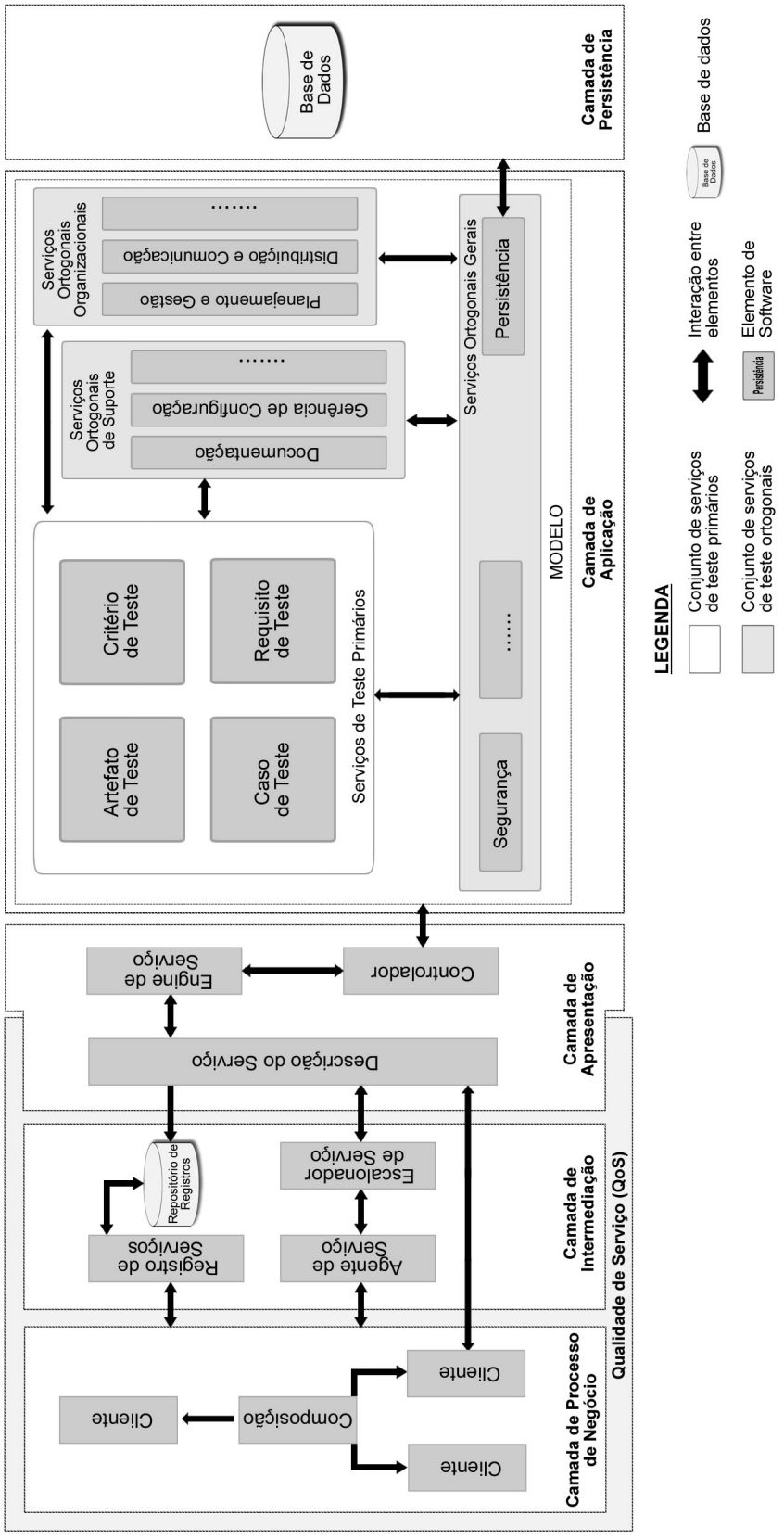

Figura 4.11: Visão geral da RefTEST-SOA

de dados, Sistemas de Gerenciamento de Bancos de Dados (SGBD) podem ser utilizados. Além disso, *frameworks* como o Hibernate[6], podem ser utilizados com o intuito de facilitar a persistência e a recuperação de dados da base de dados;

Camada de apresentação: em aplicações *stand-alone* e web, essa camada é destinada ao processamento de eventos e à apresentação visual das informações sobre a atividade de teste. Contudo, na RefTEST-SOA, destinada ao contexto de sistemas orientados a serviço, essa camada é parte de cada serviço e possui três principais elementos: (i) `Descrição do Serviço`, que define os tipos de dados utilizados na requisição de funcionalidades e também os tipos de dados retornados aos clientes após o processamento; (ii) ***`Engine`*** `de Serviço`, que processa requisições de serviços, convertendo mensagens escritas em linguagens e protocolos padrão em tipos de dados utilizados pelas implementações localizadas na `Camada de Aplicação`. A ***`Engine`*** `de Serviço` deve ser capaz de transformar tipos de dados de serviços que desempenham tanto o papel de cliente quanto o de provedor; e (iii) um `Controlador` responsável pelo tratamento de requisições encaminhadas pela ***`Engine`*** `de Serviço`;

Camada de intermediação: nessa camada, serviços de teste podem ser publicados, descobertos, associados e disponibilizados. Essa camada é composta por três elementos: (i) o `Registro de Serviços`, que representa a entidade responsável pelo recebimento de publicações e pesquisas sobre os serviços disponíveis, e que pode ser materializado de diversas formas como, por exemplo, por meio de *brokers* ou *matchmakers* (Dillon et al., 2007). Para o domínio de teste de software, um repositório de serviços já pode ser encontrado (Gondim, 2010). Tal repositório faz uso de ontologias de teste (Bai et al., 2008; Barbosa et al., 2006) para a classificação, descoberta e a inclusão de informações semânticas dos serviços de teste nele catalogados; (ii) o `Agente de Serviço`, que representa o papel de mediador e pode ser utilizado como arcabouço de roteamento e transporte de requisições entre serviços clientes e seus provedores, promovendo uma comunicação indireta entre eles. Tal arcabouço pode ser materializado, por exemplo, pela implementação ou uso de um ESB; e (iii) o `Escalonador de Serviço`, que processa as requisições de serviço feitas por clientes de acordo com as informações de dependências existentes, descritas por uma linguagem padrão como, por exemplo, a XML. Casos nos quais requisições referem-se a serviços simples, envolvendo apenas um serviço provedor, políticas de uso podem ser facilmente verificadas. Contudo, em casos nos quais o serviço solicitado é composto e

[6]O Hibernate (`http://www.hibernate.org/`) é um *framework* para programas escritos em linguagem Java (disponível também para .Net, com o nome NHibernate), que facilita o mapeamento dos atributos entre bases tradicionais de dados relacionais e o modelo de objetos de uma aplicação, mediante o uso de arquivos (XML).

possui dependências, características de interoperabilidade entre as partes agregadas e suas políticas de uso devem ser analisadas pelo escalonador;

Camada de processo de negócio: nessa camada novos serviços podem ser construídos por meio da composição de serviços de teste, organizacionais, de suporte e gerais já existentes, organizados de acordo com processos de negócio definidos. Para a organização entre os serviços compostos, abordagens como orquestração ou coreografia podem ser utilizadas. Para que processos possam ser orquestrados, linguagens padronizadas como a WS-BPEL podem ser utilizadas, caso a abordagem de desenvolvimento adotada seja a de serviços web. No caso da abordagem de interação de serviços ser por meio de coreografia pode ser utilizada, por exemplo, a WS-CDL. Por meio da camada de processo de negócio, clientes podem pesquisar funcionalidades no `Registro de Serviços`, solicitar funcionalidades de serviços clientes de forma direta ou interagir por meio de um `Agente de Serviço`; e

Camada de qualidade de serviço: é a camada utilizada para inspecionar o cumprimento de requisitos de qualidade presentes nas demais camadas de serviços. São permeadas pela `Qualidade de Serviço` as camadas relacionadas aos processos de negócio (`Camada de Processo de Negócio`), à mediação (`Camada de Intermediação`) e parte da camada de apresentação, no que tange à descrição das características de uso do serviço, definidas pela interface (`Descrição do Serviço`). Na camada de `Qualidade de Serviço` são capturadas, monitoradas, registradas e sinalizadas não-conformidades em requisitos relacionados à qualidade de serviço. Em outras palavras, essa camada observa as demais camadas relacionadas ao serviço, sinalizando e produzindo eventos quando um requisito não-funcional é desrespeitado. O tempo máximo gasto com a execução de um requisito de teste com um determinado caso de teste é um exemplo de requisito não-funcional relacionado ao domínio de teste. Nessa camada são fornecidos os meios para garantir que serviços correspondam às suas exigências no que diz respeito à confiabilidade, disponibilidade, gerenciamento, escalabilidade e segurança. A inclusão da camada de qualidade de serviço possui grande importância, uma vez que, em SOA, aplicações podem ser desenvolvidas por diferentes empresas, sendo esse desenvolvimento realizado sob infraestruturas heterogêneas com características próprias. Dessa forma, existe a necessidade de assegurar que as interações entre serviços de teste, organizacionais, de suporte e gerais sejam realizadas de maneira satisfatória.

É importante ressaltar que a implementação dos serviços ortogonais de suporte, organizacionais e gerais parece uma iniciativa viável, e pode aprimorar a qualidade da atividade de teste. Serviços ortogonais que implementem as atividades de apoio e organizacionais do

padrão ISO/IEC 12207 (ISO, 1995), tais como o gerenciamento, auditoria, documentação e treinamento, poderiam ser utilizados por diferentes serviços de teste, que automatizassem técnicas e critérios de teste distintos. Junto com os serviços gerais, como o relacionado à segurança da atividade de teste, os serviços de suporte e organizacionais auxiliariam no desenvolvimento de um ambiente de teste integrado, mais completo, acessível e escalável, destinado à condução da atividade de teste.

4.4.2 Visão de Módulo

Na visão de módulo é representada a estrutura de uma arquitetura por meio de pacotes, classes e interfaces, que descrevem as unidades de código que implementam funcionalidades representadas por suas partes componentes. Segundo Nakagawa (2006), essa visão arquitetural possui grande importância, uma vez que representa a "planta" para a construção do software. Na Figura 4.12 é ilustrada a visão de módulo da RefTEST-SOA, descrita em UML.

Na RefTEST-SOA, quatro conjuntos de serviços relacionados à atividade de teste de software são propostos: serviços de teste primários (pacote `primaryTestingServices`), serviços ortogonais de suporte (pacote `orthogonalSupportingServices`), serviços ortogonais organizacionais (pacote `orthogonalOrganizationalServices`) e serviços ortogonais gerais (pacote `orthogonalGeneralServices`). O pacote de serviços de teste primários representa o núcleo dos serviços que implementam as atividades fundamentais realizadas por uma ferramenta de teste de software, tais como o gerenciamento de casos de teste e o cálculo da cobertura do teste. Cada um desses serviços de teste deve ser implementado de maneira independente, visando facilitar a composição de ferramentas de teste mais completas, por meio de processos de negócio. Serviços presentes nos pacotes que contém os serviços ortogonais, como o destinado à documentação da atividade de teste (`Documentation`), podem ser utilizados para apoiar a execução dos serviços de teste primários, sendo invocados durante toda a atividade de teste. Além disso, outros serviços — `serviceSchedule`, `serviceAgent`, `serviceRegistry` e `qualityOfService` — tornam possíveis as interações entre os serviços de teste, de suporte, organizacionais e gerais. Nessa visão, o pacote *client* é ilustrado com o objetivo de representar como os clientes podem fazer uso dos serviços de teste disponíveis. É importante lembrar que, uma vez que a arquitetura de referência sendo proposta é baseada no estilo arquitetural SOA, os pacotes apresentados nessa visão podem estar localizados de maneira fisicamente distribuída, podendo, inclusive, pertencer a diferentes instituições. Dessa forma, as notações de uso podem ser implementadas tanto localmente, quanto por meio de chamadas remotas, para serviços situados em outros servidores.

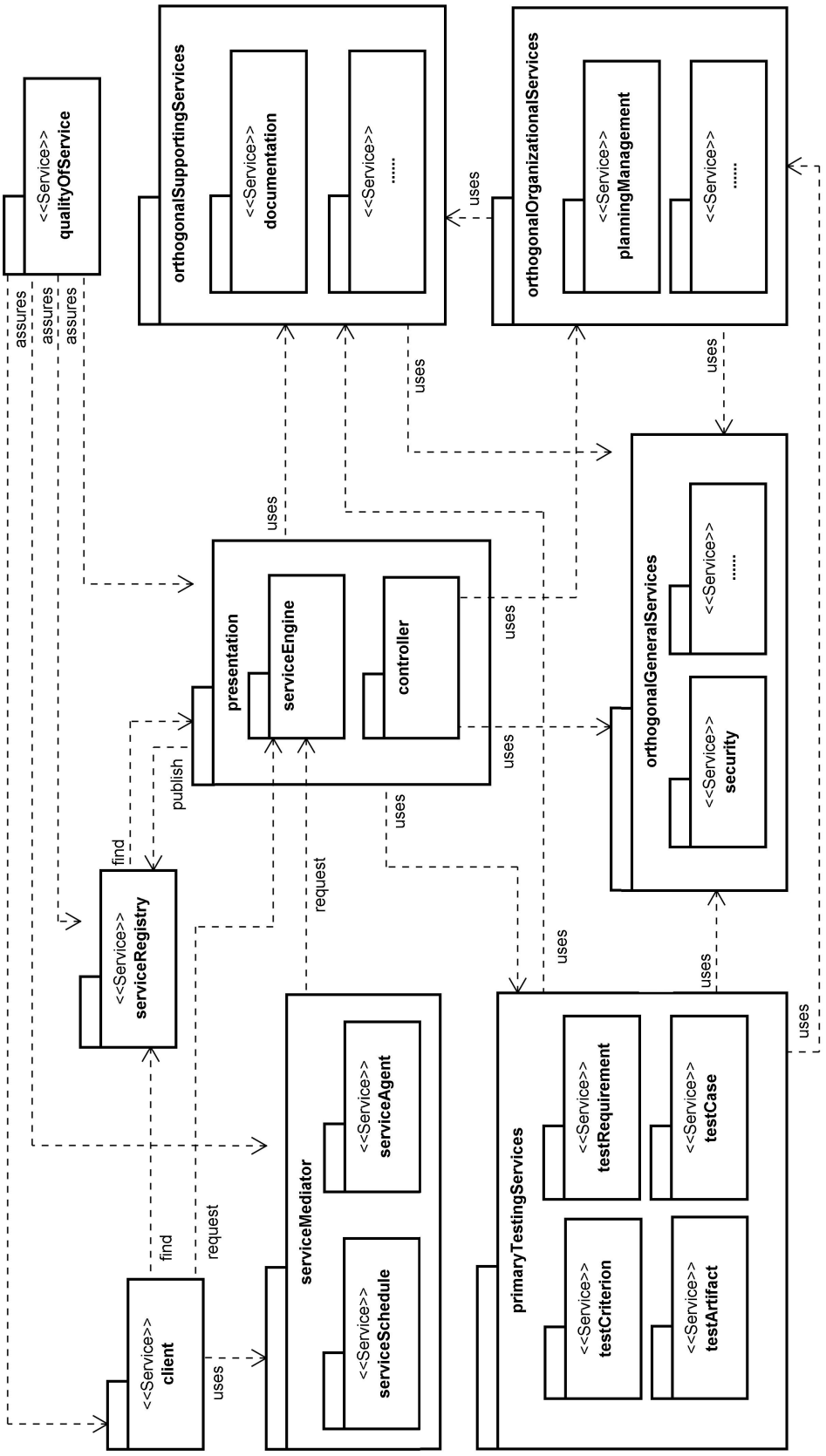

Figura 4.12: Visão de módulo da RefTEST-SOA

4.4.3 Visão em Tempo de Execução

A visão em tempo de execução apresenta a estrutura de um sistema no momento em que ele está sendo executado, por meio da representação de componentes em diferentes níveis de abstração, as interfaces providas e requeridas, o fluxo de dados do sistema, as bases de dados, os elementos replicados e as execuções que ocorrem em paralelo. Na Figura 4.13 é apresentada a visão de tempo de execução da RefTEST-SOA.

Apesar da arquitetura de referência em proposição ser independente de tecnologias de implementação, na visão em tempo de execução apresentada foram adicionadas algumas informações, tais como linguagens e protocolos utilizados por serviços web, visando auxiliar no entendimento da mesma. Contudo, as interações entre os componentes propostos não são dependentes de tal tecnologia, podendo esses serem implementados por tecnologias equivalentes.

No diagrama proposto, os componentes que representam os serviços de teste primários — `TestRequirement`, `TestArtifact`, `TestCriteria` e `TestCase` — são responsáveis por prover as funcionalidades relacionadas aos conceitos do domínio de teste detalhadas na Tabela 4.6. Dessa forma, funcionalidades como o tratamento e a geração de artefatos de teste podem ser providas pelo serviço `TestArtifact`, assim como o estabelecimento do critério de adequação e o cálculo da cobertura do teste podem são disponibilizados pelo serviço `TestCriteria`. Caso o serviço cliente (componente `Client`) desconheça a localização do serviço ao qual irá utilizar, esse deve primeiramente pesquisá-lo em um registro de serviços (componente `Service Registry`), utilizando um protocolo padrão (por exemplo, o protocolo UDDI). Caso algum serviço seja encontrado, o registro informará ao consumidor o endereço do serviço de teste (*endpoint*), de forma que eles possam se comunicar. Após estabelecer a conexão, o serviço cliente solicita informações referentes à descrição do serviço que irá utilizar, para que eles possam interagir corretamente. A descrição do serviço pode ser feita por meio de uma linguagem como a WSDL. Uma vez conhecido o endereço do serviço provedor e a forma como ele é descrito, a comunicação pode ser realizada diretamente ou por meio de serviços intermediários (componente `Service Agent`). Em ambos os casos, a comunicação entre o cliente e o provedor deve ser sempre baseada em um protocolo que seja construído sob uma linguagem padronizada como, por exemplo, o SOAP. Visando assegurar a qualidade da interação entre os serviços, o componente `Quality of Service` monitora a comunicação entre os serviços clientes e os serviços provedores, visando o cumprimento dos requisitos não-funcionais previamente estabelecidos como, por exemplo, o tempo máximo de resposta.

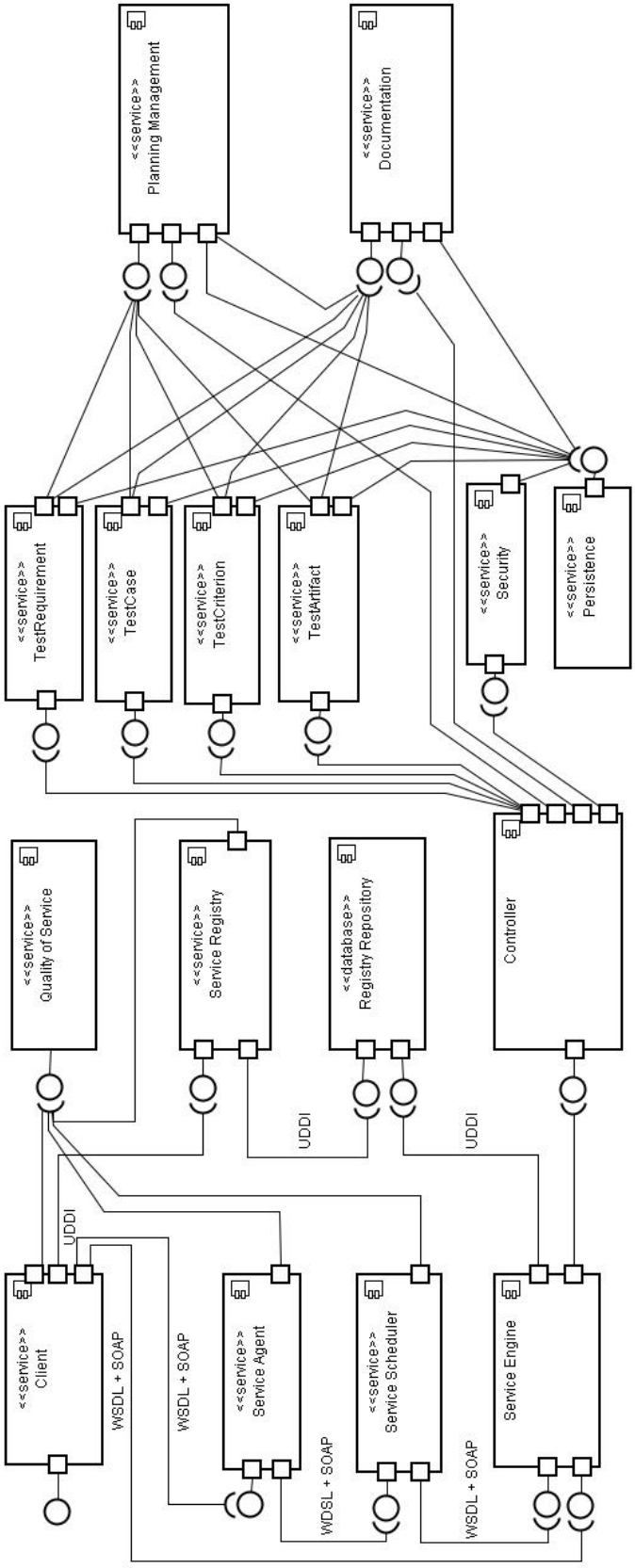

Figura 4.13: Visão em tempo de execução da RefTEST-SOA

4.4.4 Visão de Implantação

Face ao fato de serviços conterem funcionalidades que interagem de forma fisicamente distribuída, a representação das estruturas que compõem a realização de sistemas orientados a serviço com a visão de implantação é particularmente útil. Por meio dessa visão, é apresentada a estrutura de hardware sobre a qual os sistemas são alocados e as conexões de rede que simbolizam as interações entre esses dispositivos. Dessa forma, com a visão de implantação, é possível representar a estrutura física na qual os serviços de teste, seus clientes e toda a infraestrutura disponível para as interações realizadas segundo a SOA são alocados. Para a construção da visão de implantação foi utilizado o diagrama de implantação da UML. A Figura 4.14 ilustra uma possível visão de implantação da RefTEST-SOA.

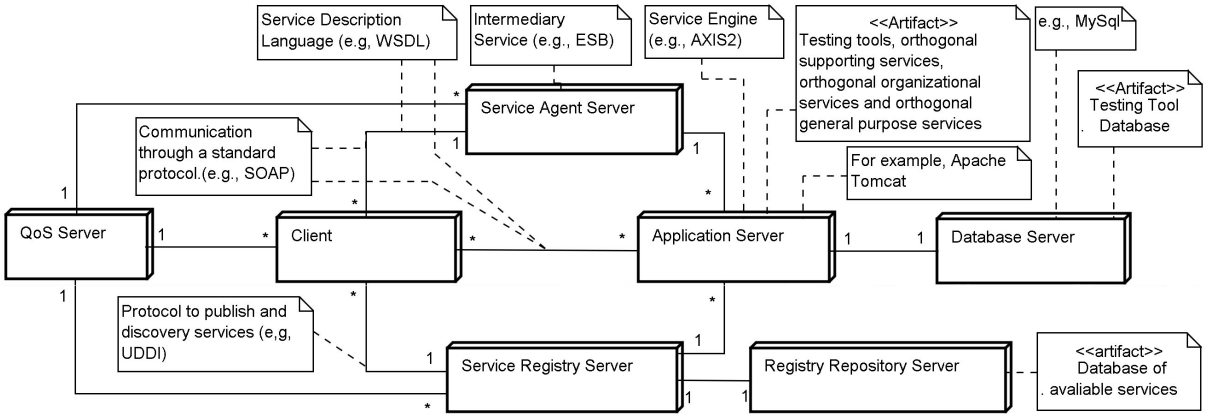

Figura 4.14: Visão de implantação RefTEST-SOA

De acordo com essa visão arquitetural, os seguintes elementos são considerados: (i) servidor de aplicação (`Application Server`), que contém os serviços de teste, de suporte, organizacionais e gerais e é responsável por executar as requisições realizadas por serviços clientes. Nesse servidor, além desses serviços, pode ser alocada também uma ***`Engine`*** `de Serviços`, como por exemplo a AXIS2 (Apache Foundation, 2010); (ii) servidor de base de dados (`Database Server`), responsável por armazenar os dados produzidos pelos serviços executados no servidor de aplicação; (iii) servidor de registro de serviços (`Service Registry Server`), que gerencia informações acerca dos serviços de teste disponíveis e também apoia o processamento de pesquisas por parte dos serviços clientes; (iv) um repositório para o armazenamento dos registros de serviço (`Registry Repository Server`) gerenciados pelo servidor de registro de serviços (`Service Registry Server`); (v) o servidor de agente de serviço (`Service Agent Server`), no qual são armazenadas aplicações relacionadas ao funcionamento do agente de serviços (componente `Service Agent`), discutido previamente na visão em tempo de execução e que pode ser materializado na forma de um ESB; e (vi) o servidor de qualidade de serviço (`QoS Server`), que é destinado à hospedagem de

aplicações relacionadas à garantia da qualidade e à auditoria dos serviços disponíveis, bem como suas interações.

4.5 Passo RA-4: Avaliação da Arquitetura de Referência

No passo RA-4 do ProSA-RA, denominado "Avaliação da Arquitetura de Referência", avaliações com o objetivo de evidenciar a qualidade da arquitetura de referência proposta devem ser realizadas. Para isso, inspeções por meio de *checklists* e a aplicação de métodos de avaliação arquitetural, como os propostos por Kazman et al. (1998) e SEI (2010), podem ser utilizados. No contexto desse trabalho, além das avaliações informais realizadas a partir de entrevistas com desenvolvedores de ferramentas de teste orientadas a serviço, visando identificar defeitos relacionados à omissão, ambiguidade, inconsistência e de informações incorretas, a avaliação da arquitetura de referência proposta foi realizada por meio de um estudo de caso. No Capítulo 5 são apresentados os detalhes de implementação de uma ferramenta de teste orientada a serviço desenvolvida como o propósito de prover evidências da aplicabilidade da arquitetura de referência proposta, bem como observar os benefícios relacionados ao reúso e a facilidade de integração.

4.6 Considerações Finais

Frente à importância da automatização da atividade de teste, bem como os benefícios da utilização da SOA no desenvolvimento de sistemas de software, neste capítulo foi apresentada a RefTEST-SOA, uma arquitetura de referência que visa prover direções ao desenvolvimento de ferramentas de teste de software orientadas a serviços. Por meio da RefTEST-SOA, objetiva-se contribuir para a construção de ferramentas de teste que possam ser mais facilmente integráveis, aprimorando também a capacidade de reúso e a qualidade da atividade de teste executada. Além disso, vale a pena ressaltar que as fontes de informação identificadas durante o passo RA-1 do ProSA-RA podem ser utilizadas por projetos de outras arquiteturas do domínio. Os requisitos relacionados ao estilo arquitetural SOA, definidos no passo RA-2, constituem também um ponto de partida para a construção de outras arquiteturas de referência orientadas a serviço.

O estabelecimento de uma arquitetura de referência não é uma tarefa trivial, sobretudo em casos nos quais se objetiva prover direções ao desenvolvimento de sistemas para um domínio de aplicação recente, que ainda não dispõe de um número significativo de arquiteturas concretas e sistemas implementados. É importante destacar, então, que é de grande importância a adoção de um processo que guie o estabelecimento da arquitetura de referência desde a identificação das fontes de informação até a atividade de avaliação

arquitetural. Nesse contexto, a utilização do ProSA-RA foi de grande importância para o estabelecimento da arquitetura proposta. Além disso, a utilização da RefTEST como uma arquitetura referência que fornece apoio ao desenvolvimento de ferramentas de teste de software, provendo um conjunto estruturado de conceitos do domínio, foi de grande valia para a redução da complexidade no estabelecimento da RefTEST-SOA.

É importante lembrar que a RefTEST-SOA possui características de uma arquitetura de referência dirigida à pesquisa, na qual os detalhes de desenvolvimento de alguns módulos ainda necessitam ser futuramente investigados. Em virtude da necessidade do se evidenciar a aplicabilidade da arquitetura proposta, no próximo capítulo, é apresentado um estudo de caso no qual serviços de teste foram implementados segundo as direções oferecidas pela RefTEST-SOA, construindo-se assim uma ferramenta de teste de software orientada a serviço.

Capítulo

5

Estudo de Caso: Utilização da RefTEST-SOA

5.1 Considerações Iniciais

Neste capítulo é apresentado um estudo de caso de utilização da RefTEST-SOA na construção de uma ferramenta de teste de software orientada a serviço. Essa ferramenta, denominada PascalMutants-Service, foi desenvolvida a partir da refatoração e do aprimoramento da ferramenta PascalMutants (Oliveira et al., 2009), que automatiza o critério Análise de Mutantes para o teste de programas desenvolvidos em linguagem Pascal. Apesar do estudo de caso apresentado neste capítulo abordar o desenvolvimento de uma ferramenta simples de teste, acredita-se que seja suficiente para evidenciar a viabilidade de utilização da RefTEST-SOA.

Para o desenvolvimento da PascalMutants-Service foram construídos os quatro serviços de teste primários, definidos na camada de aplicação da RefTEST-SOA (Critério de Teste, Artefato de Teste, Requisito de Teste e Caso de Teste), sendo esses mapeados para o critério Análise Mutantes. Tais serviços foram implementados de forma autocontida, seguindo os conceitos da SOA e visando o aumento da capacidade de reúso. Em seguida, foi criado um processo de negócio responsável por coordenar a interação entre os serviços desenvolvidos. A composição dos quatro serviços de teste, organizados conforme o processo de negócio proposto, constituem a ferramenta de teste PascalMutants-Service.

Na Seção 5.2 é ilustrado o processo de instanciação arquitetural, partindo da RefTEST--SOA até o estabelecimento das arquiteturas de software dos serviços de teste primários. Na Seção 5.3 são apresentados os detalhes de projeto dos quatro serviços que formam a base da PascalMutants-Service. As informações relacionadas à integração dos serviços que deram origem a ferramenta PascalMutants-Service são discutidas na Seção 5.4. Na Seção 5.5 são apresentadas as tecnologias utilizadas e os detalhes relacionados à implementação. Na Seção 5.6 é ilustrado um exemplo de uso da ferramenta de teste desenvolvida. Uma análise preliminar sobre o aumento da capacidade de reúso e da integração é apresentada na Seção 5.7. Por fim, na Seção 5.8 são discutidas as considerações finais deste capítulo.

5.2 Instanciação da RefTEST-SOA para o Critério Análise de Mutantes

Para o desenvolvimento da ferramenta de teste de mutação PascalMutants-Service, foi utilizado o conhecimento disponibilizado pela arquitetura de referência RefTEST-SOA. Em particular, foram utilizados os requisitos que descrevem as funcionalidades a serem providas por cada serviço de teste e a estrutura que estabelece a forma na qual esses serviços devem ser organizados. Visando estabelecer a arquitetura de software para a ferramenta a ser desenvolvida, foi realizada a instanciação arquitetural da RefTEST-SOA para o critério Análise de Mutantes, utilizando os protocolos e linguagens disponíveis para a implementação de serviços web. Na Figura 5.1 é ilustrado o projeto das arquiteturas de software dos serviços que compõem a ferramenta de teste PascalMutants-Service, partindo da arquitetura de referência até o desenvolvimento das arquiteturas concretas.

Para que o projeto das arquiteturas concretas fosse realizado, os conceitos descritos no módulo de `Serviços de Teste Primários` (presente na `Camada de Aplicação`) foram mapeados para o critério Análise de Mutantes. Dessa forma, as seguintes especializações de conceitos foram realizadas: o conceito `Artefato de Teste` foi especializado para o `Artefato de Teste para Mutação`; o conceito de `Critério de Teste` foi especializado para o critério `Análise de Mutantes`; e o `Requisito de Teste` foi especializado como `Mutante`. O conceito de `Caso de Teste` não foi especializado, uma vez que esse é independente de técnica ou critério de teste utilizado. Além das especializações realizadas na `Camada de Aplicação`, os conceitos relacionados ao contexto de serviço foram mapeados segundo a tecnologia de serviços web, sendo a `Descrição do Serviço` feita por meio da linguagem WSDL. Ademais, para que as descrições de serviço pudessem ser construídas de maneira produtiva, o AXIS2 foi utilizado como *`Engine`* `do Serviço`.

É importante destacar que o foco principal desse estudo de caso é evidenciar a viabilidade da implementação dos serviços de teste primários e observar como esses poderiam ser

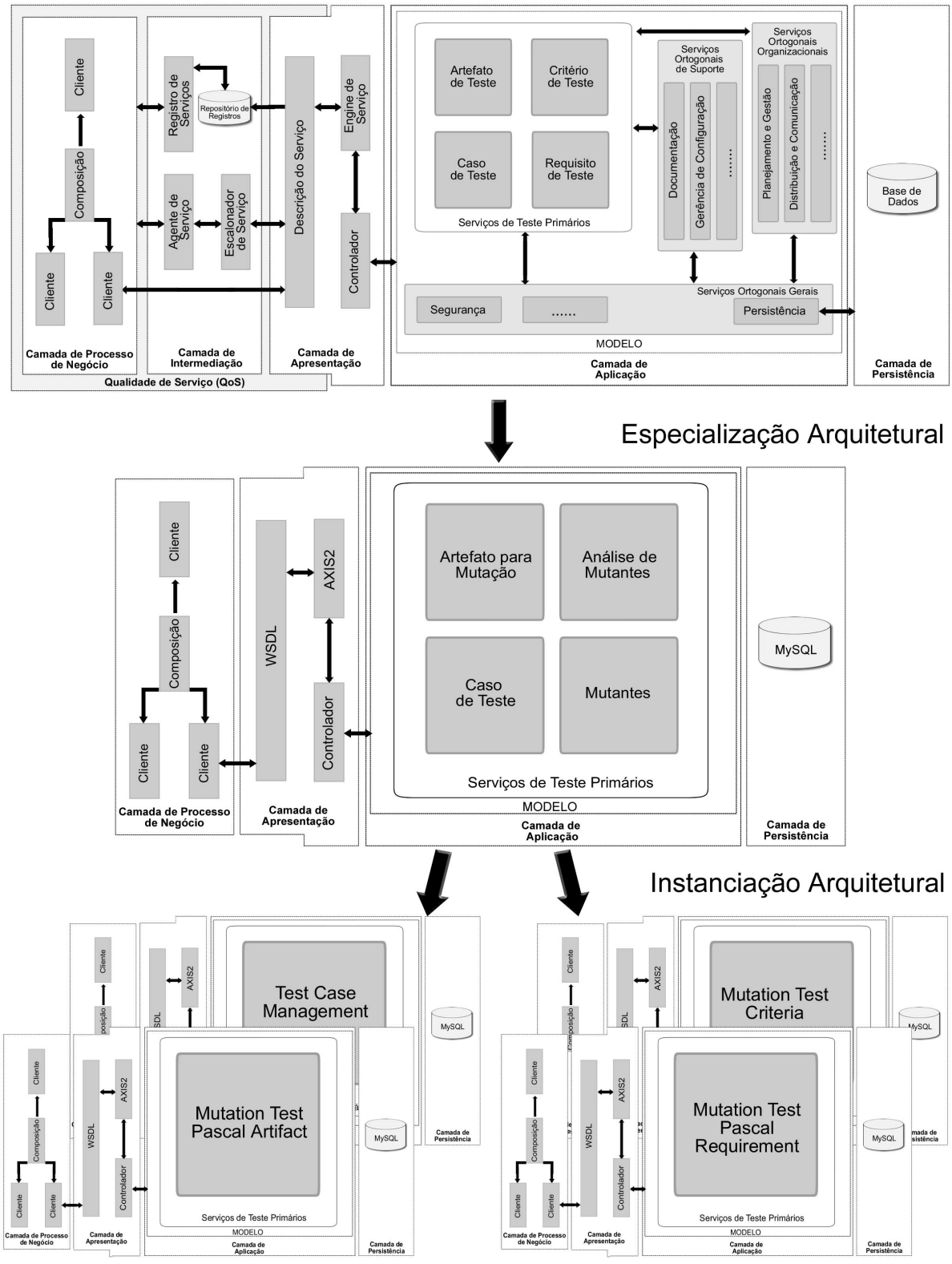

Figura 5.1: Instanciação arquitetural da RefTEST-SOA para uma ferramenta de teste de mutação

utilizados na composição de uma ferramenta de teste. Vale ressaltar também, que nesse estudo de caso as interações entre os serviços são realizadas de maneira direta, sendo previamente conhecidos os endereços dos serviços utilizados, não necessitando, portanto, da implementação de uma `Camada de Intermediação`. Além disso, a implementação da camada de `Qualidade de Serviço` também não foi considerada durante o estudo de caso.

Baseado na arquitetura resultante de uma possível especialização arquitetural da RefTEST-SOA, foram derivadas as arquiteturas dos quatro serviços de teste primários. Para implementar as funcionalidades relacionadas ao gerenciamento de `Artefato para Mutação` foi desenvolvido o serviço MuTestPascalArtifact (MTPA). No serviço MuTestPascalRequirement (MTPR) é realizado o gerenciamento dos requisitos de teste. As atividades relacionadas à `Análise de Mutantes` são implementadas pelo serviço MuTestCriteria (MTC). Por fim, o serviço TestCaseManagement (TCM) é responsável pelo gerenciamento dos casos de teste utilizados pela PascalMutants-Service. Na próxima seção, são apresentados em mais detalhes os serviços de teste primários, suas estruturas e as funcionalidades implementadas.

5.3 Descrição dos Serviços de Teste

Baseado nas diretrizes fornecidas pela RefTEST-SOA, serviços de teste que automatizam os quatro principais conceitos de teste de software – Artefato de teste, Requisito de Teste, Caso de Teste e Critério de Teste – foram implementados. Tais serviços foram desenvolvidos de forma independente, conforme a estrutura definida pela RefTEST-SOA, visando promover o aumento do reúso. A seguir, são apresentados os detalhes de projeto de cada um desses serviços.

5.3.1 Serviço de Gerenciamento de Artefatos de Teste

O serviço denominado MuTestPascalArtifact (*Mutation Test Pascal Artifact* – MTPA) é destinado ao gerenciamento de artefatos de teste relacionados ao critério Análise de Mutantes. Nesse serviço, é realizada a aquisição do código fonte a ser testado, seu tratamento e execução com os casos de teste. Na Tabela 5.1 são listadas as operações realizadas pelo serviço MuTestPascalArtifact, as funcionalidades específicas de ferramentas de teste[1] relacionadas ao conceito de Artefato de Teste, propostas pela RefTEST-SOA, e uma análise da equivalência entre elas. Na terceira coluna é analisada a equivalência entre funcionalidades e operações. O símbolo[2] (+) denota a adição de uma operação não relacionada entre

[1] As funcionalidades específicas de ferramentas de teste consideradas durante esse capítulo são as apresentadas na Tabela 4.6, uma vez que essas formam o conjunto de atividades automatizadas pelos serviços de teste primários propostos pela RefTEST-SOA.

[2] A semântica atribuída aos símbolos da Tabela 5.2 é também adotada nas Tabelas 5.2, 5.4 e 5.5.

as funcionalidades definidas na Tabela 4.6; o símbolo (✓) representa a equivalência entre uma funcionalidade específica e uma operação disponibilizada pelo serviço; e o símbolo (◦) indica uma funcionalidade específica não implementada pelo serviço.

Tabela 5.1: Equivalência entre as operações do serviço MTPA e as funcionalidades específicas de ferramentas de teste propostas pela RefTEST-SOA

Funcionalidade Específica	Operação do Serviço	Eqv
Cria um projeto no serviço de artefato de teste	createProject	+
Remove um projeto do serviço de artefato de teste	deleteProject	+
Adquire artefato a ser testado	uploadSourceCode	✓
Trata o artefato a ser testado	compileSourceCode	✓
Executa o artefato a ser testado com casos de teste	executeCode	✓
Fornece relatório de execução do artefato a ser testado	—	◦
Permite a visualização de artefato a ser testado	getOriginalCode	✓
Exclui artefato a ser testado	deleteSourceCode	✓
Inclui *drivers* e *stubs*	—	◦
Permite visualizado de *drivers* e *stubs*	—	◦

No serviço MuTestPascalArtifact foram adicionadas as operações para a criação (`createProject`) e remoção de projetos (`deleteProject`). Tais serviços têm por objetivo determinar o ciclo de vida das interação com os serviços clientes. As funcionalidades relacionadas aos *drivers* e *stubs*, bem como a geração de relatórios sobre a execução do teste, não foram implementadas no serviço desenvolvido. Entretanto, vale lembrar que a operação destinada à execução do código fonte com os casos de teste (`executeCode`) possui como informação de retorno o resultado do teste executado, possibilitando que relatórios sejam elaborados pelos serviços clientes.

A estrutura simplificada[3] do serviço implementado é apresentada na Figura 5.2. Visando auxiliar na construção do serviço desenvolvido, foram utilizados padrões de projeto. Tais padrões descrevem soluções para problemas recorrentes no desenvolvimento de sistemas e podem trazer benefícios quanto à manutenibilidade, produtividade e reúso (Gamma et al., 1995). Nesse diagrama, a representação da interface padrão do serviço é ilustrada pela interface `MTPA`. No nível de implementação, tal interface padrão é descrita em um arquivo WSDL, que é produzido automaticamente pela *engine* de serviço, com base na classe de fachada `MuPascalArtifact`[4], que é implementada segundo o padrão de projeto *Facade* (Gamma et al., 1995). Por meio dessa classe de fachada, requisições das funcionalidades são direcionadas às classes pertencentes ao pacote `Mediator`, que desempenham o papel de `Controlador` da aplicação. As classes inseridas no pacote `Mediator` são implementadas segundo o padrão *Mediator* (Gamma et al., 1995) e fazem uso das entidades do

[3]Por questão de legibilidade foram omitidos no diagrama algumas classes e também relacionamentos.

[4] O desenvolvimento de descrições de serviços a partir de classes já desenvolvidas é conhecido como abordagem *botton-up*.

pacote `Entity` e das funcionalidades disponíveis no pacote `Runner` para o desenvolvimento das regras de negócio.

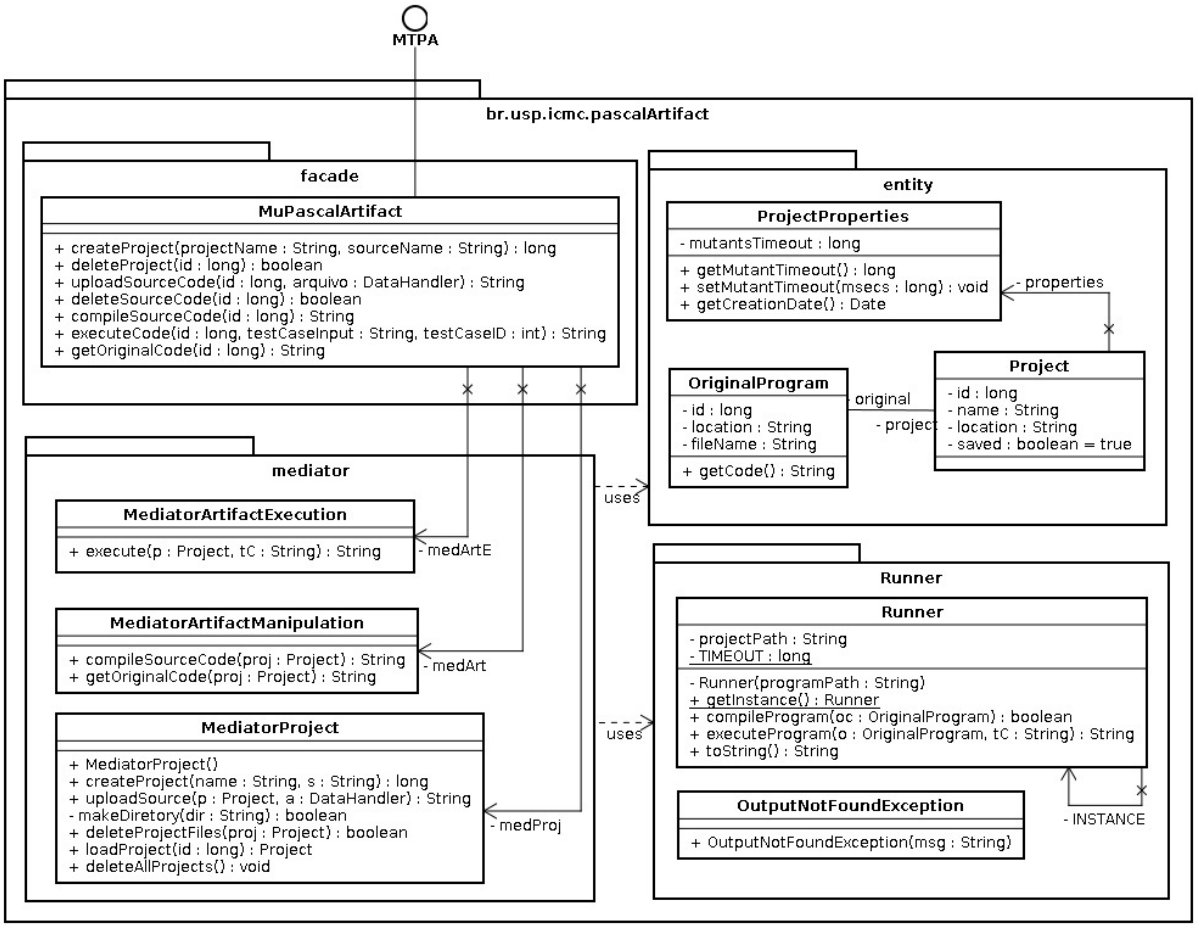

Figura 5.2: Diagrama de classes simplificado do serviço MTPA

Vale lembrar que a classe `Runner` é implementada segundo o padrão de projeto *Singleton* (Gamma et al., 1995), o que possibilita que ela seja facilmente acessada por toda a aplicação. Além disso, a utilização desse padrão garante a existência de uma única instância dessa classe, evitando que acessos simultâneos ao compilador Pascal sejam realizados.

5.3.2 Serviço de Gerenciamento de Requisitos de Teste

O serviço de gerenciamento de requisitos para o teste de mutação de programas Pascal, denominado MuTestPascalRequirements (*Mutation Test Pascal Requirements* – MTPR), destina-se à geração dos mutantes, sua compilação e execução com os casos de teste. Na Tabela 5.2 são apresentadas as funcionalidades relacionadas ao conceito de Requisito de Teste, as operações implementas pelo serviço MuTestPascalRequirements e uma análise da equivalência entre elas. De forma análoga ao serviço MTPA, foram incluídas operações para criação e remoção de projetos. Além disso, não foram implementadas operações

para a importação de requisitos de teste, coleta de *trace* e produção de relatórios. Tais operações poderão vir a ser implementadas em futuras versões desse serviço.

Tabela 5.2: Equivalência entre as operações do serviço MTPR e as funcionalidades específicas de ferramentas de teste propostas pela RefTEST-SOA

Funcionalidade Específica	Operação do Serviço	Eqv
Cria um projeto no serviço de requisito de teste	createProject	+
Remove um projeto de requisitos de teste	deleteProject	+
Importa requisitos de teste	—	∘
Gera requisitos de teste	generateMutants/ compileMutants	✓
Seleciona requisitos de teste	getAliveMutants	✓
Marca requisito de teste como não-executável	markEquivalent	✓
Desmarca requisitos de teste	unmarkEquivalent	✓
Executa requisitos de teste com casos de teste	executeMutants	✓
Coleta *trace* de execução dos requisitos	—	∘
Fornece relatório de requisitos de teste	—	∘
Permite a visualização requisitos de teste	getMutants/ getMutant	✓
Exclui requisitos de teste	removeMutant	✓

Uma representação simplificada da estrutura de classes da MuTestPascalRequirements é apresentada na Figura 5.3. De maneira análoga ao serviço MuTestPascalArtifact, foram utilizados os padrões de projeto *Facade*, *Mediator* e *Singleton*. Nesse serviço, para que os mutantes sejam gerados, no pacote `SyntacticTree`, é produzida uma árvore sintática referente ao código fonte do programa. Em seguida, são aplicados operadores de mutação para que desvios sintáticos sejam introduzidos, resultando em códigos mutantes. Os operadores de mutação utilizados pelo serviço de gerenciamento de requisitos de teste são apresentados na Tabela 5.3. É importante destacar que o conjunto de operadores utilizado, apesar de ser limitado quanto à capacidade de descoberta de defeitos, é considerado adequado para esse estudo de caso, uma vez que o foco desse estudo é observar a viabilidade de ferramentas de teste de software organizadas como serviços e baseadas na RefTEST-SOA. Após a geração dos mutantes, esses são salvos em arquivos e compilados pela classe `Runner`. Na classe `MutantComparator` é realizada a comparação entre o resultado da execução de um mutante e a saída esperada para essa execução. Caso a saída obtida seja diferente da saída esperada, o estado do mutante é alterado para morto. Em casos nos quais o código fonte do mutante possua uma alteração que resulte em um laço com um número infinito de iterações (*loop* infinito) e que inviabilize a geração de uma saída para a execução, uma exceção por "estouro de tempo" é produzida (`OutputNotFoundException`) e o mutante é considerado como morto.

Tabela 5.3: Operadores de mutação utilizados pelo serviço MTPR

Descrição do Operador de Mutação	Exemplo de Código Original	Exemplo Código Mutado
Troca de operador aritmético	a := b * c;	a := b – c;
Troca de operador relacional	if (a > b) then	if (a < b) then
Troca de operador lógico	if (a || b) then	if (a && b) then
Adição do operador `NOT` em expressões de controle	if (a == b) then	if (NOT(a == b)) then
Remoção de um comando	a := a + 1; b := a;	; b := a;
Troca de uma constante escalar por seu sucessor ou seu predecessor	const a := 6;	const a := 5;
Troca do comando `repeat` por `while` e vice-versa	repeat a:=a*c; c:=c+1; until (c<7);	while (c < 7) do begin a:=a*c; c:=c+1; end;

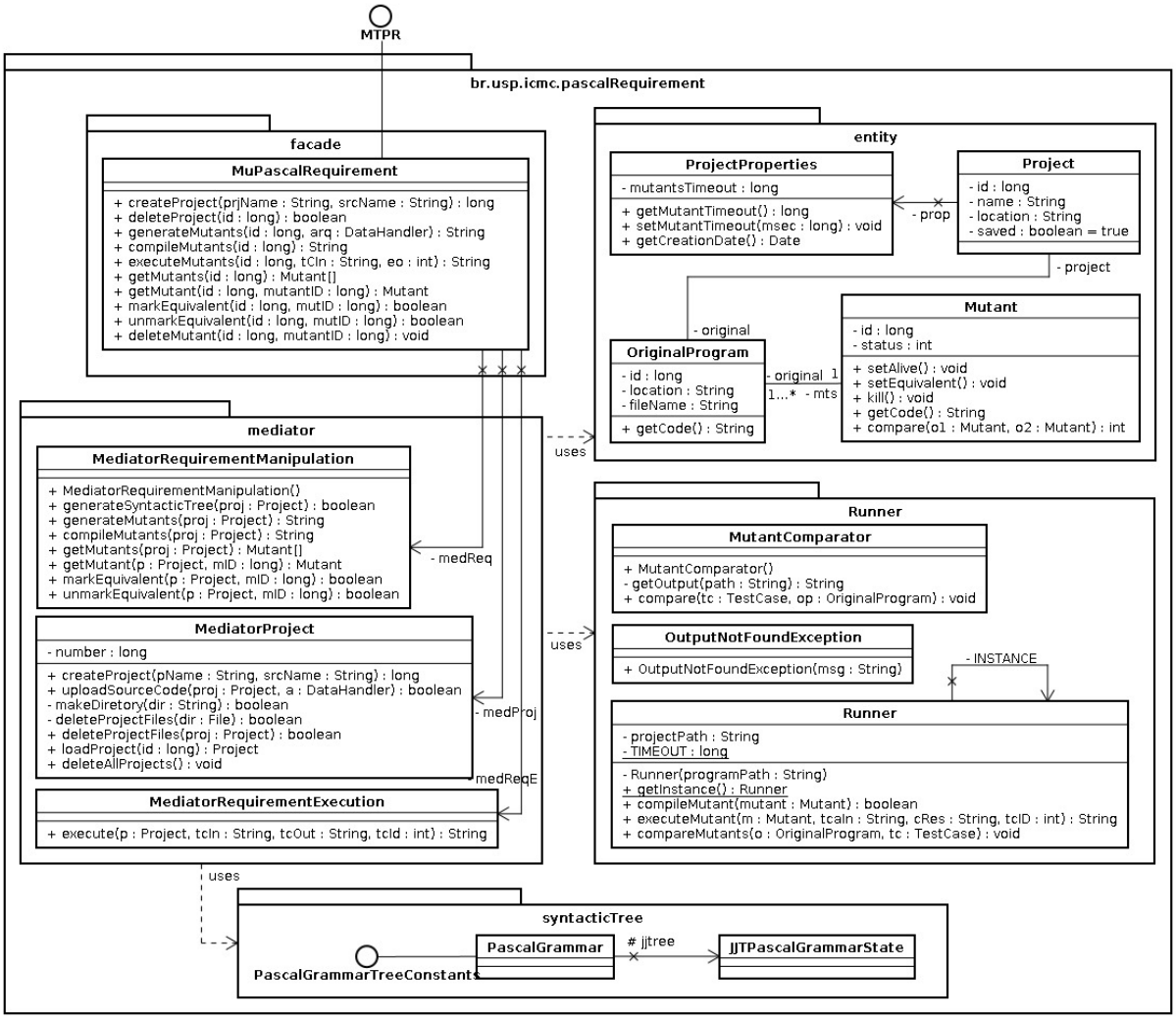

Figura 5.3: Diagrama de classes simplificado do serviço MTPR

5.3.3 Serviço de Gerenciamento de Critérios de Teste

Esse serviço, denominado MuTestCriteria (*Mutation Test Criteria* – MTC), implementa funcionalidades relacionadas ao critério Análise de Mutantes. O MuTestCriteria é destinado à verificação do número de mutantes vivos, mortos e equivalentes. Com base nessas informações, são estabelecidos também o critério de adequação e o cálculo da cobertura dos testes realizados. É por meio do critério de adequação do teste que o testador decide se novos casos de teste deverão ser adicionados ao serviço TCM, se novos casos de teste deverão ser executados nos serviços MTPA e MTPR ou se o conjunto de casos de teste é adequado. Na Tabela 5.4 são apresentadas as funcionalidades específicas relacionadas ao conceito de critério de teste, as operações disponíveis no serviço MTC e uma análise da relação entre elas. No serviço MTC, a adição de uma operação para a aquisição de informações sobre requisitos de teste (mutantes) foi necessária (`setRequirements`). Vale ressaltar que tal operação não foi considerada durante a descrição da RefTEST-SOA, pois consiste de uma decisão de projeto específica da ferramenta apresentada nesse estudo de caso. As operações de produção de relatórios e a análise estática da execução do teste não foram automatizadas por esse serviço.

Tabela 5.4: Equivalência entre as operações do serviço MTC e as funcionalidades específicas de ferramentas de teste propostas pela RefTEST-SOA

Funcionalidade Específica	Operação do Serviço	Eqv
Cria um projeto no serviço de critério de teste	createProject	+
Remove um projeto do serviço de critério de teste	deleteProject	+
Estabelece critério de adequação do teste	getMutationScore	✓
Calcula cobertura do teste	getMutationScore/ getGeneralStatistics	✓
Fornece os requisitos de teste para o cálculo da cobertura	setRequirements	+
Fornece relatório de falhas do teste	—	∘
Permite a análise estática da execução do teste pelo testador	—	∘
Informa o total de requisitos de teste analisados	getNumOfMutants	+
Informa o número de requisitos de teste ainda não cobertos	getNumOfAlives/ getAliveRatio	+
Informa o número de requisitos de teste que já foram cobertos	getNumOfDeads/ getDeadRatio	+
Informa o número de requisitos de teste considerados não executáveis	getNumOfEquivalents/ getEquivalentRatio	+

A estrutura do serviço MTC é apresentada de forma simplificada no diagrama de classes da Figura 5.4. Nesse serviço, requisições feitas por clientes são convertidas pela *engine* de serviço para que sejam recebidas pela classe de fachada `MutationTestCriteria`. Quando um serviço cliente solicita uma funcionalidade e informa ao serviço MTC o identificador do projeto que deseja acessar, essa informação é encaminhada ao `MediatorProject`. Nesse

mediador é realizada a verificação da validade do identificador recebido e a criação de um objeto da classe `Project`, referente a esse identificador. Na classe `MediatorTestCriteria` são realizados os cálculos das informações referentes ao critério Análise de Mutantes, tais como o escore de mutação e o número de mutantes vivos, mortos e equivalentes. Para isso, são utilizadas as informações contidas nas classes `Project`, `Statistics` e `Mutant`, localizadas no pacote `entity`.

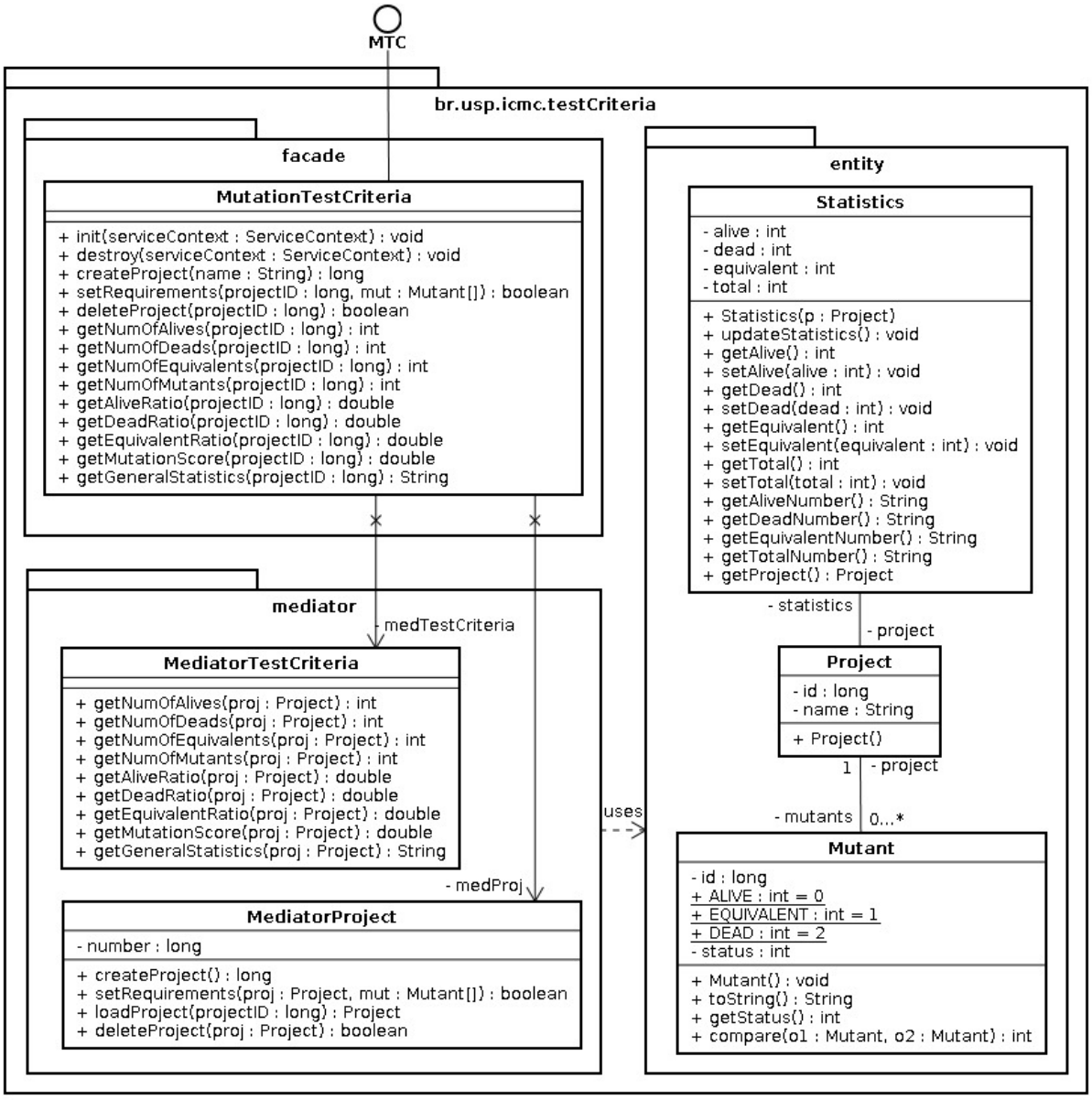

Figura 5.4: Diagrama de classes simplificado do serviço MTC

5.3.4 Serviço de Gerenciamento de Casos de Teste

O serviço TestCaseManagement (TCM) tem como principal objetivo o gerenciamento de conjuntos de casos de teste. Esse serviço disponibiliza operações destinadas à adição, remoção, edição e visualização de casos de teste pertencentes a diferentes projetos de teste. É importante destacar que o serviço de gerenciamento de casos de teste pode ser utilizado por serviços clientes que implementem diferentes técnicas e critérios de teste. Além disso, os casos de teste armazenados são independentes da linguagem de programação na qual o programa sendo testado foi desenvolvido. Na Figura 5.5 são apresentadas as operações implementadas pelo serviço, as funcionalidades relacionadas ao conceito de Caso de Teste e uma análise sobre a equivalência entre as funcionalidades previamente definidas e as que foram implementas. As funcionalidades de geração automática de casos de teste e minimização de conjuntos de casos de teste não foram implementadas nessa versão do serviço e podem ser desenvolvidas futuramente.

Tabela 5.5: Equivalência entre as operações do serviço TCM e as funcionalidades específicas de ferramentas de teste propostas pela RefTEST-SOA

Funcionalidade Específica	Operação do Serviço	Eqv
Cria um projeto no serviço de casos de teste	createProject	+
Remove um projeto do serviço de casos de teste	deleteProject	+
Importa casos de teste	addTestCase	✓
Exclui casos de teste manualmente	removeTestCase	✓
Gera casos de teste automaticamente	—	∘
Minimiza conjunto de casos de teste	—	∘
Habilita casos de teste	markNotExecuted	✓
Desabilita casos de teste	markExecuted	✓
Exporta casos de teste	getTestCases/ getNotExecutedTestCases -	✓
Editar um caso de teste	setTestCaseOutput	+
Informa sobre o número total de casos de teste	getQtdTestCases	+
Fornece relatórios de casos de teste	—	∘
Remove casos de teste	removeTestCase	✓
Permite a visualização de casos de teste	getTestCase/ getTestCases	✓

No diagrama de classes apresentado na Figura 5.5 é ilustrada, de forma simplificada, a estrutura do serviço de gerenciamento de casos de teste. Nesse serviço, solicitações de funcionalidades são recebidas pela classe de fachada `TestCaseManager`, que direciona os eventos para o controlador, composto pelas classes `MediatorProject` e `MediatorTestCase`. As classes do pacote `mediator` fazem uso das entidades (pacote `entity`) visando o gerenciamento de conjuntos de casos de teste.

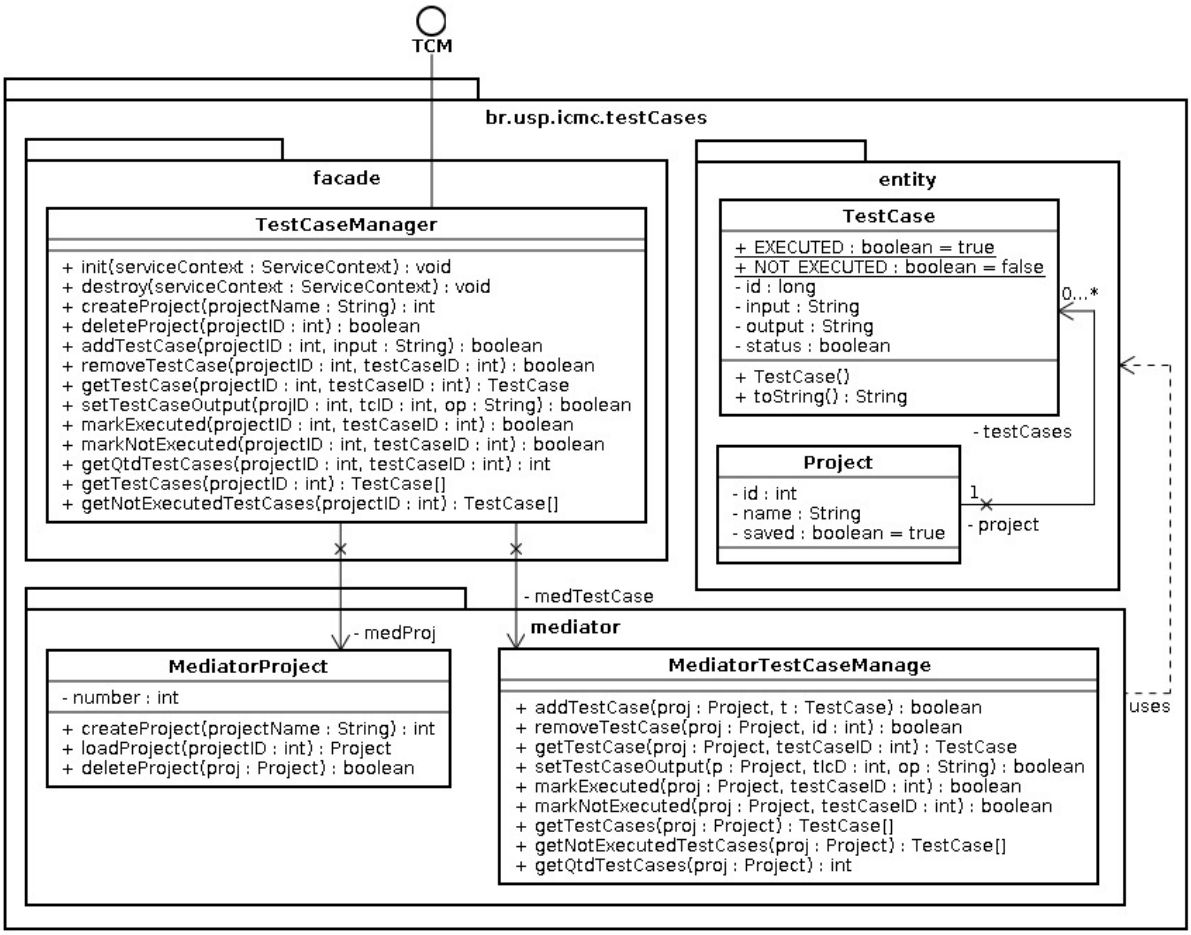

Figura 5.5: Diagrama de classes simplificado do serviço TCM

Na próxima seção é discutida a integração dos quatro serviços de teste primários apresentados. A integração desses serviços viabilizou o desenvolvimento de uma ferramenta de teste orientada a serviços, denominada PascalMutants-Service.

5.4 Integrando Serviços de Teste

A partir dos serviços de teste desenvolvidos, descritos na seção anterior, uma ferramenta de teste orientada a serviços foi implementada. A PascalMutants-Service faz uso da orquestração de serviços – na qual a interação é coordenada hierarquicamente por um dos participantes – como forma de organizar a colaboração entre os quatro serviços de teste primários. Na Figura 5.6 é ilustrado um exemplo de processo de negócio utilizado pela ferramenta PascalMutants-Service, descrito em um diagrama de atividades da UML. A representação das mensagens é feita por setas pontilhadas, a sequência de atividades por setas contínuas e os serviços de teste primários (MTPR, MTPA, TCM e MTC) por raias.

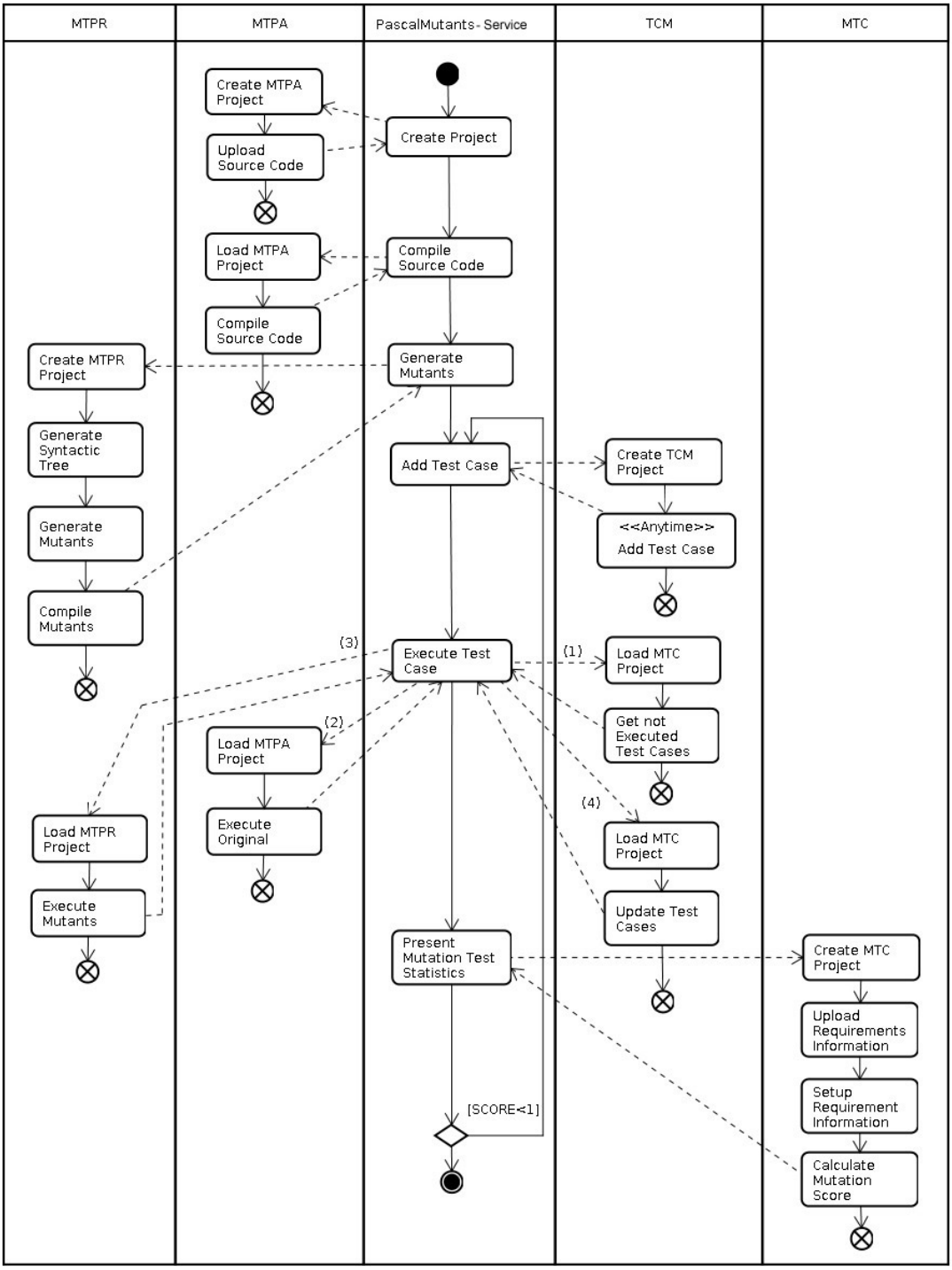

Figura 5.6: Processo de negócio da ferramenta PascalMutants-Service

No processo de negócio definido, quando um cliente (seja ele um usuário, uma aplicação ou um outro processo de negócio) cria um projeto de teste e carrega um código fonte para ser testado, a PascalMutants-Service envia uma mensagem ao serviço de artefato de teste (MTPA), requisitando o tratamento do mesmo. Em seguida, o serviço de gerenciamento de requisitos de teste (MTPR) é solicitado, visando a geração da árvore

sintática referente ao código fonte. Para que os mutantes sejam produzidos, operadores de mutação são aplicados, inserindo pequenos desvios sintáticos na árvore referente ao código fonte original. Cada uma dessas árvores é transformada em um código fonte mutante que, depois de salvo em um arquivo, é compilado pelo serviço MTPR. Casos de teste podem ser inseridos no serviço de gerenciamento de casos de teste (TCM) em qualquer momento do processo. Como forma de exercitar o programa em teste, mensagens são enviadas ao serviço (TCM), solicitando uma lista dos casos de teste ainda não executados. Cada um desses casos de teste é executado junto ao programa original. Em seguida, o serviço de gerência de requisitos de teste (MTPR) executa os mutantes ainda vivos utilizando os casos de teste. Se o resultado da execução de um mutante for diferente da saída esperada para um determinado caso de teste, o mutante é considerado morto. Para que as estatísticas do teste sejam produzidas, a PascalMutants-Service solicita ao serviço de critério de teste (MTC) que calcule o escore de mutação. Além disso, por meio do serviço MTC, outras informações sobre a cobertura dos requisitos de teste podem ser obtidas. Enquanto o conjunto de casos de teste não for considerado adequado, novos casos de teste poderão ser utilizados.

5.5 Implementação e Tecnologias Utilizadas

Com relação aos detalhes de implementação e às tecnologias utilizadas, os serviços apresentados nesse estudo de caso foram desenvolvidos por meio de serviços web, fazendo uso de linguagens e protocolos relacionados. Para o desenvolvimento dos serviços, foi utilizada a linguagem de programação Java, sendo o JavaCC[5] (*Java Compiler Compiler*) adotado como gerador de analisador sintático, por viabilizar a geração de árvores sintáticas. Para a compilação e execução dos programas escritos na linguagem Pascal, foi utilizado o GPC[6] (*GNU Pascal Compiler*). Como servidor de aplicação foi utilizado o Apache Tomcat[7], na sua versão 6.0.29, com o apoio da *engine* de serviço AXIS2.

Na Tabela 5.6 são apresentadas informações relacionadas a implementação dos cinco serviços desenvolvidos (MuTestPascalArtifact, MuTestPascalRequirement, MuTestCriteria, TestCaseManagement e PascalMutants-Service). Para a obtenção de informações sobre o número de classes e de linhas de código (LoC) implementadas foi utilizada a ferramenta Metrics[8]. Vale lembrar que, parte das LoCs referentes ao serviço MuTestPascalRequirement foi produzida com o auxílio da JavaCC.

[5] `https://javacc.dev.java.net/`
[6] `http://www.gnu-pascal.de/gpc/h-index.html`
[7] `http://tomcat.apache.org/`
[8] `http://metrics.sourceforge.net/`

Tabela 5.6: Número de classes e LoCs utilizadas na implementação dos serviços

Serviço Implementado	LoCs codificadas manualmente	LoCs produzidas automaticamente	Total de LoCs
MuTestPascalArtifact	580	–	580
MuTestPascalRequirement	1241	8.309	9.550
MuTestCriteria	411	–	411
TestCaseManagement	383	–	383
PascalMutants-Service	493	34.153	34.646
Total de LoCs	**3.108**	**42.452**	**45.560**
Serviço Implementado	Classes codificadas manualmente	Classes produzidas automaticamente	Total de classes
MuTestPascalArtifact	11	–	11
MuTestPascalRequirement	13	50	63
MuTestCriteria	7	–	7
TestCaseManagement	5	–	5
PascalMutants-Service	1	95	96
Total de classes	**37**	**145**	**182**

É de grande importância destacar que das 34.646 LoC da PascalMutants-Service, utilizadas para a integração dos serviços de teste primários, foi necessária a codificação manual de apenas 493 LoC, sendo as 34.153 restantes referentes a *stubs* produzidos automaticamente pela *engine* AXIS2. Tal fato provê indícios do aumento da produtividade na construção de ferramentas de teste, em um cenário no qual existem serviços já implementados e que podem ser reutilizados. Na Seção 5.7 são apresentadas em mais detalhes considerações a respeito do aumento da capacidade de reúso dos serviços de teste de software.

5.6 Exemplo de Utilização

Como forma de elucidar o uso da ferramenta PascalMutants-Service, nesta seção, é ilustrado o teste de um programa escrito em linguagem Pascal, denominado `Conversor`. Na Listagem 5.1 é apresentado o código fonte do programa testado. O programa `Conversor` recebe como parâmetro um número inteiro de segundos e o converte para o valor equivalente em horas, minutos e segundos (no formato "hora:minuto:segundo").

Listagem 5.1: Código fonte do programa em teste

```
1  program Conversor;
2
3  var
4    S, M, H, error: integer;
5    value : string;
6
7  begin
```

```
8
9   value := ParamStr(1);
10  Val(value, S, error);
11
12      {faz o calculo dos minutos com base nos segundos}
13      while(S >= 60) do begin
14        S := S-60;
15        M := M+1;
16      end;
17
18      {faz o calculo das horas com base nos minutos}
19      while(M >= 60) do begin
20        M := M-60;
21        H := H+1;
22      end;
23
24      write('horas:minutos:segundos ', H,': ',M,':',S);
25  end.
```

A PascalMutants-Service apresenta um menu de opções que é disponibilizado por meio de terminal de comando, como mostrado na Figura 5.7. Nesse menu, as seguintes funcionalidades foram disponibilizadas: criar um projeto de teste (*Create Test Project*); compilar o código fonte (*Compile Source Code*); gerar os mutantes para o código testado (*Generate Mutants*); visualizar os mutantes vivos (*View Alive mutants*); adicionar casos de teste (*Add Test Case*); executar os casos de teste (*Execute Test Cases*); visualizar as estatísticas do teste (*Test Statistics*); e sair da ferramenta (*End*).

```
1 - Create Test Project
2 - Compile Source Code
3 - Generate Mutants
4 - View Alive Mutants
5 - Add Test Case
6 - Execute Test Cases
7 - Test Statistics
0 - End
Option >>
```

Figura 5.7: Menu de opções para a utilização da PascalMutants-Service

Inicialmente, para que o programa `Conversor` pudesse ser testado, um projeto de teste foi criado. Após a criação desse projeto, o código fonte do programa foi inserido e compilado pelo serviço de gerenciamento de artefatos de teste (MTPA). Em seguida, foi es-

tabelecida a conexão com o serviço de gerenciamento de requisitos de teste (MTPR) e 63 mutantes foram criados e compilados, sendo que seis desses sofreram modificações que impossibilitaram a sua compilação e foram considerados mortos. Na Figura 5.8 são apresentadas as mensagens disponibilizadas pelo terminal durante a criação do projeto de teste, geração dos mutantes e também pela escolha da opção sete do menu (*Test Statistics*), que apresenta informações sobre os requisitos de teste após a compilação.

```
Project Name: ICMC
Source Path: /home/bueno/workspace/pascalTeste/
Source Name: conversor.pas

The test project has been created

The test artifact project ID (35610) has been created
The source code: conversor.pas has been uploaded
The source code has been successfully compiled

The test requirement project ID (53262) has been created
Generating mutants ...
The mutants have been generated
The mutants have been successfully compiled

The criteria project ID (51326) has been created
Getting requirements (mutants) from test service
Setting requirements (mutants) in criteria service
Making statistics of testing coverage

Project: ICMC
Alive: 57 (90,476%)
Dead: 6 (9,524%)
Equivalent: 0 (0,000%)
Number of Mutants: 63
Mutantion Score: 0,095%
```

Figura 5.8: Mensagens apresentadas após a criação de um projeto de teste

Após a geração dos mutantes, foi estabelecida a conexão entre a PascalMutants-Service e o serviço de gerenciamento de casos de teste (TCM), para que os casos de testes pudessem ser inseridos. Primeiramente, por meio da opção cinco (*Add Test Case*), foi adicionado o caso de teste `<ID:0, Input:60>`. Esse caso de teste foi executado no programa original e, em seguida, nos 57 mutantes vivos. O resultado da execução do programa original e de seus mutantes com esse caso de teste é apresentado na Figura 5.9. Nessa figura são apresentadas também informações sobre os requisitos de teste após a execução desse caso de teste.

```
Test Case Input: 60

The test case project ID (9563) has been created
The test case <60> has been added

Executing original code with test case <ID: 0, Input: 60>
      >> Execution Output =  horas:minutos:segundos 0: 1:0

Original code has been successfully executed
Test case <0> has been marked as executed
The execution output  <horas:minutos:segundos 0: 1:0>
  has been associated to test case <0>

Executing mutants with test case <ID: 0, Input: 60>
Mutants have been executed

Getting requirements (mutants) from test service
Setting requirements (mutants) in criteria service
Making statistics of testing coverage

Project ID: 0
Alive: 23 (36,508%)
Dead: 40 (63,492%)
Equivalent: 0 (0,000%)
Number of Mutants: 63
Mutantion Score: 0,635%
```

Figura 5.9: Mensagens apresentadas após a inserção e execução de um caso de teste

Como forma de satisfazer o critério de adequação, outros dez casos de teste foram adicionados e executados. Na Tabela 5.7, uma síntese dos resultados obtidos após a execução de cada caso de teste, informações sobre requisitos executados e o escore de mutação a cada etapa são apresentados. Nessa tabela, as colunas representam respectivamente: número identificador do caso de teste (ID), entrada do caso de teste, saída esperada[9], número de mutantes vivos, porcentagem de mutantes vivos, número de mutantes mortos, porcentagem de mutantes mortos, número de mutantes equivalentes, porcentagem de mutantes equivalentes e escore de mutação.

No exemplo apresentado, a aplicação dos operadores de mutação no código fonte do programa `Conversor` não resultou em mutantes equivalentes ao programa original, sendo o conjunto de casos de teste adequado e suficiente para obtenção do escore de mutação máximo (1,000). Contudo, caso uma análise da equivalência entre o código fonte original e um código mutante fosse necessária, a comparação entre eles poderia ser realizada por meio da opção quatro do menu (*View Alive mutants*).

[9]Por questão de espaço, os resultados das saídas esperadas foram abreviados. Por exemplo, a saída esperada "horas : minutos : segundos 0:1:0" é apresentada na forma " 0:1:0 ".

Tabela 5.7: Informações sobre a execução de casos dos teste no programa Conversor

ID	Entrada	Saída	Vivos	(%)	Mortos	(%)	Eq	(%)	Escore
0	0	0:0:0	44	69,841	19	30,159	0	0,000	0,302
1	1	0:0:1	40	63,492	23	36,508	0	0,000	0,365
2	20	0:0:20	40	63,492	23	36,508	0	0,000	0,365
3	59	0:0:59	39	61,905	24	38,095	0	0,000	0,381
4	60	0:1:0	22	34,921	41	65,079	0	0,000	0,651
5	61	0:1:1	21	33,333	42	66,667	0	0,000	0,667
6	2000	0:33:20	20	31,746	43	68,254	0	0,000	0,683
7	3540	0:59:0	19	30,159	44	69,841	0	0,000	0,698
8	3600	1:0:0	2	03,175	61	96,825	0	0,000	0,968
9	3660	1:1:0	1	01,587	62	98,413	0	0,000	0,984
10	3661	1:1:1	1	01,587	62	98,413	0	0,000	0,984
11	10000	2:46:40	0	00,000	63	100,000	0	0,000	1,000

5.7 Análise Preliminar do Reúso em Serviços de Teste

A PascalMutants-Service é uma ferramenta de teste de software orientada a serviço construída a partir de um processo de negócio. Em virtude disso, tal ferramenta poderá ser integrada a outras aplicações. Essa característica pode viabilizar o desenvolvimento de ambientes de teste integrados, nos quais diferentes técnicas e critérios podem ser disponibilizados e podem ser utilizados de maneira complementar. Além disso, o serviço PascalMutants-Service poderia ser agregado a Ambientes de Engenharia de Software[10], junto com outras ferramentas destinadas ao projeto, desenvolvimento e manutenção. Da mesma forma que a ferramenta de teste pode ser reutilizada como um todo, os serviços por ela compostos podem também ser reutilizados. Na Tabela 5.8 é apresentada uma análise inicial sobre a capacidade de reúso dos serviços de teste primários[11], que implementam as operações relacionadas ao critério Análise de Mutantes.

Tabela 5.8: Análise da capacidade de reúso dos serviços de teste primários

Capacidade de Reúso	TCM	MTC	MTPA	MTPR
Ferramenta Similar	✓	✓	✓	✓
Técnica/Critério	✓	–	–	–
Linguagem de Programação	✓	✓	–	–

O serviço de gerenciamento de casos de teste (TCM) foi implementado para ser independente da técnica ou do critério de teste, bem como da linguagem na qual o programa testado é desenvolvido. Dessa forma, o serviço TCM apresenta uma boa capacidade de

[10]Entende-se por Ambientes de Engenharia de Software a coleção integrada de ferramentas criadas para auxiliar no processo de desenvolvimento de sistemas de software.

[11]Nessa tabela são adotadas as mesmas siglas utilizadas para descrever os serviços de teste primários apresentados ao longo da Seção 5.3. São essas: TestCaseManagement (TCM), MuTestCriteria (MTC), MuTestPascalArtifact (MTPA) e MuTestPascalRequirement (MTPR).

reúso e poderia ser utilizado por ferramentas que dão suporte a diferentes técnicas e critérios de teste e que requerem funcionalidades de gerenciamento de casos de teste. O serviço de gerenciamento de critério de teste (MTC) gerencia informações relacionadas a um tipo específico de requisito de teste que, nesse caso, é representado pelos códigos mutantes de uma aplicação. Dessa forma, esse serviço pode ser utilizado por ferramentas de teste que automatizem a aplicação do critério Análise de Mutantes, no entanto, para o teste de programas em diferentes linguagens de programação. Além desses serviços, observa-se que os serviços que gerenciam artefatos de teste e requisitos de teste são diretamente relacionados ao critério, à técnica e à linguagem de programação utilizada para escrever o programa a ser testado. Assim, esses serviços poderiam ser reutilizados em ferramentas similares, que automatizem o critério de Análise de Mutantes para o teste de programas escritos na linguagem Pascal, mas que sejam baseadas em diferentes composições ou arquiteturas, desenvolvidas sobre infraestruturas distintas.

Na Figura 5.10 é ilustrado, de maneira geral, um exemplo da forma na qual serviços primários de teste poderiam ser reutilizados por diferentes ferramentas de teste. Nesse exemplo, quatro ferramentas de teste são consideradas: (A) ferramenta de teste de mutação para programas escritos na linguagem Pascal; (B) ferramenta de teste estrutural para programas escritos na linguagem Java; (C) ferramenta de teste estrutural de programas escritos na linguagem C++; e (D) ferramenta de teste de mutação para programas escritos na linguagem Java.

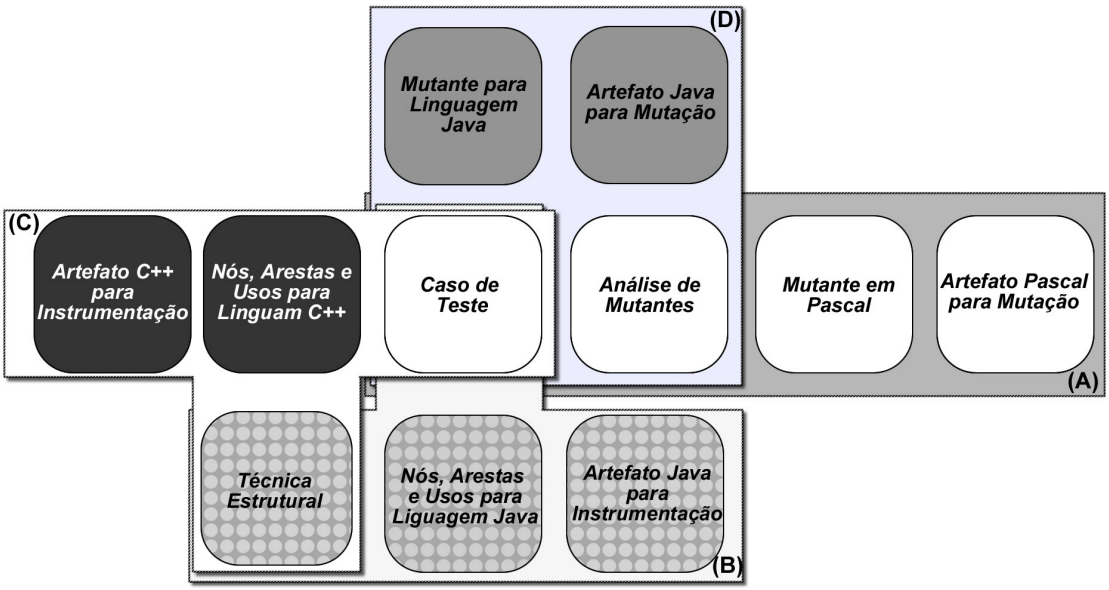

Figura 5.10: Representação geral do reúso de serviços de teste segundo a RefTEST-SOA

Apesar dessa ilustração representar uma análise preliminar da capacidade de reúso, é possível notar que serviços como o de gerenciamento de casos de teste poderiam ser utilizados pelas quatro diferentes ferramentas (A, B, C e D). Além disso, os serviços

relacionados ao critério de teste poderiam ser compartilhados entre diferentes ferramentas, como por exemplo, as que automatizem a Técnica Estrutural (ferramentas B e C) ou o critério Análise de Mutantes (ferramentas A e D). É importante ressaltar que estudos mais aprofundados sobre os benefícios da reutilização de serviços de teste ainda necessitam ser conduzidos.

5.8 Considerações Finais

A utilização da SOA no projeto de ferramentas de teste poderia contribuir para a capacidade de integração e reúso, aumentando a produtividade durante a fase de desenvolvimento. Nesse capítulo foram discutidos os detalhes relacionados ao projeto e implementação da PascalMutants-Service, uma ferramenta de teste software orientada a serviço que tem como base a arquitetura de referência RefTEST-SOA. O desenvolvimento dessa ferramenta apresentou evidências da aplicabilidade da arquitetura proposta e, em uma análise preliminar, indicou que sua utilização pode trazer benefícios quanto ao aumento da capacidade de reúso e integração. Vale ressaltar que esse mesmo estudo de caso poderia ser estendido para ferramentas de teste mais completas e que implementem critérios de teste relacionados a outras técnicas de teste, tais como a técnica estrutural e a funcional. Dentre as limitações e ameaças a validade desse estudo de caso destacam-se, respectivamente, a não utilização de métodos de engenharia de software experimental e a refatoração de uma ferramenta na qual o autor foi um dos desenvolvedores. No próximo capítulo, são discutidas em mais detalhes as conclusões sobre este trabalho.

Capítulo

6

Conclusão

Neste capítulo são retomados brevemente o contexto no qual este trabalho se insere e suas principais contribuições. São apresentadas também, as dificuldades encontradas, limitações referentes a esta pesquisa e perspectivas de trabalhos futuros.

6.1 Caracterização da Pesquisa Realizada

A atividade de teste é fundamental para o aumento da qualidade dos produtos de software desenvolvidos. Nesse contexto, a utilização de ferramentas de teste de software, que automatizem diferentes técnicas e critérios de teste, pode aprimorar a qualidade dos sistemas sendo desenvolvidos. Entretanto, muitas das ferramentas de teste são produzidas de maneira isolada, possuindo problemas relacionados à integração e uma consequente limitação quanto à capacidade de reúso, o que pode estar dificultando o desenvolvimento de ambientes de teste mais completos e eficazes. Dessa forma, esforços no sentido de estabelecer arquiteturas de software adequadas a essas ferramentas possuem grande importância e podem auxiliar no desenvolvimento de ferramentas mais facilmente integráveis, reusáveis e escaláveis.

Recentemente, a SOA tem emergido como um estilo arquitetural que visa o desenvolvimento de sistemas fracamente acoplados, que podem se comunicar por meio de protocolos padrão. Isso tem viabilizado as interações entre sistemas desenvolvidos em diferentes linguagens de programação e executados em plataformas de software distintas. Esse fato têm

atraído atenção de pesquisadores de teste de software, que vêm buscado disponibilizar suas ferramentas de teste segundo esse estilo arquitetural, o que pode trazer benefícios quanto à capacidade de integração, reúso e escalabilidade das mesmas. Em especial, observa-se que o estabelecimento de arquiteturas de referência possui particular importância, uma vez que essas contêm o conhecimento de um dado domínio e auxiliam no entendimento acerca do desenvolvimento de um conjunto de sistemas para esse domínio. Dessa forma, uma arquitetura de referência que auxilie no desenvolvimento de ferramentas de teste orientadas a serviço constitui um importante ponto de partida em direção ao aumento da capacidade de integração, reúso e escalabilidade dessas ferramentas.

6.2 Contribuições

Como principal contribuição deste trabalho, destaca-se o estabelecimento de uma arquitetura de referência orientada a serviço, que objetiva prover diretrizes ao desenvolvimento de ferramentas de teste de software que tenham como base a SOA. É importante destacar que, até onde se tem conhecimento, a arquitetura de referência proposta é a única a dar base ao desenvolvimento de ferramentas de teste orientadas a serviço, um tema considerado recente e que tem recebido atenção por parte dos pesquisadores. Além disso, quatro serviços de teste foram desenvolvidos e podem ser utilizados na construção de outras ferramentas de teste orientadas a serviço. Os detalhes do projeto e implementação desses serviços podem, inclusive, servir como um exemplo concreto da instanciação de uma arquitetura de referência para a construção de ferramentas desse domínio.

Além da contribuição principal, pode ser considerado como contribuição deste trabalho o conjunto de requisitos arquiteturais de sistemas orientados a serviço, — apresentados na Seção 4.3.2 — que pode servir de base para o desenvolvimento de outros sistemas orientados a serviço ou para o estabelecimento de arquiteturas de referência para outros domínios. Além disso, uma outra contribuição no contexto deste trabalho foi a apresentação de um panorama geral e detalhado dos modelos e arquiteturas de referência orientados a serviço disponíveis na literatura. Tal panorama foi estabelecido por meio de uma revisão sistemática, o que viabiliza sua atualização e replicação futura. Essa revisão sistemática foi publicada em um relatório técnico e pode vir a servir como base para a condução de outras revisões sistemáticas.

De uma maneira geral, buscou-se com o desenvolvimento deste trabalho contribuir para a comunidade de Teste de Software com uma arquitetura que seja base para o desenvolvimento de ferramentas de teste orientadas a serviço. Buscou-se também contribuir para a comunidade de Arquitetura de Software, como a apresentação de um panorama sobre arquiteturas e modelos de referência baseados na SOA.

6.3 Dificuldades e Limitações

Uma das dificuldades encontradas durante o desenvolvimento deste trabalho foi a necessidade da investigação de diferentes temas, de forma relativamente aprofundada. Foi necessária a investigação de temas variados, tais como: teste de software, arquitetura de software, arquitetura de referência e arquitetura orientada a serviço. Outra dificuldade encontrada foi a escassez de arquiteturas concretas e ferramentas de teste de software desenvolvidas com base na SOA, que pudessem prover um conjunto mais concreto de funcionalidades a serem representadas pela arquitetura de referência proposta. Em solução a isso, uma análise mais abrangente foi realizada, sendo consideradas também, ferramentas de verificação e análise de software, bem como diretrizes ao desenvolvimento de sistemas baseados na SOA.

As limitações deste trabalho estão relacionadas, principalmente, à validação da arquitetura de referência proposta. Com relação ao estudo de caso, ferramentas de teste de software que automatizassem outras técnicas e critérios poderiam ter sido implementadas. Além disso, métodos de avaliação arquitetural poderiam ser adaptados e utilizados, como forma de evidenciar a qualidade da arquitetura de referência proposta. Por fim, uma vez que este trabalho propõe uma arquitetura de referência dirigida à pesquisa, alguns conceitos relacionados ao domínio podem não ter sido previstos durante o estabelecimento. Entretanto, atualizações na arquitetura de referência poderão ser realizadas a medida que novas ferramentas de teste forem desenvolvidas, sendo essa uma prática prevista em estudos como o de Angelov et al. (2008).

6.4 Trabalhos Futuros

Visando a dar continuidade ao trabalho desenvolvido, a seguir são apresentadas as principais perspectivas de trabalhos futuros relacionados a este trabalho:

Implementação de uma Interface Web para Utilização da Ferramenta de Teste Desenvolvida: Com o objetivo de prover um meio de interação que permita um acesso mais fácil à ferramenta desenvolvida no estudo de caso, pode-se desenvolver uma interface web para essa ferramenta. Por meio dessa interface, a ferramenta poderá ser facilmente acessada e utilizada por testadores e alunos de disciplinas relacionadas à programação e ao teste de software;

Avaliação da Qualidade da Arquitetura de Referência Proposta: Visando conduzir uma avaliação mais criteriosa da RefTEST-SOA, métodos de avaliação arquitetural, como o SAAM e o ATAM, poderão ser adaptados e aplicados. A utilização

de tais métodos de avaliação arquitetural têm por objetivo aprimorar a qualidade da arquitetura de referência proposta. Além disso, por meio da aplicação desses métodos de avaliação, subsídios para o refinamento e reestruturação da RefTEST-SOA poderão ser obtidos;

Desenvolvimento de Outras Ferramentas de Teste de Software Orientadas a Serviço: Como forma de melhor avaliar a arquitetura de referência proposta tem-se, como trabalho futuro, o desenvolvimento de ferramentas que automatizem outras técnicas e critérios de teste. Pretende-se, inclusive, como o apoio da RefTEST-SOA, a reestruturação de uma ferramenta destinada ao teste estrutural de programas escritos em linguagem Pascal, para que essa seja adequada à SOA. Tal ferramenta poderá ser utilizada em paralelo à PascalMutants-Service, podendo inclusive compartilhar uma mesma interface web, visando aprimorar a qualidade do teste realizado;

Elaboração de uma Análise Quantitativa sobre o Aumento da Capacidade de Reúso: Baseado no desenvolvimento de diferentes ferramentas de teste de software, uma análise quantitativa do aumento da capacidade de reúso proporcionado pela utilização da RefTEST-SOA poderá ser realizada. Por meio dessa análise, será possível se ter evidências dos benefícios da utilização da arquitetura de referência proposta no desenvolvimento de ferramentas de teste orientadas a serviço;

Evolução da Arquitetura de Referência Proposta: Pretende-se evoluir a arquitetura de referência proposta com base na experiência obtida por meio da sua aplicação no desenvolvimento de ferramentas de teste orientadas a serviço. Com isso, objetiva-se aprimorar a RefTEST-SOA quanto à capacidade de apoiar o desenvolvimento de novas ferramentas de teste; e

Publicação dos Serviços Desenvolvidos em um Registro de Serviços de Ferramentas de Teste: Para que os serviços de teste desenvolvidos segundo a arquitetura proposta possam ser publicados e descobertos, um registro de serviços pode ser utilizado. Nesse contexto, pode-se fazer uso de um registro de serviços específico para ferramentas de teste de software, como o proposto por Gondim (2010), para armazenar as descrições dos serviços desenvolvidos.

Referências

Agrawal, H.; Alberi, J. L.; Horgan, J. R.; Li, J. J.; London, S.; Wong, W. E.; Ghosh, S.; Wilde, N. Mining system tests to aid software maintenance. *IEEE Computer*, v. 31, p. 64–73, 1998.

Agrawal, H.; DeMillo, R. A.; Hathaway, R.; Hsu, W.; Hsu, W.; Krauser, E. W.; Martin, R. J.; Mathur, A. P.; Spafford, E. H. *Design of mutant operators for the C programming language*. Tech. Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette/IN - USA, 1989.

Andriole, S. J. *Software validation, verification, testing and documentation: A source book*. Princeton, NJ, USA: Petrocelli Books, Inc., 1986.

Angelov, S.; Grefen, P. An e-contracting reference architecture. *The Journal of Systems and Software*, v. 81, n. 11, p. 1816–1844, 2008.

Angelov, S.; Grefen, P. W. P. J.; Greefhorst, D. A classification of software reference architectures: Analyzing their success and effectiveness. In: *Proceedings of the 8th Working IEEE/IFIP Conference on Software Architecture (WICSA 2009)*, Cambridge, UK, 2009, p. 141–150.

Angelov, S.; Trienekens, J. J.; Grefen, P. Towards a method for the evaluation of reference architectures: Experiences from a case. In: *Proceedings of the 2nd European Conference on Software Architecture (WICSA/ECSA 2008)*, Paphos, Cyprus: Springer-Verlag, 2008, p. 225–240 (LNCS v.5292).

ANSI/IEEE *Recommended Practice for architectural description of software intensive systems (ANSI/IEEE 1471)*. Standard 1471/2000, American National Standards Institute (ANSI)/ Institute of Electric and Electronic Engineers (IEEE), 2000.

Apache Foundation Apache Axis2 User's Guide. Online, `http://ws.apache.org/axis2` - Acessado em 03/12/2010, 2010.

Arsanjani, A.; Zhang, L.-J.; Ellis, M.; Allam, A.; Channabasavaiah, K. S3: A service-oriented reference architecture. *IT Professional*, v. 9, n. 3, p. 10–17, 2007.

Babar, M.; Gorton, I. Comparison of scenario-based software architecture evaluation methods. In: *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, Busan, Korea, 2004, p. 600–607.

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 2008)*, Washington, DC, USA: IEEE Computer Society, 2008, p. 465–472.

Baldamusa, M.; Bengtsona, J.; Ferrari, G.; Raggi, R. Web services as a new approach to distributing and coordinating semantics-based verification toolkits. In: *Proceedings of the 1st International Workshop on Web Services and Formal Methods (WSFM 2004)*, Pisa, Italy, 2004.

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 and Knowledge Engineering (SEKE 2006)*, San Francisco Bay, USA, 2006.

Barcelos, R. F.; Travassos, G. H. ArqCheck: Uma abordagem para inspeção de documentos arquiteturais baseada em checklist. In: *Anais do 5º Simpósio Brasileiro de Qualidade de Software (SBQS 2006)*, Vila Velha, ES, 2006a, p. 175–189.

Barcelos, R. F.; Travassos, G. H. Evaluation approaches for software architectural documents: a systematic review. In: *Proceedings of the 9th Ibero-American Workshop on Requirements Engineering and Software Environments (IDEAS 2006)*, La Plata, Argentina, 2006b, p. 433–446.

Bartolini, C.; Bertolino, A.; Marchetti, E. Introducing service-oriented coverage testing. In: *Proceedings of the 23rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2008)*, L'Aquila, Italy: IEEE, 2008, p. 57–64.

Bass, L.; Clements, P.; Kazman, R. *Software architecture in practice*. Addison-Wesley, 2003.

Bengtsson, P.; Lassing, N.; Bosch, J.; van Vliet, H. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, v. 69, n. 1-2, p. 129–147, 2004.

Biolchini, J.; Mian, P. G.; Natali, A. C. C.; Travassos, G. H. *Systematic review in software engineering.* Tech. Report RT-ES 679/05, Departamento de Engenharia de Sistemas e Ciência da Computação, COPPE/UFRJ, Rio de Janeiro, RJ, 2005.

Boehm, B. A view of 20^{th} and 21^{st} century software engineering. In: *Proceedings of the 28^{th} International Conference on Software Engineering (ICSE 2006)*, New York, NY, USA: ACM, 2006, p. 12–29.

Borges, K. N.; Ramos, F. S.; Maldonado, J. C.; Chaim, M. L.; Jino, M. Poke-Tool versão Clipper - Uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados. In: *Sessão de Ferramentas do 9º Simpósio Brasileiro de Engenharia de Software (SBES 1995)*, Recife, PE, 1995.

Brehm, N.; Gómez, J. M. Web service-based specification and implementation of functional components in federated ERP-systems. In: *Proceedings of the 10^{th} International Conference on Business Information Systems (BIS 2007)*, Poznan, Poland: Springer-Verlag, 2007, p. 134–146 (LNCS v.4439).

Budd, T. A. Mutations analysis: Ideas, examples and prospects. Computer Program Testing North-Holand Publishing Company, p. 129-148, 1981.

Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *Pattern-oriented software architecture: A system of patterns.* John Wiley & Sons, Inc., 1996.

Canfora, G.; Di Penta, M.; Esposito, R.; Villani, M. L. An approach for QoS-aware service composition based on genetic algorithms. In: *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO 2005)*, New York, NY, USA: ACM, 2005, p. 1069–1075.

Canfora, G.; Di Penta, M.; Esposito, R.; Villani, M. L. A framework for QoS-aware binding and re-binding of composite web services. *Journal of Systems and Software*, v. 81, n. 10, p. 1754–1769, 2008.

Cangussu, J. W.; DeCarlo, R. A.; Mathur, A. P. A formal model of the software test process. *IEEE Transactions on Software Engineering*, v. 28, n. 8, p. 782–796, 2002.

Cangussu, J. W.; DeCarlo, R. A.; Mathur, A. P. Using sensitivity analysis to validate a state variable model of the software test process. *IEEE Transactions on Software Engineering*, v. 29, p. 430–443, 2003.

Chaim, M. L. *Poke-Tool – Uma ferramenta para o suporte ao teste estrutural de programas baseados em análise de fluxo de dados.* Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas, SP, 1991.

Choi, H.; Lim, C.; Kim, J. Defining reference architecture for NTIS development. In: *Proceedings of the 11th International Conference on Advanced Communication Technology (ICACT 2009)*, Phoenix Park, Korea, 2009, p. 284–287.

Chusho, T. Test data selection and quality estimation based on the concept of essential branches for path testing. *IEEE Transaction on Software Engineering*, v. 13, n. 5, p. 509–517, 1987.

Clements, P.; Kazman, R.; Klein, M. *Evaluating software architectures: Methods and case studies*. The SEI Series in Software Engineering. Boston, MA: Addison-Wesley, 2002.

Costagliola, G.; Ferrucci, F.; Fuccella, V. SCORM run-time environment as a service. In: *Proceedings of the 6th International Conference on Web engineering (ICWE 2006)*, New York, NY, USA, 2006, p. 103–110.

Costagliola, G.; Ferrucci, F.; Fuccella, V. Boosting the adoption of computer managed instruction functionalities in e-learning systems. *Journal of Web Engineering*, v. 7, n. 1, p. 42–69, 2008.

Cugola, G.; Ghezzi, C. Software processes: a retrospective and a path to the future. *Software Process: Improvement and Practice*, v. 4, n. 3, p. 101–123, 1998.

Curbera, F.; Khalaf, R.; Mukhi, N.; Tai, S.; Weerawarana, S. The next step in web services. *Communications of the ACM*, v. 46, n. 10, p. 29–34, 2003.

Dai, G.; Bai, X.; Wang, Y.; Dai, F. Contract-based testing for web services. In: *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, Washington, DC, USA, 2007, p. 517–526.

Dan, A.; Davis, D.; Kearney, R.; Keller, A.; King, R.; Kuebler, D.; Ludwig, H.; Polan, M.; Spreitzer, M.; Youssef, A. Web services on demand: WSLA-driven automated management. *IBM Systems Journal*, v. 43, n. 1, p. 136–158, 2004.

Delamaro, M. E. *Proteum: Um ambiente de teste baseado em análise de mutantes*. Dissertação de Mestrado, ICMC/USP, São Carlos, SP, 1993.

Delamaro, M. E. *Mutação de Interface: Um critério de adequação interprocedimental para o teste de integração*. Tese de Doutoramento, IFSC/USP, São Carlos, SP, 1997.

Delamaro, M. E.; Maldonado, J. C.; Jino, M. *Introdução ao teste de software*. Ed. Campus, 2007.

Delamaro, M. E.; Maldonado, J. C.; Vincenzi, A. M. R. Proteum/IM 2.0: An integrated mutation testing environment. In: *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION 2000)*, San Jose, CA, USA: Kluwer Academic Publishers, 2000, p. 91–101.

DeMillo, R. A.; Lipton, R. J.; Sayward, F. G. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, v. 11, n. 4, p. 34–43, 1978.

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.

Dieste, O.; López, M.; Ramos, F. Formalizing a systematic review process in requirements engineering. In: *Proceedings of the 11th Wokshop on Requirements Engineering (WER 2007)*, Pernambuco, Brasil, 2007, p. 96–103.

Dieste, O.; López, M.; Ramos, F. Formalizing a systematic review updating process. In: *Proceedings of the 6th International Conference on Software Engineering Research, Management and Applications (SERA 2008)*, Washington, DC, USA: IEEE Computer Society, 2008, p. 143–150.

Dijkstra, E. W. *A discipline of programming.* Prentice Hall, 1976.

Dillon, T. S.; Wu, C.; Chang, E. Reference architectural styles for service-oriented computing. In: *Proceedings of the 4th International Conference of Network and Parallel Computing (NPC 2007)*, Dalian, China: Springer-Verlag, 2007, p. 543–555 (LNCS v. 4672).

Dobrica, L.; Niemel, E. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, v. 28, n. 7, p. 638–653, 2002.

Duro, N.; Moreira, F.; Rogado, J.; Reis, J.; Peccia, N. Technology harmonization - developing a reference architecture for the ground segment software. In: *IEEE Aerospace Conference*, Big Sky, Montana, USA, 2005, p. 3968–3979.

Eckerson, W. W. Three tier client/server architecture: Achieving scalability, performance, and efficiency in client server applications. *Open Information Systems Journal*, v. 10, n. 1, 1995.

Eickelmann, N.; Richardson, D. An evaluation of software test environment architectures. In: *Proceedings of the 18th International Conference on Software Engineering (ICSE 1996)*, Berlin, Germany, 1996.

Eler, M. M.; Delamaro, M. E.; Maldonado, J. C.; Masiero, P. C. Built-in structural testing of web services. In: *Anais do 24º Simpósio Brasileiro de Engenharia de Software (SBES 2010), Congresso Brasileiro de Software: Teoria e Prática (CBSoft 2010)*, Salvador, BA, 2010, p. 71–80.

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. JaBUTiService: A Web Service for Structural Testing of Java Programs. In: *Proceedings of the 33rd Annual IEEE Software Engineering Workshop (SEW 2009)*, Skövde, Sweden, 2009, p. 1–9.

Erl, T. *Service-oriented architecture: Concepts, technology, and design.* Upper Saddle River, New Jersey, USA: Prentice Hall., 2005.

Fabbri, S. C. P. F.; Maldonado, J. C.; Masiero, P. C.; Delamaro, M. E. Mutation analysis for finite state machines. In: *Proceedings of 5th International Symposium on Software Reliability Engineering (ISSRE 1994)*, Monterey, CA, USA, 1994, p. 220–229.

Fabbri, S. C. P. F.; Maldonado, J. C.; Sugeta, T.; Masiero, P. C. Mutation testing applied to validate specifications based on statecharts. In: *Proceedings of 10th International Symposium on Software Reliability Engineering (ISSRE 1999)*, Boca Raton, Florida, USA, 1999, p. 210–219.

Ferrari, F. C. *Apoio ao Teste Estrutural e de Mutação de Software Orientado a Objetos e a Aspectos.* Plano de Trabalho de Doutorado (trabalho em andamento), ICMC/USP, São Carlos, SP, 2005.

Ferrari, F. C.; Nakagawa, E. Y.; Rashid, A.; Maldonado, J. C. Automating the mutation testing of aspect-oriented Java programs. In: *Proceedings of the 5th Workshop on Automation of Software Test (AST 2010)*, New York, NY, USA: ACM, 2010, p. 51–58.

Fioravanti, F.; Spinu, M.; Campanai, M. AXMEDIS as the service oriented architecture for the media: Is it feasible? In: *Proceedings of the 3rd International Conference on Automated Production of Cross Media Content for Multi-Channel Distribution (AXMEDIS 2007)*, Washington, DC, USA: IEEE Computer Society, 2007, p. 264–269.

Fioravanti, M. L.; Nakagawa, E. Y.; Barbosa, E. F. EDUCAR: uma arquitetura de referência para ambientes educacionais. In: *Anais do 21º Simpósio Brasileiro de Informática na Educação (SBIE 2010)*, João Pessoa, Paraíba, 2010, p. 1–10.

Fonseca, R. P. *Suporte ao teste estrutural de programas Fortran no ambiente Poke-Tool.* Dissertação de Mestrado, DCA/FEEC/UNICAMP, Campinas, SP, 1993.

Frankl, P.; Weyuker, E. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, v. 14, n. 10, p. 1483–1498, 1988.

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

Gallagher, B. P. *Using the architecture tradeoff analysis method to evaluate a reference architecture: A case study.* Tech. Report CMU/SEI-2000-TN-007, Software Engineering Institute (SEI), 2000.

Gamma, E.; Beck, K. JUnit home page. Online, `http://www.junit.org/` - Acessado em 03/12/2010, 2010.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design patterns: Elements of reusable object-oriented software.* Boston, MA: Addison-Wesley, 1995.

Gao, J.; Chen, C.; Toyoshima, Y.; Leung, D. K. Developing an integrated testing environment using the world wide web technology. In: *Proceedings of the 21st International Computer Software and Applications Conference (COMPSAC 1997)*, Washington, DC, USA: IEEE Computer Society, 1997, p. 594–601.

Garlan, D. Software architecture: a roadmap. In: *Proceedings of the Conference on The Future of Software Engineering (ICSE 2000)*, New York, NY, USA: ACM Press, 2000, p. 91–101.

Ghezzi, G.; Gall, H. Towards software analysis as a service. In: *Proceedings of the 23rd ACM/IEEE International Conference on Automated Software Engineering (ASE Workshops 2008)*, L'Aquila, Italy: IEEE, 2008, p. 1–10.

Gill, A. *Introduction to the theory of finite-state machines.* New York: McGraw-Hill, 1962.

Gondim, R. P. *Desenvolvimento e avaliação de um registro de serviços de ferramentas de teste.* Dissertação de Mestrado, ICMC/USP, São Carlos, SP, 2010.

Graaf, B.; van Dijk, H.; van Deursen, A. Evaluating an embedded software reference architecture – industrial experience report. In: *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR 2005)*, Manchester, UK, 2005, p. 354–363.

Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, v. 8, n. 3, p. 231–274, 1987.

Harrold, M. J. Testing: A roadmap. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, New York, NY, USA: ACM Press, 2000, p. 61–72.

Hemalatha, T.; Athisha, G.; Jeyanthi, S. Dynamic web service based image processing system. In: *Proceedings of the 16th International Conference on Advanced Computing and Communications (ADCOM 2008)*, Chennai, India, 2008, p. 323–328.

Hewlett-Packard HP QuickTest Professional software. Online, `https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1352_4000_100` - Acessado em 03/12/2010, 2010.

Howden, W. E. *Software engineering and technology: Functional program testing and analysis*. New York: McGrall-Hill, 1987.

IEEE *Standard Glossary of Software Engineering Terminology (IEEE 610.12)*. Standard 610.12/1990, Institute of Electric and Electronic Engineers (IEEE), 1990.

Ionita, M.; Hammer, D.; Obbink, H. Scenario-based software architecture evaluation methods: An overview. In: *Proceedings of the Workshop on Methods and Techniques for Software Architecture Review and Assessment*, Orlando, Florida, 2002, p. 1–12.

ISO *Information Technology – Software Life-cycle Processes (ISO 12207)*. Standard 12207/1995, International Organization for Standardization (ISO), 1995.

ISO *Quality Vocabulary (ISO 8402, Part of the ISO 9000/2002)*. Standard 8402/2002, International Organization for Standardization (ISO), 2002.

ISO/IEC *Recommended Practice for Architectural Description of Software-Intensive Systems (ISO/IEC 42010)*. Standard 42010/2007, International Organization for Standardization (ISO)/ International Electrotechnical Commission (IEC), 2007.

Josuttis, N. M. *SOA in practice: The art of distributed systems design*. O'Reilly, 2007.

Kazhamiakin, R.; Pistore, M.; Santuari, L. Analysis of communication models in web service compositions. In: *Proceedings of the 15th International Conference on World Wide Web (WWW 2006)*, New York, NY, USA, 2006, p. 267–276.

Kazman, R.; Bass, L.; Abowd, G.; Webb, M. SAAM: A method for analyzing the properties of software architectures. In: *Proceedings of the 16th International Conference on Software Engineering (ICSE 1994)*, Sorrento, Italy, 1994, p. 81–90.

Kazman, R.; Klein, M.; Barbacci, M.; Longstaff, T.; Lipson, H.; Carriere, J. The architecture tradeoff analysis method. In: *Proceedings of the 4th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 1998)*, Monterey, CA, USA, 1998, p. 68–78.

Kiczales, G.; Irwin, J.; Lamping, J.; Loingtier, J.-M.; Lopes, C.; Maeda, C.; Menhdhekar, A. Aspect-oriented programming. In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, Jyväskylä, Finland: Springer-Verlag, 1997, p. 220–242 (LNCS v.1241).

Kitchenham, B. *Procedures for performing systematic reviews.* Tech. Report TR/SE-0401, Software Engineering Group, Department of Computer Science Keele University, United King and Empirical Software Engineering, National ICT Australia Ltd, Austrália, 2004.

Kruchten, P. The 4+1 view model of architecture. *IEEE Software*, v. 12, n. 6, p. 42–50, 1995.

Kruchten, P.; Obbink, H.; Stafford, J. The past, present, and future for software architecture. *IEEE Software*, v. 23, n. 2, p. 22–30, 2006.

Laddad, R. Aspect-oriented programming will improve quality. *IEEE Software*, v. 20, n. 6, p. 90–91, 2003.

Lan, J.; Liu, Y.; Chai, Y. A solution model for service-oriented architecture. In: *Proceedings of the 7th World Congress on Intelligent Control and Automation (WCICA 2008)*, Chongqing, China, 2008, p. 4184–4189.

Land, R. *A brief survey of software architecture.* Relatório Técnico, Department of Computer Engineering, Mälardalen University, Sweden, 2002.

Leitão Jr., P. S. *Suporte ao teste estrutural de programas Cobol no ambiente Poke-Tool.* Dissertação de Mestrado, DCA/FEEC/UNICAMP, Campinas, SP, 1992.

Leppaniemi, J.; Linna, P.; Soini, J.; Jaakkola, H. Toward a flexible service-oriented reference architecture for situational awareness systems in distributed disaster knowledge management. In: *Proceedings of the International Conference on Management of Engineering & Technology (PICMET 2009)*, Portland, Oregon, USA, 2009, p. 959–965.

Leymann, F.; Roller, D.; Schmidt., M.-T. Web services and business process management. *IBM Systems Journal*, v. 41, n. 2, p. 198–211, 2002.

Li, N.; Praphamontripong, U.; Offut, J. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW 2009)*, Denver, USA, 2009, p. 220–229.

Ma, Y.-S.; Offutt, J.; Kwon, Y.-R. MuJava: a mutation system for java. In: *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, New York, NY, USA: ACM, 2006, p. 827–830.

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

Medvidovic, N.; Rosenblum, D. S.; Redmiles, D. F.; Robbins, J. E. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology*, v. 11, n. 1, p. 2–57, 2002.

Medvidovic, N.; Taylor, R. N. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, v. 26, p. 70–93, 2000.

Mendes, A. *Arquitetura de Software: Desenvolvimento orientado para arquitetura.* Campus, 2002.

Merson, P. Como documentar arquiteturas de software. Software Engineering Institute (SEI), Carnegie Mellon University, material de Apoio, mini-curso do 19^{o} Simpósio Brasileiro de Engenharia de Software (SBES 2005), 2005.

Microsoft Application architecture for .NET: Designing applications and services. Microsoft, 2002.

Morell, L. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, v. 16, n. 8, p. 844–857, 1990.

Muller, G. *A reference architecture primer.* Relatório Técnico, Eindhoven Univ. of Techn, Eindhoven, whitepaper, 2008.

Murakami, E.; Saraiva, A. M.; Ribeiro, Junior, L. C. M.; Cugnasca, C. E.; Hirakawa, A. R.; Correa, P. L. P. An infrastructure for the development of distributed service-oriented information systems for precision agriculture. *Computers and Electronics in Agriculture*, v. 58, n. 1, p. 37–48, 2007.

Myers, G. J.; Sandler, C.; Badgett, T.; Thomas, T. M. *The art of software testing.* New Jersey: John Wiley & Sons, Inc., 2004.

Nakagawa, E. Y. *Uma contribuição ao projeto arquitetural de ambientes de engenharia de software*. Tese de doutorado, Instituto de Ciências Matemáticas e de Computação – ICMC/USP, São Carlos, SP, 2006.

Nakagawa, E. Y.; Maldonado, J. C. Representing aspect-based architecture of software engineering environments. In: *Proceedings of the 1st Workshop on Aspects in Architectural Description, 6th International Conference on AOSD*, Vancouver, British Columbia, Canada, 2007, p. 1–4.

Nakagawa, E. Y.; Maldonado, J. C. Architectural requirements as basis to quality of software engineering environments. *Journal IEEE Latin America*, v. 6, n. 3, p. 1–20, 2008.

Nakagawa, E. Y.; Martins, R. M.; Felizardo, K.; Maldonado, J. C. Towards a process to design aspect-oriented reference architectures. In: *Proceedings of the 35th Latin American Informatics Conference*, Pelotas, RS, 2009a, p. 1–10.

Nakagawa, E. Y.; Oliveira, L. B. R. Using systematic review to elicit requirements of reference architectures. In: *Proceedings of the 14th Worskshop on Requirements Engineering (WER 2011), 14th Ibero-American Conference on Software Engineering (CIbSE 2011)*, Rio de Janeiro, RJ, 2011, p. 1–14.

Nakagawa, E. Y.; Sasaki, M. M. F.; Maldonado, J. C. An aspect-oriented framework for software documentation: An example on testing. In: *Proceedings of the 12th Iberoamerican Conference on Requirements Engineering and Software Environments (IDEAS 2009)*, Medellín, Colombia, 2009b, p. 225–238.

Nakagawa, E. Y.; Simão, A. S.; Ferrari, F.; Maldonado, J. C. Towards a reference architecture for software testing tools. In: *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE 2007)*, Boston, USA, 2007, p. 1–6.

OASIS *Reference model for service oriented architecture 1.0*. Relatório Técnico, Advanced Open Standards for the Information Society (OASIS), 2006.

OASIS *Reference architecture for service oriented architecture version 1.0*. Relatório Técnico, Advanced Open Standards for the Information Society (OASIS), 2008.

OASIS – Advanced Open Standards for the Information Society (OASIS), Universal, Discovery, Description and Integration (UDDI). Online, `http://www.oasis-open.org/specs/#uddiv2` - Acessado em 03/12/2010, 2010a.

OASIS – Advanced Open Standards for the Information Society (OASIS), Web Services Business Process Execution Language Version 2.0 (WS-BPEL). Online, `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html` - Acessado em 03/12/2010, 2010b.

Oliveira, L. B. R.; Felizardo, K. R.; Feitosa, D.; Nakagawa, E. Y. Reference models and reference architectures based on service-oriented architecture: A systematic review. In: *Proceedings of the 4th European Conference on Software Architecture (ECSA 2010)*, Copenhague, Denmark: Springer-Heidelberg, 2010, p. 360–367 (LNCS v.6285).

Oliveira, L. B. R.; Nakagawa, E. Y. Um levantamento de métodos de avaliação de arquiteturas de software específicas. In: *Anais do 3º Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS 2009)*, Natal, RN, 2009, p. 52–65.

Oliveira, L. B. R.; Nakagawa, E. Y. *A systematic review on service-oriented reference models and service-oriented reference architecture.* Relatório Técnico TR-353, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo - ICMC/USP, São Carlos, SP, 2010.

Oliveira, L. B. R.; Oliveira, R. A. P.; Cafeo, B. B. P.; Durelli, V. H. S. PascalMutants. Online, `http://ccsl.icmc.usp.br/drupal5/pt-br/content/pascal-mutants` - Acessado em 03/12/2010, 2009.

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.

Papazoglou, M. P.; Traverso, P.; Dustdar, S.; Leymann, F. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, v. 40, p. 38–45, 2007.

Papazoglou, M. P.; Traverso, P.; Dustdar, S.; Leymann, F. Service-oriented computing: a research roadmap. *International Journal of Cooperative Information Systems*, v. 17, n. 2, p. 223–255, 2008.

Peltz, C. Web Services Orchestration and Choreography. *IEEE Computer*, v. 36, n. 10, p. 46–52, 2003.

Peristeras, V.; Fradinho, M.; Lee, D.; Prinz, W.; Ruland, R.; Iqbal, K.; Decker, S. CERA: A collaborative environment reference architecture for interoperable CWE systems. *Service Oriented Computing and Applications*, v. 3, n. 1, p. 3–23, 2009.

Peterson, J. L. Petri nets. *ACM Computer Surveys*, v. 9, n. 3, p. 223–252, 1977.

Pressman, R. S. *Software engineering: A practitioner's approach.* 5th ed. McGraw-Hill, 2005.

Ramanathan, S.; Alexander, M.; Kerr, G. The IBM telecommunications service delivery platform. *IBM Systems Journal*, v. 47, n. 3, p. 433–443, 2008.

Rapps, S.; Weyuker, E. J. Data flow analysis techniques for test data selection. In: *Proceedings of the 6th International Conference on Software Engineering (ICSE 1982)*, Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, p. 272–278.

Reiff-Marganiec, S.; Truong, H.-L.; Casella, G.; Dorn, C.; Dustdar, S.; Moretzky, S. The incontext pervasive collaboration services architecture. In: *Proceedings of the 1st European Conference on Towards a Service-Based Internet*, Madrid, Spain: Springer-Verlag, 2008, p. 134–146 (LNCS v. 5377).

Richardson, D. J. TAOS: Testing with analysis and oracle support. In: *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA 1994)*, New York, NY, USA: ACM, 1994, p. 138–153.

Schmidt, M.-T.; Hutchison, B.; Lambros, P.; Phippen, R. The enterprise service bus: making service-oriented architecture real. *IBM Systems Journal*, v. 44, n. 4, p. 781–797, 2005.

SEI –Software Engineering Institute, Community Software Architecture Definitions. Online, `http://www.sei.cmu.edu/architecture/start/community.cfm` - Acessado em 03/12/2010, 2010.

SEI *CBAM: Cost benefit analysis method*. Relatório Técnico, Software Engineering Institute (SEI), `http://www.sei.cmu.edu/architecture/tools/cbam/index.cfm` – Acessado em 03/12/2010, 2010.

Shaw, M.; Clements, P. A field guide to boxology: Preliminary classification of architectural styles for software systems. In: *Proceedings of the 21st Computer Software and Applications Conference (COMPSAC)*, Washington, DC, USA, 1997, p. 6–13.

Shaw, M.; Clements, P. The golden age of software architecture. *IEEE Software*, v. 23, n. 2, p. 31–39, 2006.

Shaw, M.; Garlan, D. *Software architecture in practice*. Prentice-Hall, 1996.

Simão, A. S.; Maldonado, J. C.; Fabbri, S. C. P. F. Proteum-RS/PN - A tool to support edition, simulation and validation of Petri Nets based on mutation testing. In: *Anais do 14º Simpósio Brasileiro de Engenharia de Software (SBES 2000)*, João Pessoa, PB, 2000, p. 227–242.

Slyngstad, O. P.; Li, J.; Conradi, R.; Babar, M. A. Identifying and understanding architectural risks in software evolution: An empirical study. In: *Proceedings of the 9^{th} International Conference on Product-Focused Software Process Improvement (PROFES 2008)*, Berlin, Heidelberg: Springer-Verlag, 2008, p. 400–414 (LNCS v. 5089).

Sommerville, I. *Software engineering.* Addison Wesley, 2003.

Sugeta, T.; Maldonado, J. C.; Masiero, P. C.; Fabbri, S. C. P. F. Proteum-RS/ST - A tool to support statecharts validation based on mutation testing. In: *Anais do 4^o Workshop Iberoamericano de Engenharia de Requisitos e Ambientes de Software (IDEAS 2001)*, Santo Domingo, Costa Rica, 2001, p. 370–384.

Technologies, T. xSud Toolsuite. Online, `http://xsuds.argreenhouse.com` - Acessado em 07/01/2010, 1997.

Vincenzi, A. M. R. *Orientação a objeto: definição, implementação e análise de recursos de teste e validação.* Tese de Doutoramento, ICMC/USP, São Carlos, SP, 2004.

Vincenzi, A. M. R.; Delamaro, M. E.; Simão, A. S.; Maldonado, J. C. Muta-pro: Towards the definition of a mutation test process. *Journal of the Brazilian Computer Society*, v. 12, n. 2, p. 1–13, 2006.

W3C – World Wide Web Consortium, Extensible Markup Language (XML). Online, `http://www.w3.org/XML/` - Acessado em 03/12/2010, 2010a.

W3C – World Wide Web Consortium, Simple Object Access Protocol (SOAP). Online, `http://www.w3.org/TR/soap/` - Acessado em 03/12/2010, 2010b.

W3C – World Wide Web Consortium, Web Service Description Language (WSDL). Online, `http://www.w3.org/TR/wsdl` - Acessado em 03/12/2010, 2010c.

W3C – World Wide Web Consortium, Web Services Choreography Description Language Version 1.0 (WS-CDL). Online, `http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/` - Acessado em 03/12/2010, 2010d.

Wasserman, A. Toward a discipline of software engineering. *IEEE Software*, v. 13, n. 6, p. 23–31, 1996.

Weerawarana, S.; Curbera, F.; Leymann, F.; Storey, T.; Ferguson, D. F. *Web services platform architecture: Soap, wsdl, ws-policy, ws-addressing, ws-bpel, ws-reliable messaging, and more.* Upper Saddle River, New Jersey, USA: Prentice Hall., 2005.

Yang, J.-T.; Huang, J.-L.; Wang, F.-J.; Chu, W. Constructing an object-oriented architecture for web application testing. *Journal of Information Science and Engineering*, v. 18, n. 1, p. 59–84, 2002.

Yang, J.-T.; Huang, J.-L.; Wang, F.-J.; Chu, W. C. An object-oriented architecture supporting web application testing. *Proceedings of the 23rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 1999)*, p. 122–127, 1999.

Yano, T. *Estudo do teste de mutação em programas funcionais SML.* Dissertação de Mestrado, ICMC/USP, São Carlos, SP, 2004.

Yap, N.; Chiong, H. C.; Grundy, J.; Berrigan, R. Supporting dynamic software tool integration via web service-based components. In: *Proceedings of the Australian Conference on Software Engineering (ASWEC 2005)*, Washington, DC, USA: IEEE Computer Society, 2005, p. 160–169.

Zheng, Q. H.; Dong, B.; Tian, F.; Chen, W. A service-oriented approach to integration of e-learning information and resource management systems. In: *Proceedings of the 12th International Conference on Computer Supported Cooperative Work in Design*, Xian, China, 2008, p. 1047–1052.

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.

Zimmermann, O.; Kopp, P.; Pappe, S. Architectural knowledge in an SOA infrastructure reference architecture. In: *Software Architecture Knowledge Management*, Berlin, Heidenberg: Springer-Verlag, p. 217–241, 2009.