UNIVERSIDADE ESTADUAL DE MARINGÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

JOSE REINALDO MERLIN

Uma abordagem para criação de casos de teste funcionais para sistemas de conhecimento

Maringá
2011

JOSE REINALDO MERLIN

Uma abordagem para criação de casos de teste funcionais para sistemas de conhecimento

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Orientadora: Profa. Dra. Maria Madalena Dias

Maringá
2011

Dados Internacionais de Catalogação-na-Publicação (CIP)
(Biblioteca Central - UEM, Maringá – PR., Brasil)

M565U

Merlin, Jose Reinaldo
Uma abordagem para criação de casos de teste funcionais para sistemas de conhecimento / Jose Reinaldo Merlin. -- Maringá, 2011.
85 f. : il. col., figs., tabs.

Orientador: Profa. Dra. Maria Madalena Dias.
Dissertação (mestrado) - Universidade Estadual de Maringá, Centro de Tecnologia, Departamento de Informática, Programa de Pós-Graduação em Ciência da Computação, 2011.

1. Sistema de conhecimento. 2. Ontologia - Recuperação de informação. I. Dias, Maria Madalena, orient. II. Universidade Estadual de Maringá. Centro de Tecnologia. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. III. Título.

CDD 21.ed.006.332

MN-0000157

FOLHA DE APROVAÇÃO

JOSÉ REINALDO MERLIN

Uma abordagem para criação de casos de teste funcionais para sistemas de conhecimento

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática, Centro de Tecnologia da Universidade Estadual de Maringá, como requisito parcial para obtenção do título de Mestre em Ciência da Computação pela Banca Examinadora composta pelos membros:

BANCA EXAMINADORA

Profa. Dra. Maria Madalena Dias
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. Renato Balancieri
Universidade Estadual de Maringá – DIN/UEM

Profa. Dra. Marilde Terezinha Prado Santos
Universidade Federal de São Carlos – DC/UFSCar

Aprovada em: 23 de agosto de 2011.
Local da defesa: Sala 101, Bloco C56, *campus* da Universidade Estadual de Maringá

AGRADECIMENTO(S)

Agradeço primeiramente a Deus por me colocar no lugar certo na hora certa.

A minha orientadora, Maria Madalena Dias, pela atenção, paciência e dedicação.

A Maria Inês Davanço, pela simpatia e eficiência.

Aos companheiros de estrada, Claudia Candia, Carlos Eduardo Ribeiro, Natália e Ana Lúcia, por terem compartilhado os piores e melhores momentos.

Aos colegas do Centro de Ciências Tecnológicas da UENP, pelo incentivo.

A todos os demais que de alguma forma estiveram presentes.

A Fundação Araucária, pelo apoio financeiro.

Uma abordagem para criação de casos de teste funcionais para sistemas de conhecimento

RESUMO

O teste de software é uma importante atividade de garantia de qualidade. Para que os testes sejam bem conduzidos é necessário utilizar critérios de teste. Os critérios são abordagens sistemáticas para o projeto de testes. Cada critério utiliza procedimentos diferentes de seleção de dados de teste. Os critérios devem, também, ser adaptados para cada tipo de software sob teste. Assim, um software de sistema crítico exige um procedimento de teste diferente de um software para controle de comércio eletrônico. O tipo de software objeto deste trabalho são os programas que manipulam bases de conhecimento expressas em OWL (*Web Ontology Language*). Esse tipo de software necessita de uma abordagem diferenciada de teste, pois, ao contrário dos programas convencionais, nesses programas o conhecimento é separado do algoritmo que o manipula. Na maioria das vezes, esses dois elementos foram desenvolvidos por pessoas diferentes, pois a ontologia é definida por um engenheiro de ontologias ou especialista no domínio, enquanto que o programa é desenvolvido por um engenheiro de software. Problemas podem ocorrer quando, apesar de a ontologia estar correta, o programador tem um entendimento errado sobre o que deve ser feito. Nesta dissertação é apresentada uma análise das características desses programas e propõe–se uma abordagem para criação de testes funcionais baseada no critério Particionamento de Equivalência. O critério divide o domínio de entrada (ou saída) de um campo ou de uma função em conjuntos de dados válidos e inválidos. Durante a definição de casos de teste, dados representativos de diversos conjuntos são utilizados, a fim de evitar a redundância e impedir que dados relevantes sejam ignorados. A abordagem foi aplicada no teste de um protótipo de programa que realiza consultas em uma ontologia e os resultados mostraram que esse critério, considerado limitado para programas convencionais, é adequado ao teste de software alvo desta pesquisa, pois foi possível escrever casos de teste com base no critério.

Palavras-chave: Teste de Software. Ontologia. Particionamento de Equivalência.

An approach to create functional test cases for knowledge systems

ABSTRACT

Software testing is an important activity of quality assurance. To ensure that tests be well conducted is necessary to use testing criteria. The criteria are systematic approaches to the design of tests. Each criterion uses different procedures for selecting test data. The criteria also should be adapted for each type of software under test. Thus, a critical system software requires a different test procedure that a software for electronic commerce control. The object of this work are software programs that manipulate knowledge bases expressed in OWL (Web Ontology Language). This type of software requires a different approach because, unlike conventional programs, knowledge is separated from the algorithm that handles. Most often, these two elements were developed by different people, because the ontology is defined by an ontology engineer or specialist in the domain, while the program is developed by a software engineer. Problems can occur when, although the ontology is correct, the programmer has a misunderstanding about what should be done. This work presents an analysis of the characteristics of these programs and proposes an approach to creating functional tests based on Equivalence Partitioning Criterion. The criterion divides the domain of input (or output) of a field or a function on sets of valid and invalid data. During the definition of test cases, representative data from different sets are used in order to avoid redundancy and prevent that important data be left out. The approach was applied to test a prototype that queries an ontology and the results showed that this criterion, considered limited to conventional programs, is appropriate to software testing subject of this research, because it was possible to write test cases based the criterion.

Keywords: Software Testing. Ontology. Equivalence Partitioning.

LISTA DE FIGURAS

Figura 1.1 Papéis na construção de um Sistema de Conhecimento. 13
Figura 1.2 Delimitação da pesquisa. 14

Figura 2.1 Exemplo de Particionamento de Equivalência. 21
Figura 2.2 Representação do teste de unidade. Adaptado de Agarwal et al. (2010). 23
Figura 2.3 Exemplo de grafo RDF. Adaptado de (Noy e McGuinness, 2001) . 28
Figura 2.4 Exemplo de literal. 28
Figura 2.5 Trecho de ontologia sobre vinhos. 29
Figura 2.6 Teste baseado em aplicação. Adaptado de Porzel e Malaka (2004). 33
Figura 2.7 Estrutura do Modelo de Teste. Adaptado de El-Korany (2007). . 35

Figura 3.1 Criação de um objeto `OntModel` a partir de um arquivo owl. . . . 41
Figura 3.2 Arquitetura do protótipo 41
Figura 3.3 Doenças catalogadas na ontologia. 44
Figura 3.4 Doenças por grupo. 44

Figura 4.1 Exemplo de particionamento de equivalência. 48
Figura 4.2 Pseudo-algoritmo para teste de relacionamentos. 50
Figura 4.3 Exemplo de método de teste. 53
Figura 4.4 Diretório contendo os casos de teste e fontes. 53
Figura 4.5 Trecho de código do método `listarIndividuos()`. 55
Figura 4.6 Subclasses da classe `Agrotoxico`. 56
Figura 4.7 Trecho de código do método `testObterGrupos()`. 56
Figura 4.8 Caso de teste para a propriedade `hasEmpresaRegistrante`. . . . 59
Figura 4.9 Caso de teste para a propriedade `hasIngredienteAtivo`. 60
Figura 4.10 Caso de teste para a propriedade `Formula`. 63
Figura 4.11 Caso de teste para a propriedade `ClasseToxicologica`. 63
Figura 4.12 Relacionamento entre as classes `Sintomas`, `Intoxicacao` e `Agrotoxico`. 64
Figura 4.13 Caso de teste do relacionamento entre `Agrotoxico` e `Sintomas` . 65

Figura A.1 Classes da OntoTox. 77
Figura A.2 Diagrama gerado pelo *plugin* Jambalaya. 79

LISTA DE TABELAS

Tabela 3.1 Principais classes de Jena. Adaptado de Hebeler et al. (2009). . . 40
Tabela 3.2 Principais consultas do protótipo. 43

Tabela 4.1 Principais métodos *assert* de JUnit. 52
Tabela 4.2 Teste do método `obterIndividuos()` para classe `MetabolismoCelular`. 54
Tabela 4.3 Teste do método `obterGrupos()` da classe `Agrotoxico`. 57
Tabela 4.4 Propriedades dos indivíduos da classe `Agrotoxico`. 58
Tabela 4.5 Teste da propriedade `hasEmpresaRegistrante` da classe `Agrotoxico`. 59
Tabela 4.6 Teste do método `getIngredienteAtivo` da classe `Agrotoxico`. . . 60
Tabela 4.7 Propriedades de tipos de dados da classe `Agrotoxico`. 61
Tabela 4.8 Teste da propriedade `Formula` da classe `Agrotoxico`. 62
Tabela 4.9 Teste da propriedade `ClasseToxicologica` da classe `Agrotoxico`. 64
Tabela 4.10 Teste do relacionamento indireto entre `Agrotoxico` e `Sintomas`. . 65

LISTA DE ABREVIATURAS E SIGLAS

EC	Engenharia de Conhecimento
DL	*Description Logic*
ES	Engenharia de Software
OWL	*Ontology Web Language*
RDF	*Resource Description Framework*
RDFS	*RDF Schema*
SC	Sistemas de Conhecimento
UML	*Unified Modeling Language*
V e V	Verificação e Validação

SUMÁRIO

1 Introdução **11**

1.1 Considerações Iniciais 11
1.2 Contexto e Motivação 12
1.3 Objetivos 13
1.4 Delimitação do Trabalho 14
1.5 Metodologia da Pesquisa 14
1.6 Organização do Trabalho 15

2 Fundamentação Teórica **16**

2.1 Definições 17
2.2 Princípios de Teste de Software 17
2.3 Técnicas e Critérios de Teste 19
2.3.1 Teste Funcional 19
2.3.2 Teste Estrutural 21
2.3.3 Teste Baseado em Defeitos 22
2.4 Fases do Teste 22
2.4.1 Teste de Unidade 22
2.4.2 Teste de Integração 23
2.4.3 Teste de Sistema 24
2.5 Teste de Regressão 24
2.6 Engenharia do Conhecimento 25
2.7 Ontologias 25
2.8 Linguagens e Representações 27
2.8.1 *Resource Description Framework* 27
2.8.2 RDF Schema 28
2.8.3 Linguagem OWL 28
2.8.4 Descrição de classes 29
2.8.5 Propriedades em OWL 30
2.8.6 Restrições de Cardinalidade 31
2.9 Avaliação e Teste de Sistemas de Conhecimento 32
2.9.1 Avaliação e Teste de Ontologias 32
2.9.2 Processo de Teste de Sistemas 34
2.10 Considerações Finais 37

3 Protótipo Desenvolvido **38**
3.1 Protótipo de Aplicação . . . 38
3.1.1 Tecnologias Utilizadas . . . 38
3.1.2 Jena . . . 39
3.1.3 Arquitetura do Protótipo . . . 40
3.1.4 Tipos de Consultas . . . 42
3.1.5 Teste do Protótipo . . . 42
3.2 Teste Baseado em Questões de Competência . . . 45
3.3 Considerações Finais . . . 45

4 Abordagem de Teste Proposta **46**
4.1 Níveis de Teste . . . 46
4.2 Abordagem Proposta . . . 47
4.2.1 Particionamento de Equivalência . . . 47
4.2.2 Características dos Programas . . . 48
4.2.3 Passos do Teste . . . 50
4.3 Aplicação da Abordagem . . . 51
4.3.1 Ferramentas Utilizadas . . . 52
4.3.2 Testes Realizados . . . 53
4.3.3 Relacionamentos Indiretos . . . 63
4.4 Resumo da Abordagem . . . 66
4.5 Resultados e Discussão . . . 67
4.6 Considerações Finais . . . 68

5 Conclusões **69**
5.0.1 Contribuições . . . 70
5.1 Trabalhos Futuros Sugeridos . . . 71

Referências **72**

A Ontologia OntoTox **76**
A.1 A OntoTox . . . 76
A.2 Questões de Competência . . . 78

B Casos de teste aplicados **81**
B.1 Casos de teste para classe `Agrotoxico` . . . 81
B.2 Casos de teste para outras classes . . . 84

Capítulo

1

Introdução

1.1 Considerações Iniciais

Garantir a qualidade do software é uma preocupação da Engenharia de Software (ES) desde o seu surgimento. Esse tem sido um trabalho difícil, pois os sistemas tornaram–se cada vez mais complexos, heterogêneos e críticos. Apesar do esforço e dos métodos criados pela ES, a presença de defeitos nos produtos de software tem sido uma constante. O custo desses defeitos faz com que sejam desenvolvidos e aprimorados métodos de garantia de qualidade.

A atividade de teste tem sido uma das técnicas mais utilizadas para garantir software livre de defeitos. Ao longo dos anos surgiram técnicas e critérios de teste que procuram aumentar a eficácia e eficiência dessa tarefa. De maneira geral, as técnicas e critérios buscam encontrar o maior número de defeitos ao custo mais baixo possível.

Considerando que existem diferentes tipos de sistemas, é necessário analisar qual a abordagem, ou combinação de abordagens, é mais produtiva para cada tipo. Nesse sentido, este trabalho analisa os testes de sistemas de conhecimento (SC).

Sistemas de Conhecimento são sistemas de informação que fazem uso intensivo de conhecimento. Esses sistemas são empregados em áreas como processamento de linguagem natural, auxílio ao diagnóstico de doenças e detecção de fraudes em transações eletrônicas, entre outras. O objetivo de um SC é apoiar a criação, transferência e aplicação de conhecimento de um determinado domínio. A Engenharia do Conhecimento é a disciplina dedicada à produção de tais sistemas.

Os SCs não são centrados em software, e sim em conhecimento, mas utilizam programas para executar suas tarefas. Garantir a qualidade do software que apóia um SC é fundamental para se conseguir a qualidade do sistema como um todo, pois a qualidade de um sistema depende da qualidade de cada componente.

Os SCs fazem uso de uma base de conhecimento. Essa base é desenvolvida com o conhecimento obtido de um especialista no domínio e/ou de outras fontes, como literatura da área. Nos últimos anos ontologias têm sido utilizadas para representar conhecimento. Segundo Gruber (1995), uma ontologia é uma especificação formal e explícita de conhecimento de um domínio. Ontologias possibilitam estruturação, compartilhamento e reuso de informações a respeito de um campo de conhecimento.

Existem diversas abordagens de avaliação e teste que podem garantir a qualidade da ontologia construída. Por outro lado, a construção do programa que manipula a base de conhecimento é realizada por desenvolvedores de software. Esses profissionais, muitas vezes, criam programas utilizando linguagens convencionais, como Java ou C#. Torna–se necessário, por isso, garantir a correta integração entre o programa e a ontologia. As abordagens de teste aplicadas a software convencional podem não ser adequadas para SCs.

1.2 Contexto e Motivação

O desenvolvimento de um Sistema de Conhecimento é uma tarefa que envolve diversas pessoas executando papéis diferentes. Embora não exista um processo único para a construção de sistemas, alguns papéis são típicos. Um cenário simplificado é apresentado na Figura 1.1.

Nesse cenário, o desenvolvimento da ontologia é feito por um engenheiro de ontologia, com o apoio de um especialista no domínio e outras fontes de conhecimento, como literatura especializada na área. Depois de construída, a ontologia deve ser testada. As ferramentas de desenvolvimento, como Protégé (Protégé, 2011) por exemplo, contam com funcionalidades de teste, como teste de sanidade [1] e restrições. Um especialista no domínio pode auxiliar a avaliar a ontologia utilizando métodos de avaliação e validação, como os expostos na Seção 2.9.

Para que a ontologia possa ser utilizada fora do ambiente de uma ferramenta de construção de ontologias como Protégé, é necessário uma aplicação amigável ao usuário, que nem sempre é um especialista no domínio. Assim, o papel do desenvolvedor de

[1] Uma exemplo de teste de sanidade é verificar se uma propriedade corresponde à sua inversa.

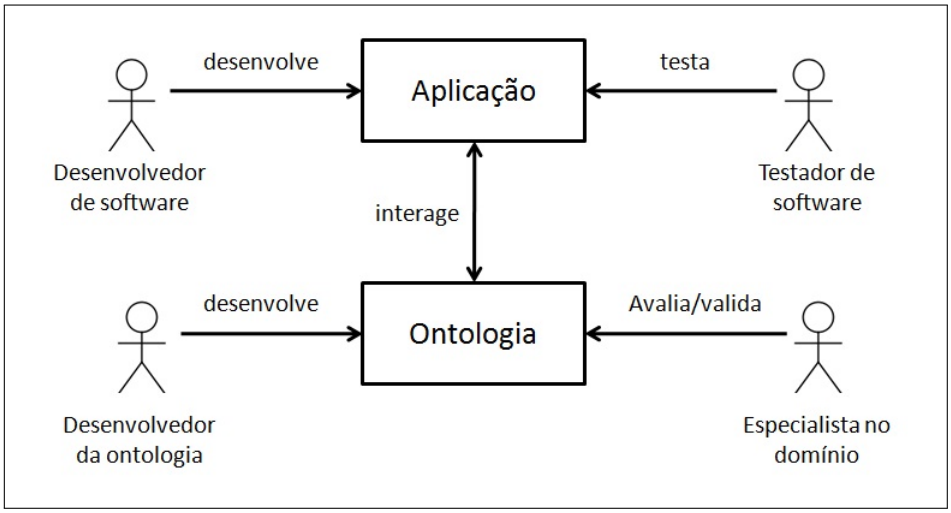

Figura 1.1: Papéis na construção de um Sistema de Conhecimento.

software é criar um programa que permita ao usuário realizar as consultas na ontologia sem que ele precise conhecer o ambiente Protégé ou similar. Problemas podem ocorrer pelo fato de o desenvolvedor não estar familiarizado com as representações da ontologia.

O testador é o responsável por garantir a qualidade do sistema integrado, encontrando possíveis defeitos que possam ter sido introduzidos durante o processo. Na fase de teste, é necessário levar em conta características específicas de uma aplicação que manipula ontologias.

Este trabalho é motivado pela necessidade de se definir uma abordagem de teste que contribua com a entrega de sistemas livres de defeitos e que, assim, possam ser úteis ao usuário final.

1.3 Objetivos

O objetivo deste trabalho é a elaboração de uma abordagem para criação de casos de teste para programas que manipulam bases de conhecimento expressas em ontologias.

Como objetivos específicos têm-se:

- Identificar características de abordagens de casos de teste de software a serem consideradas como adequadas para sistemas de conhecimento;
- Desenvolver um protótipo de aplicação para a realização de testes usando a abordagem proposta nesta pesquisa; e

- Realizar os testes necessários, utilizando o protótipo desenvolvido para demonstrar o uso da abordagem de casos de teste proposta.

1.4 Delimitação do Trabalho

A delimitação da pesquisa está representada na Figura 1.2.

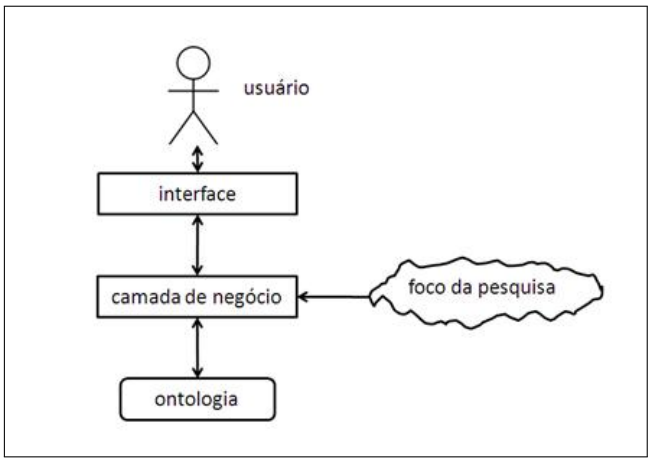

Figura 1.2: Delimitação da pesquisa.

Pressupõe–se que a ontologia esteja correta. Para avaliação/teste da ontologia, ferramentas e métodos estão disponíveis (Seção 2.9). O foco deste trabalho é testar a camada do programa que faz as consultas na ontologia e recebe as respostas. Já a camada de interface utiliza tecnologias convencionais que não envolve características específicas para SCs e não será aqui considerada.

1.5 Metodologia da Pesquisa

Segundo Gil (2008), pesquisa é o procedimento racional e sistemático para se encontrar a solução de um problema. As pesquisas podem ser classificadas em diferentes categorias. Do ponto de vista dos objetivos, esta é uma pesquisa exploratória, pois visa proporcionar maior familiaridade com o problema e aprimorar ideias. Do ponto de vista da abordagem do problema, esta pesquisa é qualitativa por se tratar de um estudo que não envolve quantificação de dados. A utilização da abordagem qualitativa justifica–se por se tratar de um estudo inicial que deve ser complementado com estudos experimentais em trabalhos futuros.

O trabalho possui um aspecto teórico-prático, pois ao mesmo tempo em que aumenta o conhecimento sobre o assunto, seus resultados podem ser utilizados em aplicações práticas.

Para o desenvolvimento deste trabalho, as seguintes etapas foram realizadas:

- Revisão bibliográfica: nesta etapa foram estudados conceitos relacionados às áreas envolvidas neste trabalho, que são teste de software, sistemas de conhecimento e ontologias;
- Construção de um protótipo de aplicação que manipula uma ontologia escrita na linguagem OWL;
- Elaboração de uma abordagem para o teste de integração entre o programa e a ontologia;
- Aplicação de testes segundo a abordagem proposta; e
- Análise da abordagem.

1.6 Organização do Trabalho

Este trabalho está organizado como segue. No Capítulo 2 é apresentada uma revisão de literatura sobre teste de software, mostrando as principais técnicas e critérios, bem como uma visão geral sobre sistemas de conhecimento e ontologias. No Capítulo 3 é mostrada a ontologia utilizada como base ao desenvolvimento da abordagem proposta e o protótipo de aplicação desenvolvido. A abordagem de teste, foco deste trabalho, é descrita no Capítulo 4 e as conclusões e trabalhos futuros são discutidos no Capítulo 5.

Capítulo

2

Fundamentação Teórica

O desenvolvimento de sistemas baseados em software é um processo complexo. Mesmo empregando-se técnicas, métodos e ferramentas de Engenharia, é possível que permaneçam erros no produto. As atividades de Verificação e Validação (V e V) são executadas com a finalidade de minimizar a ocorrência de erros e contribuir para que seja entregue um produto de qualidade. A atividade de teste é uma das principais tarefas de V e V e consiste, basicamente, em executar um programa com uma determinada entrada e comparar a saída obtida com a saída esperada. No entanto, para que se consiga eficiência e eficácia, os testes devem ser realizados com a utilização de técnicas e critérios.

A abordagem utilizada para teste de programas deve ser diferente. Por exemplo, a abordagem para um sistema de tempo real deve ser diferente de um sistema de controle de estoque. Assim, como os sistemas de conhecimento possuem características próprias que os diferenciam dos demais, torna-se necessário definir estratégias de teste específicas. Entre essas características estão a separação entre o conhecimento e o algoritmo que o manipula e o fato de o desenvolvedor da base de conhecimento nem sempre ser o desenvolvedor do programa que o utiliza. Outra importante característica, quando ontologias são utilizadas para representar o conhecimento, é a possibilidade de realizar inferências sobre o conhecimento explícito, sendo essa uma das principais diferenças entre ontologias e banco de dados. Por consequência, o teste de tais sistemas deve enfocar suas particularidades.

Neste capítulo, são apresentados inicialmente definições, conceitos, técnicas e critérios de teste de software. A seguir é apresentada uma revisão bibliográfica sobre temas ligados a sistemas de conhecimento, discutindo a disciplina Engenharia do Conhecimento,

ontologias e a linguagem OWL. Por último, são comentados alguns trabalhos relacionados à avaliação e ao teste de ontologias e sistemas de conhecimento.

2.1 Definições

Teste de software pode ser definido como a atividade de executar um programa em busca de defeitos (Myers, 2004). A fase de teste é essencial no processo de software, sendo largamente utilizada pela indústria para garantir qualidade. Bertolino (2007) define teste como a observação da execução do programa para analisar se ele se comporta como esperado e identificar potenciais falhas no produto.

Casos de teste são pares formados por um dado de teste (entrada) e pelo resultado esperado quando o programa é executado com aquele dado (saída) (Delamaro et al., 2007). A definição dos casos de teste é uma tarefa importante no processo, pois a eficácia do teste depende da qualidade dos casos utilizados. Segundo Agarwal et al. (2010), um bom caso de teste é aquele que tem alta probabilidade de encontrar um erro ainda não conhecido e um teste bem sucedido é aquele que descobre defeitos.

Durante os testes, o oráculo [1] analisa o resultado produzido pelo programa e compara com o resultado esperado (Agarwal et al., 2010). O papel de oráculo pode ser desempenhado por um ser humano, no entanto, isso aumentaria o tempo de análise. Por isso, muito esforço tem sido empregado no desenvolvimento de ferramentas automatizadas para teste de software.

2.2 Princípios de Teste de Software

Segundo Myers (2004), há dois importantes fundamentos no teste: o econômico e o psicológico. O econômico refere-se à necessidade de se encontrar a maior quantidade de defeitos com a menor quantidade de testes. O segundo fundamento está ligado à atitude do testador, que deve ser a de procurar encontrar erros, ao invés de provar que o programa está correto. Para atingir o objetivo dos testes, segundo estes fundamentos, o autor apresenta dez princípios que devem ser observados durante o teste de software:

1. Uma parte necessária do caso de teste é a definição da saída esperada ou resultado;

2. Um programador deveria evitar testar seus próprios programas;

[1] Oráculo é o ente (ser humano ou programa) que confere os resultados dos testes.

3. Uma organização não deveria testar seus próprios programas;
4. Os resultados de cada teste devem ser inspecionados minuciosamente;
5. Casos de teste devem ser escritos para as condições de entrada que são inválidas e inesperadas, bem como para aquelas que são válidas e esperadas;
6. O programa deve ser examinado para ver se ele faz o que deveria fazer, bem como se ele não faz o que não deveria fazer;
7. Os casos de teste não devem ser descartados, a menos que o programa seja descartável;
8. O esforço de teste não deve ser planejado supondo que nenhum erro será encontrado;
9. A probabilidade da existência de mais erros em uma parte de um programa é proporcional ao número de erros já encontrados naquela seção; e
10. O teste é uma atividade extremamente criativa e uma tarefa intelectualmente desafiadora.

Durante o projeto dos testes, nem sempre é fácil saber qual o resultado que o programa deve apresentar. Muitas vezes, dado um conjunto de entradas, a saída esperada não é obvia. A ausência de uma especificação completa para o software é uma das dificuldades a que se refere o princípio um.

O segundo princípio refere-se à tendência do programador em escolher para os testes os mesmos valores que ele tinha em mente quando desenvolveu o programa. Existe uma tendência natural na escolha de valores médios ou comportamentos normais. De acordo com este princípio, devem ser escolhidos dados de teste que apresentem maior probabilidade de evidenciar os defeitos. O terceiro princípio, de certa forma relacionado ao segundo, diz respeito à maior neutralidade que uma empresa independente possa ter em revelar defeitos em um produto.

Por definição, os resultados do teste apenas revelam a existência de defeito, ficando por conta da depuração descobrir a causa. No entanto, o exame dos resultados do teste pode indicar a origem do erro bem como revelar outros defeitos correlatos. Por isso o princípio quatro deve ser observado.

Sabe-se que a maioria dos erros acontece quando entradas inválidas são fornecidas ou ações inesperadas são executadas. Um usuário pode fornecer um caractere em um campo numérico, ou pode fechar o programa sem salvar os dados. O quinto e o sexto

princípios referem-se à necessidade de se escolher os dados ou condições que levem o programa a falhar.

Teste de software é uma atividade com alto custo. Considerando que o programa sofrerá manutenção e os testes deverão ser refeitos, o sétimo princípio prescreve que os casos de teste devem ser armazenados para reutilização posterior.

O oitavo princípio está relacionado ao problema dos prazos. Se não for reservado tempo suficiente para a atividade de teste, corre-se o risco de ter que entregar o software sabendo-se que alguns defeitos não foram removidos. Também o princípio nove está ligado a este fato. Muitas vezes, a descoberta de um defeito revela muitos outros no mesmo módulo, aumentando o tempo para o teste.

As organizações devem investir em profissionais especializados em testes, ao invés de relegar a atividade para principiantes. Muitas vezes é exigido mais conhecimento para testar do que para desenvolver (princípio dez).

Esses 10 princípios clássicos nortearam o desenvolvimento das técnicas e critérios existentes. As mais conhecidas são comentadas na próxima seção.

2.3 Técnicas e Critérios de Teste

Idealmente, um programa deveria ser testado com todas as entradas possíveis. No entanto, mesmo para pequenos sistemas, a quantidade de entradas e a combinação dessas entradas tornam impraticável tal procedimento (Delamaro et al., 2007). Faz-se necessário, assim, utilizar técnicas e critérios de teste.

Técnicas de teste são abordagens sistemáticas para a execução de testes. As duas principais técnicas de teste são a funcional e a estrutural (Bertolino, 2007). O que diferencia cada técnica é a fonte utilizada para definir os requisitos de teste (Delamaro et al., 2007). Além dessas duas, também existe a técnica de teste baseado em defeitos.

Critérios de teste são métodos de seleção de dados de teste. Cada técnica de teste possui diversos critérios de teste. Segundo Delamaro et al. (2007), cada critério procura explorar determinados tipos de defeito, selecionando os dados de teste que tenham maior probabilidade de revelar o defeito.

2.3.1 Teste Funcional

O teste funcional (ou teste caixa-preta) caracteriza-se por focar-se nos requisitos funcionais do software. Este tipo de teste exige apenas a observação das saídas que são produzidas por determinadas entradas, sem analisar o código que produziu a saída. Uma vez

que o teste funcional não leva em consideração detalhes internos das estruturas de controle do programa, a atenção volta-se ao domínio da informação (Pressman, 2006). O teste funcional exige que a seleção de dados de teste seja feita com base em propriedades importantes de elementos nos domínios de entrada e saída das variáveis do programa (Howden, 1980).

Segundo Agarwal et al. (2010), o teste funcional procura identificar os seguintes tipos de erros, entre outros:

- funções incorretas ou ausentes;
- erros no modelo de dados;
- erros no acesso a fonte de dados externa.

Os dois principais critérios de teste funcional são a)particionamento de equivalência e b)análise de valor limite.

a)Particionamento de Equivalência

O particionamento de equivalência, ou particionamento em classes de equivalência, consiste em dividir o domínio de entrada em classes de tal forma que, se um valor de uma classe revelar um erro, qualquer membro desta classe tem a mesma probabilidade de revelar o mesmo erro. Por outro lado, se um membro não revelar erro, qualquer membro tem a mesma probabilidade de não revelar este erro. Isso evita teste com valores redundantes, ao mesmo tempo que impede que dados relevantes sejam deixados de lado durante os testes.

Segundo Burnstein (2003), o particionamento de equivalência apresenta as seguintes vantagens:

- Elimina a necessidade de teste exaustivo;
- Guia o testador na seleção de um subconjunto de entradas de teste com alta probabilidade de detectar um defeito; e
- Permite ao testador cobrir um domínio extenso de entradas/saídas com um subconjunto menor selecionado a partir de uma classe de equivalência.

Um exemplo de particionamento é mostrado na Figura 2.1. Três classes de equivalência são criadas, com base na especificação da função que determina que devem ser aceitos valores inteiros entre zero e dez. Uma classe com os dados válidos contém os

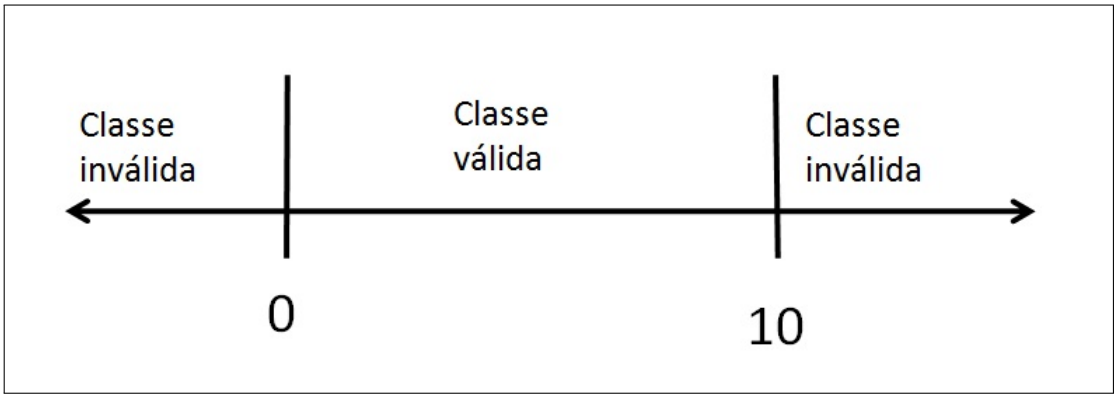

Figura 2.1: Exemplo de Particionamento de Equivalência.

valores entre zero e dez e duas com dados inválidos contém os valores abaixo de zero e acima de dez. Durante os testes, é selecionado um dado de cada classe.

b)Análise de Valor Limite

A análise de valor limite é um critério que estende o particionamento de equivalência. Surgiu da constatação que a maior parte dos defeitos dos programas apareciam quando os valores nas fronteiras das classes eram utilizados como entrada. Assim, esse critério prescreve que sejam selecionados os valores situados nos limites das classes de equivalência como dados de teste. No exemplo apresentado na Figura 2.1, os dados de teste seriam os valores -1, 0, 1, 9, 10 e 11.

2.3.2 Teste Estrutural

O teste estrutural (ou teste caixa-branca) é uma técnica de teste que seleciona os casos de teste por meio do exame do código. O testador precisa conhecer a estrutura do software e a implementação. É uma técnica aplicada, geralmente, a pequenas unidades do programa, principalmente durante os testes de unidade (Agarwal et al., 2010).

O teste estrutural especifica os requisitos de teste em termos da cobertura de um determinado conjunto de elementos da estrutura do programa (Zhu et al., 1997). Cobertura é o grau, medido em porcentagem, em que um item de cobertura específico foi exercitado por um conjunto de casos de teste. Uma cobertura de 80% de comandos significa que 80% dos comandos foram executados durante os testes. Muitas vezes, a análise de cobertura é utilizada para determinar quais os testes adicionais necessários para aumentar a cobertura.

Utilizando critérios de teste estrutural, deve-se criar casos de teste que (Agarwal et al., 2010):

- garantam que os caminhos independentes do módulo tenham sido exercitados pelos menos uma vez;
- exercitem todas as decisões lógicas em seus valores verdadeiro e falso; e
- executem os *loops* e seus limites.

Os critérios da técnica estrutural são divididos em dois grupos. Os critérios baseados em fluxo de controle focam as estruturas de controle do programa, tais como comandos e desvios. Os critérios baseados em fluxo de dados utilizam a análise de fluxo de dados para criar os casos de teste (Delamaro et al., 2007). Um importante critério desta técnica é o teste de caminho.

O teste de caminho é uma estratégia de teste estrutural cujo objetivo é exercitar cada caminho de execução independente (Sommerville, 2007). Um caminho é o conjunto de instruções que é executado para uma determinada entrada. Se cada caminho independente for executado, então todos os comandos do programa terão sido executados, bem como todas as decisões condicionais, em suas opções de verdadeiro e falso.

2.3.3 Teste Baseado em Defeitos

O teste baseado em defeitos utiliza defeitos típicos do processo de desenvolvimento de software para o estabelecimento dos requisitos de teste (Delamaro et al., 2007). Um exemplo de defeito típico explorado por esta técnica é a substituição de um operador "<=" por um operador "=".

2.4 Fases do Teste

Testes podem ser executados de maneira incremental durante os diferentes estágios do desenvolvimento do software. Os principais tipos de teste são: teste de unidade, teste de integração e teste de sistema.

2.4.1 Teste de Unidade

No teste de unidade, os componentes individuais (funções, procedimentos, métodos ou classes) são testados para assegurar que operam corretamente. Cada componente é testado independentemente dos outros componentes do sistema. O objetivo é encontrar defeitos relacionados a algoritmos ou estruturas de dados.

Executar testes de unidade é importante pelos seguintes motivos (Agarwal et al., 2010):

- o tamanho de um único módulo é pequeno o suficiente para se localizar um erro facilmente, tão logo o módulo tenha sido codificado;
- por ser pequeno, o módulo pode ter uma cobertura de teste praticamente total; e
- testar individualmente impede que os erros de um módulo se confundam, posteriormente, com erros de outros módulos.

Durante os testes de unidade é comum a utilização de *drivers* e *stubs*. Uma vez que o módulo é testado individualmente, não há um programa principal para "passar os dados". O *driver* de teste é uma unidade construída especialmente para fornecer dados de teste à unidade sob teste. Por outro lado, se o módulo sob teste chama outra unidade, um *stub* deve ser construído para simular esta unidade que está sendo chamada (Figura 2.2).

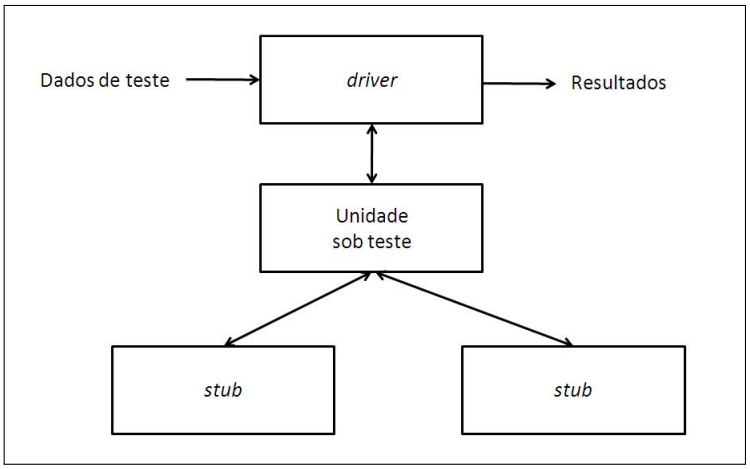

Figura 2.2: Representação do teste de unidade. Adaptado de Agarwal et al. (2010).

2.4.2 Teste de Integração

Testes de integração são testes que são efetuados à medida que a estrutura do programa é construída. Os módulos que já foram testados individualmente são combinados em subsistemas, que então são testados. O objetivo é descobrir erros nas interfaces entre os módulos (Delamaro et al., 2007).

Durante o teste de integração, uma abordagem incremental pode ser adotada. Isso é feito juntando apenas dois componentes e testando a integração entre eles. Se

houver erros, estes são removidos e os componentes são combinados com outro módulo e os testes são realizados. O processo é repetido até que todo o sistema tenha sido integrado (Agarwal et al., 2010).

2.4.3 Teste de Sistema

Conforme destaca Pressman (2006), software é apenas um dos elementos de um sistema baseado em computador. O software é incorporado a outros elementos do sistema (hardware, pessoas, etc) e são necessários testes para esta integração. Tais testes saem do escopo do processo de software e não são conduzidos apenas por engenheiros de software, necessitando da participação de administradores de rede, administradores de banco de dados, entre outros.

Teste de sistema é, na verdade, uma série de diferentes testes cujo principal propósito é exercitar completamente o sistema baseado em computador. Embora cada teste tenha diferentes propósitos, todos objetivam verificar que os elementos do sistema foram integrados apropriadamente e desempenham as funções a eles alocadas. Nessa fase também são verificados os requisitos não-funcionais, tais como desempenho, segurança e robustez (Pressman, 2006).

2.5 Teste de Regressão

Além das fases citadas na seção anterior, há também o "teste de regressão", que é o teste efetuado no software após cada manutenção. O objetivo é garantir que as modificações no programa não tenham introduzidos novos defeitos (Delamaro et al., 2007). Devido à constante necessidade do teste de regressão, é recomendável armazenar os casos de teste criados durante o desenvolvimento, para repeti-los após cada alteração no programa.

Segundo Oliveira (2007), regressão é o comportamento incorreto do software anteriormente correto. Podem ser de três tipos:

Regressão local: quando a mudança ou a correção de um defeito cria um novo defeito;

Regressão exposta: ocorre quando a mudança ou a correção de um defeito revela um defeito existente; e

Regressão remota: existe quando a mudança ou a correção de um defeito numa determinada área do sistema afeta outra área. Este tipo é normalmente o mais difícil de ser detectado.

2.6 Engenharia do Conhecimento

A Engenharia de Conhecimento (EC) é a disciplina que trata da construção de sistemas de conhecimento. É uma das disciplinas típicas da Era da Informação/Conhecimento, da mesma maneira que a engenharia mecânica teve sua importância durante a Revolução Industrial (Schreiber et al., 2000).

Tradicionalmente, a EC era vista como um processo de "extrair" ou "minerar" conhecimento da cabeça de um especialista e traduzir em um formato computacional. O conceito evoluiu para uma abordagem da EC como uma atividade de modelagem (Schreiber et al., 2000). Ao invés de se focar em questões como formalismos e mecanismos de inferência, o paradigma atual procura analisar o processo de construção e manutenção e o desenvolvimento de métodos, linguagens e ferramentas específicas para o desenvolvimento de SCs (Studer et al., 1998).

Alavi e Leidner (2001) definem sistemas de conhecimento como sistemas baseados em tecnologia da informação, desenvolvidos para apoiar processos de criação, armazenamento, recuperação, transferência e aplicação de conhecimento.

Conhecimento é um conceito muito discutido, mas de difícil definição. Alavi e Leidner (2001) afirmam que conhecimento é informação processada na mente dos indivíduos, é informação personalizada relativa aos fatos, procedimentos, conceitos, observações e julgamentos; é resultado de processo cognitivo. Para as autoras, os sistemas projetados para gerenciar conhecimento nas organizações não parecem tão diferentes de outros sistemas de informação, mas são orientados para permitir que os usuários associem significado à informação e extraiam conhecimento da informação ou dos dados.

Para Schreiber et al. (2000), existem dois aspectos distintos ligados ao conhecimento. O primeiro é o sentido de "propósito", uma vez que o conhecimento é o mecanismo intelectual usado para atingir algum objetivo. O segundo é a "capacidade geradora", pois uma das mais importantes funções do conhecimento é produzir novas informações.

Para poder ser utilizado em um SC, o conhecimento precisa ser representado de alguma forma. Atualmente é crescente a utilização de ontologias para representação de conhecimento.

2.7 Ontologias

O termo "ontologia", original da Filosofia, tornou-se popular na computação graças à promessa de resolver um dos grandes problemas enfrentados na utilização de computadores por humanos: a interoperabilidade. A interoperabilidade é necessária entre múltiplas

representações da realidade no computador e entre a representação computacional da realidade e a percepção dessa realidade pelos usuários humanos (Hepp, 2008). Studer et al. (1998) afirmam que as ontologias tornaram-se comuns no campo da Engenharia do Conhecimento, principalmente pelo fato de permitir o compartilhamento de conceitos tanto por humanos como por computadores.

Noy e McGuinness (2001) definem ontologia como a descrição formal e explícita dos conceitos de um domínio de conhecimento, propriedades de cada conceito e restrições. Linhalis (2007) explica que o conhecimento deve ser formal, para poder ser processado por computadores. Além disso, os conceitos são consensuais, no sentido de que são aceitos pela comunidade da área. Martimiano (2006) destaca que, por ser específica de um domínio, existe a necessidade de se ter um especialista do domínio para acompanhar a construção e validar os conceitos e relacionamentos que estão sendo modelados.

O papel das ontologias na engenharia do conhecimento é fornecer um vocabulário de termos e relacionamentos com os quais o modelo de domínio é construído (Studer et al., 1998). Ontologias permitem, desta forma, organizar o conhecimento.

Uma vez que uma ontologia define termos e relacionamentos, torna-se necessário apresentar alguns conceitos:

Classe: um mecanismo de abstração para agrupar recursos com características similares (W3C., 2004);

Indivíduo: cada "coisa" individualmente considerada. Um indivíduo pertencente a uma classe é chamado "instância" da classe, tal qual uma instância em UML [2] (Colomb, 2007);

Propriedade: uma relação binária entre classes. As propriedades de objeto associam indivíduos a indivíduos enquanto que as propriedades de dados associam indivíduos a valores de dados (W3C., 2004);

Noy e McGuinness (2001) utilizam uma ontologia para vinhos para explicar os conceitos. A classe `vinhos` representa todos os vinhos. Um vinho específico é uma instância da classe vinhos. Uma classe pode ter `subclasses`, que representam conceitos mais específicos, como por exemplo, vinho branco, vinho tinto, vinho rosé. `Propriedades` definem características, relacionamentos e restrições de classes e instâncias. O vinho `las perdices` tem o valor `corpo médio` para a propriedade `corpo`. Uma classe vinho pode ter a propriedade `fabricante` que é uma instância de `vinicola`. Todas as instâncias da

[2] *Unified Modeling Language.*

classe `vinicola` tem a propriedade `produz` que se refere a todos os vinhos que a vinícola produz.

Embora o termo "classe" seja similar ao utilizado em orientação a objetos, ontologias tratam o conceito de forma diferente. Na programação orientada a objetos a atenção se volta aos métodos nas classes (propriedades operacionais). O projetista de uma ontologia, por outro lado, baseia–se nas propriedades estruturais (Noy e McGuinness, 2001). Em outras palavras, enquanto a orientação a objetos se volta ao que a classe "faz", ontologias se preocupam em como a classe "organiza" os conceitos. Além disso, linguagens como a UML representam estruturas, enquanto que ontologias também mostram a população (Colomb, 2007). Uma ontologia junto com um conjunto de instâncias de classes constitui uma base de conhecimento (Noy e McGuinness, 2001).

2.8 Linguagens e Representações

Este trabalho considera apenas as ontologias formalizadas na linguagem OWL. Uma vez que OWL incorpora o modelo RDF, inicialmente este é comentado para, em seguida, introduzir a linguagem OWL.

2.8.1 Resource Description Framework

Resource Description Framework (RDF) é um modelo de dados especificado pela W3C. Esse modelo representa a informação como um conjunto de declarações chamadas *statements*. Um *statement* é composto de três partes: sujeito (*subject*), predicado (*predicate*) e objeto (*object*). *Statements* são também chamados de triplas, devido à essa estrutura.

Um *sujeito* de um *statement* é a coisa que o *statement* descreve; o *predicado* descreve a relação entre o *sujeito* e o *objeto* (Hebeler et al., 2009). Por exemplo, a declaração *ChiantiClassico hasMaker McGuinnesso*, extraída de uma ontologia de vinhos, significa que o vinho *ChiantiClassico* (sujeito) tem o fabricante (predicado) McGuinnesso (objeto).

As declarações, se representadas graficamente, formam um grafo orientado, com sujeitos e objetos de cada *statement* como nós e os predicados como arcos. Na Figura 2.3 é mostrado um exemplo simplificado de grafo RDF.

Os nós de um grafo RDF são os sujeitos e os objetos. Há dois tipos de nós: recursos (*resources*) e literais (*literals*). Literais representam valores concretos, como números e *strings* e só podem ser objetos. Recursos representam todos os demais

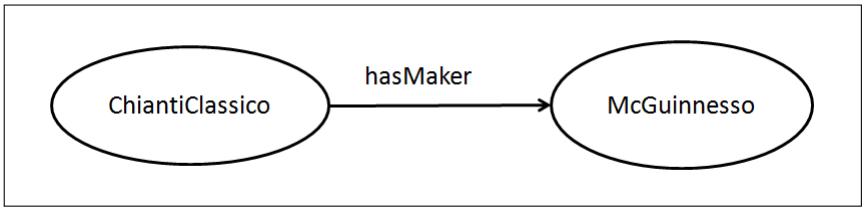

Figura 2.3: Exemplo de grafo RDF. Adaptado de (Noy e McGuinness, 2001)

elementos e podem ser sujeitos e objetos. No trecho de código reproduzido na Figura 2.4 é mostrado um exemplo de literal. O literal `nome` é do tipo *string* e possui o valor `Vinho branco`.

```
<Tipos rdf:ID=‘‘VinhoBranco’’>
   <Nome rdf:datatype=‘‘http://www.w3.org/2001/XMLSchema#string’’>
      Vinho branco
   </Nome>
</Tipos>
```

Figura 2.4: Exemplo de literal.

Arcos são os predicados, também chamados de propriedades, e representam conexões entre os recursos. No exemplo da Figura 2.3, a propriedade *hasMaker* conecta o recurso *ChiantiClassico* ao recurso *McGuinnesso*.

2.8.2 RDF Schema

Por meio de RDF é possível expressar informações sobre recursos, utilizando propriedades e valores. RDF fornece um modelo para capturar informação, porém não provê uma maneira de capturar o significado da informação. RDF *Schema* (RDFS) fornece uma linguagem com a qual o desenvolvedor pode criar seu próprio vocabulário (Hebeler et al., 2009).

RDFS descreve classes de recursos e propriedades usados no modelo RDF. Por meio de RDFS, os membros das classes são declarados, as classes e propriedades podem ser organizadas em hierarquias de generalização/especialização e o domínio e faixa de cada propriedade podem ser definidos.

2.8.3 Linguagem OWL

Existem várias linguagens para representação de ontologias, sendo que uma das mais utilizadas é a *Ontology Web Language* (OWL). OWL fornece mais opções de modelagem

sobre como os dados se relacionam, em relação às linguagens precursoras, tais como *Ontology Inference Language (OIL)* e *DARPA Agent Markup Language + OIL (DAML + OIL)* (Allemang e Hendler, 2008).

A linguagem OWL possui três sublinguagens: Lite, DL e Full, sendo a DL (*Description Logics*) a mais utilizada (Allemang e Hendler, 2008). *Lite* é uma versão mais limitada; permite a definição de hierarquias simples de classes e algumas restrições de propriedade, mas não permite definir disjunções entre classes, por exemplo. Já com OWL DL classes podem ser construídas por união ou intersecção bem como podem ser definidas disjunções. OWL Full, embora muito mais completa, não garante computabilidade das ontologias desenvolvidas (Martimiano, 2006).

A OWL fornece um meio de representar explicitamente um conjunto de conceitos e relacionamentos de um domínio de conhecimento, possibilitando que se façam inferências sobre essa representação. OWL possui um vocabulário que permite descrever classes, relacionamentos entre classes, restrições de cardinalidade, entre outras coisas (Martimiano, 2006). Na Figura 2.5 é mostrado um trecho de código em OWL. Na linha 1 é definida a classe `Vinho`, que é uma subclasse de `LiquidoPotavel` (linha 2). As linhas de 4 a 10 especificam que a propriedade `temFabricante` possui uma restrição de cardinalidade 1, indicando que um vinho é produzido por um único fabricante.

```
1   <owl:Class rdf:ID=‘‘Vinho’’>
2     <rdfs:subClassOf rdf:resource=‘‘LiquidoPotavel’’ />
3       <rdfs:subClassOf>
4         <owl:Restriction>
5           <owl:onProperty rdf:resource=‘‘#temFabricante’’ />
6             <owl:cardinality
7               rdf:datatype=‘‘&xsd;nonNegativeInteger’’>
8                  1
9               </owl:cardinality>
10        </owl:Restriction>
11    </rdfs:subClassOf>
```

Figura 2.5: Trecho de ontologia sobre vinhos.

2.8.4 Descrição de classes

OWL provê alguns métodos para descrição de classes. Por exemplo, o conjunto de membros de uma classe pode ser descrito em termos de outras classes utilizando o conjunto de operadores *union-of* e *intersection-of*. Também é possível definir que duas classes são disjuntas.

O operador *union–of* especifica que os elementos de uma classe pertencem a pelo menos uma das classes de uma lista. Por exemplo, a classe `WineDescriptor` pode ser definida como a união dos indivíduos que são `WineTaste` ou `WineColor`.

A propriedade *intersection-of* relaciona uma classe com um conjunto de outras classes. A declaração *intersection-of* descreve a classe que contém precisamente aqueles indivíduos que são membros de todas as classes descritas na lista (W3C., 2004). Por exemplo, os indivíduos da classe `WhiteBurgundy` devem ser definidos como os indivíduos pertencentes à classe `Burgundy` e `WhiteWine`.

2.8.5 Propriedades em OWL

Propriedades OWL são utilizadas para estabelecer relacionamentos entre recursos, além de restrições sobre esses relacionamentos. As duas classes principais de relacionamentos são (Hebeler et al., 2009):

Propriedade de objeto: (*ObjectProperty*) estabelecem relacionamentos entre indivíduos, por exemplo, `Loire localizadoEm ToursRegion`.

Propriedade de tipos de dados: (*DatatypeProperty*) estabelecem relacionamento entre um indivíduo e um dado literal, por exemplo, `LongridgeMerlot fabricadoEm 1970`.

OWL permite descrever as relações de domínio e intervalo entre propriedades e classes ou tipos de dados. Domínio (*domain*) especifica o tipo de indivíduo que é sujeito de triplas que utilizam a propriedade sendo descrita. Intervalo ou escopo (*range*) especifica o tipo de todos os indivíduos ou o tipo de todos os literais que são objeto das triplas que utilizam a propriedade sendo descrita. Por exemplo, em uma ontologia de doenças, a propriedade `temSintoma` liga indivíduos da classe `Doença` a indivíduos da classe `Sintoma`. Nesse caso, o domínio da propriedade `temSintoma` é `Doenca` e o escopo é `Sintoma`.

Além desses dois grupos de propriedades, há outros tipos de propriedades. As principais delas são descritas a seguir.

a) Propriedades Inversas

Propriedades declaram relacionamentos diretos, do domínio para o escopo ou do sujeito para o objeto. Muitas vezes, a existência de relacionamento em uma direção implica na existência de um relacionamento na direção inversa. Por exemplo, o relacionamento `temFabricante` ligando `vinho` a `vinícola` implica no relacionamento inverso `fabrica`,

entre `vinícola` e `vinho`.

b) Propriedades Funcionais e Funcionais Inversas

Uma propriedade funcional associa um único valor a um indivíduo em particular. Por exemplo, `dataDeNascimento` é uma propriedade funcional, pois um indivíduo possui apenas uma data de nascimento. Isso não impede que dois indivíduos compartilhem a mesma data.

Uma propriedade funcional inversa indica que nunca dois indivíduos terão o mesmo valor para um dado objeto, embora um indivíduo possa ter mais de um valor para o mesmo objeto. Em um *statement* com um predicado desse tipo, o objeto identifica, de forma única, o sujeito. Um exemplo desse tipo de propriedade é `temEmail`: um endereço de *email* só pode pertencer a uma única pessoa, embora uma pessoa possa ter mais de um endereço de *email*.

c) Propriedades Transitivas

Propriedades transitivas são geralmente usadas para estabelecer relacionamentos *parte de um todo* ou do tipo *contém*. Esse tipo de propriedade define que para qualquer indivíduo A, B e C e uma propriedade transitiva p, (A p B) e (B p C) implica (A p C).

d)Propriedades Simétricas e Assimétricas

Uma propriedade simétrica indica que ela é sua própria inversa. A existência de relacionamento em uma direção (sujeito para objeto) implica que o mesmo relacionamento existe na direção oposta (objeto para sujeito). Um exemplo desse tipo de relacionamento é `temConjuge`.

Por outro lado, uma propriedade assimétrica especifica que nunca haverá relacionamento bidirecional. Por exemplo, a propriedade `temPai` jamais poderá ser simétrica.

2.8.6 Restrições de Cardinalidade

As restrições de cardinalidade permitem especificar quantas vezes uma propriedade pode ser usada para descrever uma instância de classe. As três restrições de cardinalidade de OWL são:

- Cardinalidade mínima: deve haver ao menos N propriedades
- Cardinalidade máxima: pode haver no máximo N propriedades
- Cardinalidade: há exatamente N propriedades

2.9 Avaliação e Teste de Sistemas de Conhecimento

Sistemas de conhecimento, assim como qualquer sistema, precisam ser avaliados e testados. SCs tipicamente separam a base de conhecimento dos algoritmos que manipulam essa base. Diversas abordagens de avaliação e teste buscam validar a base de conhecimento. Na Subseção 2.9.1 são comentadas alguns trabalhos específicos sobre avaliação e teste de ontologias. Na Subseção 2.9.2 é apresentado o trabalho de El-Korany (2007), que aborda o teste integrado de SCs.

2.9.1 Avaliação e Teste de Ontologias

A avaliação (o quanto a ontologia é adequada) e o teste (o quanto a ontologia é correta) são, muitas vezes, tratados conjuntamente e sem distinção entre os termos. De maneira geral, as abordagens para avaliação e teste de ontologias podem ser classificadas em uma das seguintes categorias (Brank et al., 2005):

- baseadas em "padrão-ouro";
- baseadas no uso da ontologia em uma aplicação e avaliação dos resultados da aplicação;
- baseadas na comparação com alguma fonte de dados do domínio; e
- baseadas na avaliação por especialistas no domínio.

a) Padrão-ouro

A avaliação de uma ontologia com base em padrão-ouro é especialmente adequada quando ontologias são extraídas de forma automática ou semi-automática. Este método de avaliação confronta a ontologia gerada com outra já existente, que pode ter sido criada manualmente por um engenheiro de ontologia. O grau de similaridade entre a nova ontologia e o padrão-ouro é a medida da qualidade da ontologia (Maedche e Staab, 2002).

b) Teste baseado em aplicação da ontologia

Porzel e Malaka (2004) propõem uma abordagem para avaliar uma ontologia por meio da utilização em uma aplicação. Uma aplicação é um algoritmo que usa a ontologia para executar uma tarefa. A saída (resultados) da aplicação é comparada com um padrão-ouro pré-estabelecido. A abordagem é ilustrada na Figura 2.6.

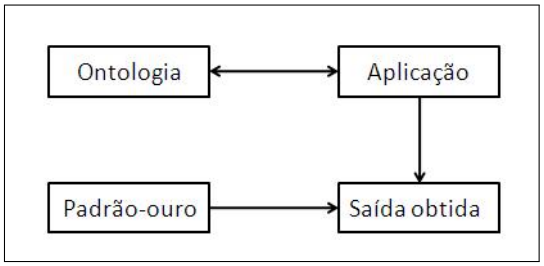

Figura 2.6: Teste baseado em aplicação. Adaptado de Porzel e Malaka (2004).

Apesar de utilizar uma aplicação durante a avaliação, o foco é a ontologia utilizada, e não a integração, sendo que a aplicação funciona como um *driver* de teste [3].

c) Comparação com fonte de dados do domínio

Este método de avaliação consiste em extrair um conjunto de termos relevantes do domínio em um grupo de documentos, utilizando análise semântica. A interseção entre o conjunto de termos extraído daqueles documentos e os termos presentes na ontologia oferecem a medida do quanto a ontologia é adequada (Brewster et al., 2004).

d) Avaliação por especialistas no domínio

Lozano-Tello e Gómez-Pérez (2004) apresentam um método de avaliação aplicável em situações de reuso de ontologias. Quando um novo projeto de software é desenvolvido, torna-se necessário avaliar qual ontologia é mais adequada dentre um conjunto de ontologias disponíveis, ou então decidir se uma determinada ontologia é adequada para aquele projeto. Os autores propõem um conjunto de passos a serem seguidos por um especialista durante a avaliação.

e) Outros tipos de teste em bases de conhecimento

Apesar da popularidade atingida pelas ontologias com o advento da web, existem outros tipos de bases de conhecimento, tais como regras de produção (Studer et al., 1998) e casos base (von Wangenheim e von Wangenheim, 2003). Preece (2001) apresenta um algoritmo para analisar a base de conhecimento representada por meio de regras de produção. Os principais erros que devem ser procurados referem-se a:

- regras inatingíveis : a pré-condição de uma regra não pode ser verdadeira para nenhuma entrada possível;

[3]Um *driver* é um módulo ou programa desenvolvido apenas para possibilitar o teste de uma unidade, no caso, de uma ontologia.

- regras não usadas: a pós-condição de uma regra não é usada para nenhuma inferência; e
- regras "inferiorizadas": existem regras mais gerais no conjunto de regras.

Santos et al. (2001) apresentam a ferramenta VERITAS, que permite realizar um conjunto de testes nas bases de conhecimento com vistas à detecção de anomalias. A detecção de anomalias consiste em pesquisar e isolar situações que possam produzir erros durante o funcionamento de um SC. As anomalias podem ser classificadas em:

- Circularidade: uma base de conhecimento contém circularidade se e somente se possui um conjunto de regras que permitem que aconteça uma situação de ciclo quando do disparo de uma regra;
- Ambivalência: uma base de conhecimento é ambivalente se e somente se, para um ambiente permissível, é possível inferir um conjunto de hipóteses não permissível;
- Redundância: a redundância acontece quando o conjunto de hipóteses finais passíveis de serem inferidas é o mesmo na presença ou na ausência de uma regra ou literal; e
- Deficiência: uma base de conhecimento é deficiente quando, e somente quando, existe um ambiente possível para o qual uma ou mais hipóteses poderiam ser inferidas e não o são.

2.9.2 Processo de Teste de Sistemas

Segundo El-Korany (2007), qualquer sistema, com o passar do tempo, torna-se menos útil se não sofre manutenção (evolução). Por isso, a Engenharia de Conhecimento, assim como a Engenharia de Software, adota o conceito de desenvolvimento evolucionário, que prega que o sistema nunca "fica pronto", mas evolui com o tempo, adaptando-se às mudanças. O processo evolucionário necessita de testes constantes, pois a cada etapa o sistema deve ser verificado e validado para garantir que continue apresentando o comportamento esperado. Neste contexto, teste e manutenção são componentes complementares.

El-Korany (2007) propõe um modelo de processo de teste evolucionário. Devido à característica fundamental dos SCs, que é a clara separação entre a base de conhecimento e o programa que a manipula, o modelo proposto é capaz de testar tanto o programa quanto o conhecimento correspondente. O autor decompõe o modelo de teste em dois componentes complementares: teste e manutenção, conforme mostrado na Figura 2.7.

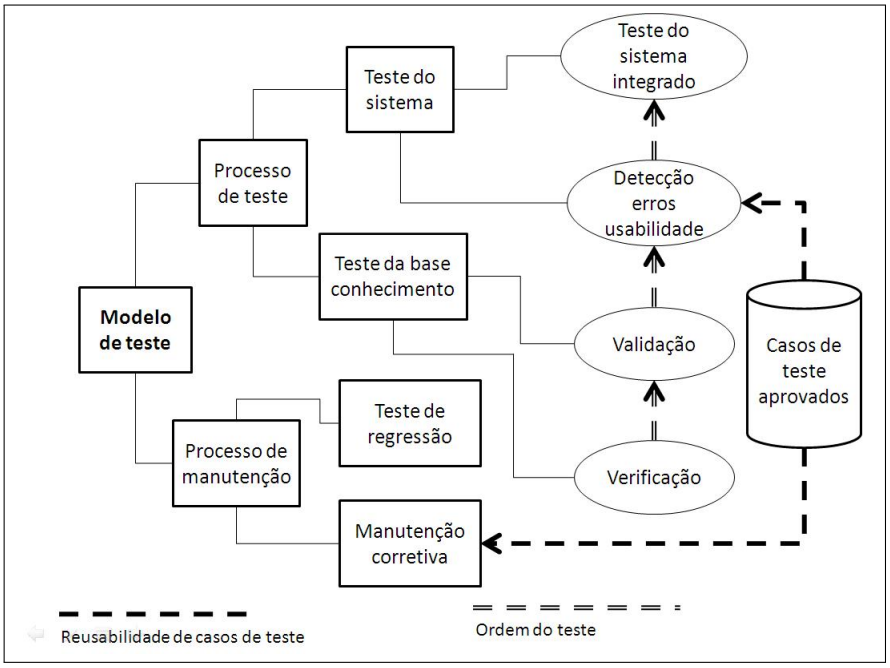

Figura 2.7: Estrutura do Modelo de Teste. Adaptado de El-Korany (2007).

Teste visa assegurar que o SC fornece o resultado correto quando é requisitado para resolver um problema. Para atingir este objetivo, o autor divide o processo de teste em duas fases:

- Teste da base de conhecimento
- Teste de sistema

A manutenção ocupa um importante papel em manter o sistema com a mesma qualidade original. A fase de manutenção pode ser dividida em duas etapas: manutenção corretiva e teste de regressão. A manutenção corretiva é essencial quando erros aparecem na base de conhecimento. O teste de regressão é aplicado após cada manutenção na base de conhecimento.

O modelo é composto por sucessivas fases, cujas características são: a) abordagem hierárquica e b) teste incremental.

a) Abordagem hierárquico:

A ordem de aplicação das fases de teste, mostrada na Figura 2.7, possui as seguintes vantagens:

- Uma vez que a verificação é aplicada antes da validação, erros de sintaxe e semântica são eliminados prematuramente; e

- Teste da base de conhecimento é aplicado antes do teste de sistema, o que permite corrigir possíveis falhas antes de completar o produto.

b) Teste incremental:

As atividades propostas pelo modelo são realizadas de maneira *bottom-up*, o que permite reusar os casos de teste em fases sucessivas da seguinte maneira:

- As técnicas de geração de casos de teste usadas na atividade de validação dependem principalmente do reuso dos casos de teste do domínio de conhecimento;
- Casos de teste usados durante o teste de sistema são baseados em casos de teste aceitos no processo de validação; e
- Casos de teste de regressão seletivos são um subconjunto do total de casos de teste usados para armazenar os casos de teste aprovados.

O processo proposto por El-Korany é dividido em duas fases: teste da base de conhecimento e teste do sistema.

A primeira fase concentra-se no teste da base de conhecimento. O objetivo é descobrir se o sistema desenvolvido está livre de erros internos, tais como inconsistência e incompleteza bem como assegurar que o conhecimento está de acordo com a especificação. Esta fase é composta de duas subfases: teste estático e teste dinâmico. O teste estático, também conhecido por verificação, não envolve a execução do sistema e procura erros na base de conhecimento introduzidos durante os estágios de projeto e implementação. O teste dinâmico (ou validação) requer a execução da base de conhecimento com um conjunto bem definido de casos de teste para avaliar os aspectos estrutural, funcional e computacional.

O objetivo da fase de teste de sistema é assegurar que o sistema executa apropriadamente. Diferentes testes que servem para examinar o desempenho em diferentes situações são aplicados. Esta fase é composta de duas importantes atividades: detecção de erros de usabilidade e teste de sistema integrado. O teste de usabilidade envolve questões como avaliar se o sistema é de fácil interação com usuário, com ações claras, páginas consistentes, validação de campos, etc. A robustez também deve ser avaliada. O teste de sistema integrado avalia se a instalação e atualização são documentadas, se no ambiente real do usuário o sistema se comporta como esperado, o que acontece se o usuário fechar o programa e abrir novamente, entre outros aspectos.

2.10 Considerações Finais

As atividades e técnicas de verificação e validação desempenham um importante papel no processo de Engenharia do Conhecimento, pois contribuem com a construção de um sistema confiável. O teste de software é uma das principais atividades de V e V. Em relação à avaliação de ontologias, existem diversas abordagens. Quanto ao teste de sistemas que fazem uso de ontologias, destaca-se o trabalho de El-Korany (2007). No entanto, este autor propõe um processo de teste, sem sugerir como criar casos de teste. No próximo capítulo é descrito o protótipo construído com o objetivo de ser usado para aplicação dos casos de teste definidos de acordo com a abordagem proposta e apresentada no Capítulo 4.

Capítulo

3

Protótipo Desenvolvido

Para tornar possível a definição de uma abordagem de teste, foi desenvolvido um protótipo que facilita a busca e visualização do conhecimento representado pela ontologia OntoTox. Neste capítulo é descrito o protótipo desenvolvido, destacando os temas mais relevantes aos testes objetos do próximo capítulo e no Apêndice A é descrita a OntoTox.

3.1 Protótipo de Aplicação

Para visualizar a ontologia sendo utilizada fora do ambiente Protégé, um protótipo de aplicação foi desenvolvido. A seguir as tecnologias usadas e o protótipo são descritos.

3.1.1 Tecnologias Utilizadas

A construção do protótipo foi realizada na plataforma J2EE (*Java 2 Enterprise Edition*), mais especificamente *Java Server Faces* (JSF) [1]. Como *container* e *web server* foi usado o servidor Tomcat [2]. Para manipulação da ontologia foi utilizada a biblioteca JENA [3] e para a elaboração da interface gráfica, a biblioteca IceFaces [4]. Optou–se pela plataforma web pela facilidade de instalação e para que o protótipo ficasse mais próximo de uma

[1] www.oracle.com
[2] tomcat.apache.org/
[3] jena.sourceforge.net/
[4] www.icefaces.org

aplicação real. A seguir são descritas estas tecnologias.

a) Java Server Faces

Java Server Faces é uma tecnologia Java para desenvolvimento de aplicações web do lado servidor (Geary e Horstmann, 2004), a qual é composta por três partes:

- Um conjunto de componentes de interface gráfica pré-fabricados;
- Um modelo de programação dirigido a eventos; e
- Um modelo de componente que permite a utilização de componentes adicionais desenvolvidos por terceiros.

Com JSF, o programador pode dedicar–se à lógica sem se preocupar com detalhes das requisições e respostas das chamadas ao servidor.

b)Tomcat

Tomcat é um *servlet container*, um ambiente no qual os *servlets* são executados. *Servlets* são classes Java que processam as requisições e devolvem as respostas ao usuário (Chetty, 2009). Além disso, o servidor também executa classes Java comuns.

c)ICEfaces

ICEfaces é uma biblioteca de componentes que permite o desenvolvimento de "aplicações ricas para *internet*" (*Rich Internet Applications - RIA*), como são chamadas as aplicações que executam em um navegador *web* (Eschen, 2009). Componentes ICEfaces são componentes visuais que fornecem suporte a AJAX de forma nativa. AJAX é um método de atualização de páginas que possibilita que uma parte da tela seja atualizada sem a necessidade de recarregar toda ela.

d)Jena

Jena é uma biblioteca *open source* desenvolvida pela HP para manipulação de ontologias. Diferentemente das tecnologias expostas anteriormente, Jena é específica para ontologias e, por isso, é descrita em mais detalhes a seguir.

3.1.2 Jena

As ontologias organizam conhecimento, no entanto, para que as informações armazenadas possam ser úteis, elas precisam ser processadas. Isso requer um programa que realize as

consultas na ontologia e apresente os resultados ao usuário. O predomínio de linguagens orientadas a objeto faz com que seja necessária a utilização de *frameworks* para manipular ontologias. Um dos mais conhecidos é Jena.

Jena é uma biblioteca de classes Java que provê recursos para a manipulação de ontologias expressas em OWL, além de outras representações, tais como *Resource Description Framework*(RDF) e *RDF Schema* (RDFS). Com a utilização das classes Jena, o programador pode trabalhar com os conceitos da ontologia na forma de objetos, com seus atributos e métodos (Hebeler et al., 2009). As principais classes da biblioteca são descritas na Tabela 3.1.

Tabela 3.1: Principais classes de Jena. Adaptado de Hebeler et al. (2009).

Classe	**Descrição**
`Resource`	Classe que representa um elemento de `statement`, como `subject`, `predicate` ou `object`.
`Statement`	Uma tripla contendo um `subject`, `predicate` e `object`. A classe `Statement` permite interrogações simples sobre seus componentes.
`Model`	`Model` é a representação da base de conhecimento. A classe derivada `OntModel` representa um modelo em OWL.
`Reasoner`	Classe que permite a utilização de mecanismos de inferência.

Jena permite carregar uma ontologia a partir de diversas origens, tais como um arquivo em disco, um endereço *web* ou um *stream* de dados. Na Figura 3.1 é mostrada parte do código para ler uma ontologia a partir de um arquivo. Na linha 5, o método `createOntologyModel()` retorna um objeto que representa a ontologia e pode ser manipulado na memória. O modelo é efetivamente "preenchido" quando executada a linha 7, que adiciona as triplas do arquivo em disco ao objeto.

3.1.3 Arquitetura do Protótipo

O protótipo de aplicação foi desenvolvido para ambiente *web* e pode ser dividido em três camadas principais, mostradas na Figura 3.2.

A camada de *apresentação* (ou camada de aplicação) contém as interfaces gráficas para a interação com o usuário. Basicamente, cada tela está relacionada a uma questão de competência. Ao contrário de aplicações convencionais, nas quais é possível definir claramente funcionalidades, na OntoTox é possível estabelecer o tipo de conhecimento que se quer obter, que é identificado por meio das questões de competência. Em outras

```
/*  1 */ private void carregarOntotox() {
/*  2 */    String inputFile = "owl/ontotox.owl"; //arquivo de origem
/*  3 */    OntModel model = null;
/*  4 */    try{
/*  5 */       model = ModelFactory.createOntologyModel();
/*  6 */       InputStream in = FileManager.get().open( inputFile );
/*  7 */       model.read(in,`` '');
/*  8 */    }
/*  9 */    catch (Exception e){
/* 10 */       System.out.println("Erro ao abrir arquivo:  "+ e );
/* 11 */    }
/* 12 */ }
```

Figura 3.1: Criação de um objeto `OntModel` a partir de um arquivo owl.

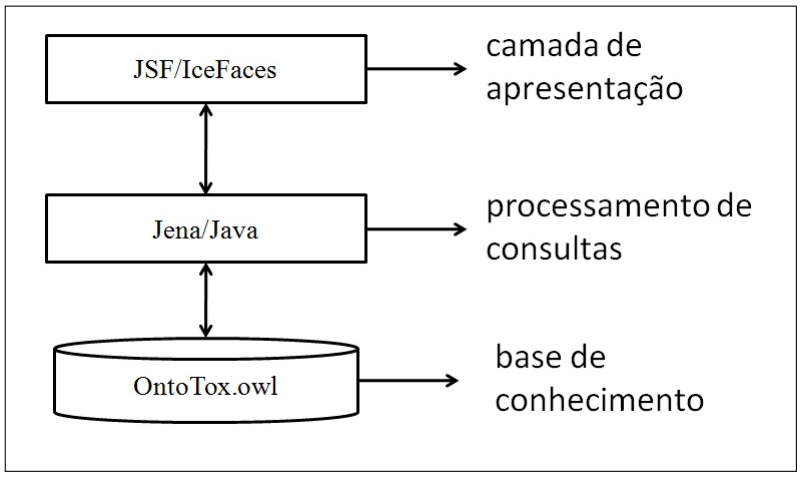

Figura 3.2: Arquitetura do protótipo

palavras, esta camada é responsável pela entrada de dados por parte do usuário e pela apresentação das respostas.

Na camada de *processamento de consultas* (ou camada de negócios), encontram–se classes Java que utilizam a biblioteca Jena para realizar consultas na ontologia. Cada classe OWL foi mapeada para uma classe Java. A idéia básica é ter uma classe Java representando uma classe da ontologia, com suas propriedades. Além disso, foram criadas classes adicionais, tais como a classe `Combo`, responsável por fornecer dados para montagens de componentes visuais para a camada de apresentação. Uma classe especial, `OntoTox`, carrega a ontologia a partir do arquivo .owl e fornece um objeto (instância de `OntModel`) para que as classes realizem as consultas neste modelo. Existem certas diferenças básicas entre classes Java e classes OWL. Por exemplo, para se obter o valor

de um atributo de objeto, invoca–se o método assessor (`getter`). Em OWL, deve–se interrogar o modelo (ontologia) em busca do recurso desejado.

Na *base de conhecimento* encontra–se a ontologia OntoTox, em um arquivo .owl. Este arquivo, uma vez lido do disco, passa a ser manipulado na memória.

A geração de classes Java a partir de classes OWL pode ser realizada de forma automática por ferramentas, como Protégé (Protégé, 2011). No entanto, optou–se por não utilizar os geradores automáticos de código, uma vez que as ferramentas para geração automática de código Java a partir de arquivos OWL apenas criam as assinaturas dos métodos. Por outro lado, como este é um estudo exploratório, apenas parte das funcionalidades foram implementadas. A criação manual de código permite escolher apenas os elementos que interessam para as consultas desejadas.

3.1.4 Tipos de Consultas

As consultas implementadas, baseadas nas questões de competência (descritas no Apêndice A), estão enumeradas na Tabela 3.2. Uma vez que são consultas simples, optou–se por descrevê–las na forma de tabela, ao invés de casos de uso.

As consultas, como mencionado anteriormente, estão relacionadas ao tipo de conhecimento que se quer obter. A consulta "Agrotóxicos por produto agrícola" aplica–se, por exemplo, nos casos em que um profissional de saúde não sabe qual o agrotóxico que pode ter intoxicado o paciente. Sabendo–se com qual cultura agrícola o intoxicado esteve trabalhando, pode–se deduzir os possíveis agrotóxicos que podem ter causado a intoxicação.

3.1.5 Teste do Protótipo

O protótipo foi inicialmente testado com base nas questões de competência. A questão "Quais são as doenças cadastradas relacionadas ao uso de agrotóxicos?", deve retornar a lista de todas as doenças presentes na ontologia. Na Figura 3.3 é mostrada a resposta da aplicação.

Para responder a consulta "Doenças por grupo" foi desenvolvida a tela apresentada na Figura 3.4. Nessa tela, quando o usuário seleciona um grupo no *combo* à esquerda, as doenças pertencentes ao grupo são mostradas à direita.

Tabela 3.2: Principais consultas do protótipo.

Consulta	**Descrição**
Doenças	O programa apresenta uma listagem com todas as doenças cadastradas.
Doenças por grupo	Usuário escolhe o grupo (ex. Metabolismo Celular) e o programa exibe todas as doenças do grupo
Sintomas por agrotóxico	Usuário informa o nome do agrotóxico e são listados os possíveis sintomas de intoxicação pelo agrotóxico.
Sintomas por doença	Usuário informa o nome da doença e são listados os sintomas.
Agrotóxicos por ingrediente ativo	Enumera todos os agrotóxicos que possuem o ingrediente ativo informado.
Fabricante do agrotóxico	Mostra a empresa que produz o agrotóxico informado.
Agrotóxicos por fabricante	Lista todos os agrotóxicos produzidos por uma empresa informada.
Pragas combatidas por agrotóxico	Lista as pragas que são combatidas por determinado agrotóxico.
Agrotóxicos por produto agrícola	Lista os agrotóxicos que geralmente são aplicados em determinado produto agrícola.
Pragas por produto agrícola	Mostra as pragas que comumente atacam o produto agrícola informado.
Agrotóxicos por sintoma	Dado um sintoma, o programa mostra os possíveis agrotóxicos que podem ser a causa.
Sintomas por grupo	Dados três sintomas, o programa mostra o provável grupo de agrotóxicos relacionado aos sintomas.

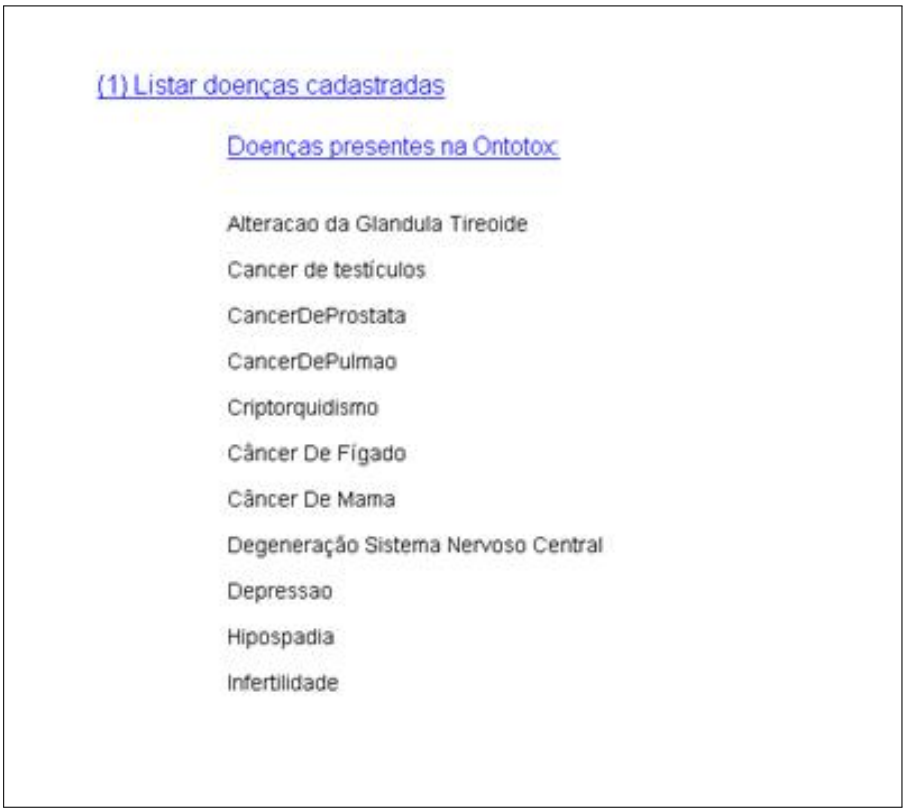

Figura 3.3: Doenças catalogadas na ontologia.

Figura 3.4: Doenças por grupo.

3.2 Teste Baseado em Questões de Competência

O teste realizado no protótipo consistiu em executar o programa e observar se as respostas apresentadas pela aplicação são as mesmas fornecidas pelo Protégé. Esse tipo de teste pode ser útil durante a fase de construção da aplicação, quando as funcionalidades estão sendo implementadas. No entanto, do ponto de vista do teste, algumas deficiências podem ser notadas, tais como execução manual e impossibilidade de repetição.

Apesar do teste manual permitir encontrar vários erros em uma aplicação, é um trabalho que consome bastante tempo e está limitado a programas simples. Existe um esforço, tanto na indústria como na área acadêmica, para o desenvolvimento de ferramentas para automação de teste.

A geração de casos de teste é uma tarefa de alto custo. Um importante princípio do teste é que não se deve descartar os casos a menos que o programa seja descartável. É praticamente impossível que uma aplicação não sofra manutenção no decorrer do tempo, para corrigir erros, adicionar funcionalidades e assim por diante. Nesses casos, é necessário realizar os testes de regressão, a fim de verificar se as novas funcionalidades foram implementadas corretamente e as alterações não interferiram em outras partes do programa. Se os testes forem feitos manualmente, não existe a possibilidade de repeti–los.

3.3 Considerações Finais

Neste capítulo foi apresentada uma descrição sucinta do protótipo desenvolvido para tornar possível a definição da abordagem de teste apresentada no próximo capítulo. Os testes iniciais realizados não seguiram nenhum critério. Considerando as limitações que uma abordagem *ad hoc* para o teste do protótipo apresenta, no próximo capítulo é descrita a abordagem de teste proposta por este trabalho.

Capítulo

4

Abordagem de Teste Proposta

Neste capítulo é apresentada a abordagem de teste proposta para o teste de aplicações que manipulam ontologias expressas em OWL. Inicialmente são apresentados os níveis do teste de aplicações web; em seguida é descrita a abordagem e explicado porque o critério Particionamento de Equivalência foi escolhido; são mostrados exemplos de testes realizados e, por último, os resultados são discutidos.

4.1 Níveis de Teste

O teste de uma aplicação do tipo abordado neste trabalho envolve três níveis: teste da base de conhecimento, teste da camada de processamento de consultas e teste da camada de apresentação.

O teste da base de conhecimento, que corresponde ao teste da camada de dados em aplicações convencionais, deve ser realizado utilizando–se os métodos de teste específicos para validação de ontologias, como os expostos na Seção 2.9 e não são o foco deste trabalho.

O teste da camada de apresentação em nada difere do teste de aplicações convencionais. Os pontos básicos a serem observados, para aplicações web, são apresentados por Myers (2004). Entre outros, são:

- Verificar se as fontes são uniformes nos principais navegadores;
- Assegurar que os *links* apontam para os arquivos ou endereços corretos;

- Conferir se não há erros gramaticais; e
- Verificar se os elementos gráficos são visualizados de maneira uniforme nos diversos navegadores.

O foco deste trabalho é a camada de processamento de consultas, que é abordada no restante deste capítulo.

4.2 Abordagem Proposta

Este trabalho envolve duas áreas de conhecimento cuja integração é pouco abordada na literatura: teste de software e ontologias. Por um lado, a atividade de teste tem sido objeto de muitos estudos teóricos e empíricos, tais como os descritos por Beizer (1990); por outro lado, ontologias têm sido alvo de vários trabalhos (Subseção 2.9.1). No entanto, o teste de aplicações que manipulam ontologias parece ser pouco explorado. Embora as técnicas e critérios de teste apresentados na Seção 2.3 tenham sido desenvolvidos para serem aplicadas a programas convencionais, nesta pesquisa investigou-se se é possível aplicá–los aos programas que fazem consultas em ontologias. Devido às características das aplicações que manipulam ontologias expressas em OWL, o critério Particionamento de Equivalência foi adotado pelo exposto a seguir.

4.2.1 Particionamento de Equivalência

O critério de teste Particionamento de Equivalência é bastante conhecido e considerado um dos mais simples. No entanto, é tido como um critério fraco, no sentido de ter pouca probabilidade de revelar os defeitos do programa (Myers, 2004). Porém, no decorrer desta pesquisa, constatou–se que esse critério é adequado para os propósito de testes de programas que manipulam ontologias.

Considerando que o conhecimento sobre determinado domínio é representado em ontologias por meio de classes e os relacionamentos entre os indivíduos dessas classes, o particionamento de equivalência aplica–se nas seguintes situações de teste: 1) relação de classes e indivíduos; 2) relação entre classes na generalização/especialização; 3) relacionamentos entre indivíduos das classes.

Na primeira situação, pode–se dividir (particionar) os indivíduos de uma ontologia em dois conjuntos de equivalência: os indivíduos pertencentes a classe sob teste e os indivíduos não pertencentes a essa classe. No contexto do teste, esta divisão aplica–se

para verificar se indivíduos de uma classe não são reconhecidos como indivíduos de outra classe.

Quanto à relação entre classes na generalização/especialização, segunda situação de teste, pode–se dividir as classes em dois conjuntos de equivalência: as classes que são subclasses da classe sob teste e aquelas que não são.

Na terceira situação de teste, os relacionamentos entre indivíduos são definidos por meio de "propriedades" em ontologias. Uma propriedade vincula um sujeito a um objeto, ligando um indivíduo a um conjunto de outros indivíduos ou dados. Assim, para a terceira situação de teste, pode–se particionar o escopo de um indivíduo ao menos em dois conjuntos, os elementos que podem ser objetos de um relacionamento (elementos válidos) e indivíduos que não podem ser objeto do relacionamento (elementos inválidos). Por exemplo, uma `doença` tem um conjunto de `sintomas`. Assim, para uma doença, existem os sintomas válidos (são sintomas da doença) e sintomas inválidos (embora possam ser sintomas, não se aplicam à doença analisada). Na Figura 4.1 é ilustrado o exemplo.

Percebe–se, pelo exposto, que uma consulta sempre retorna um conjunto (conjunto vazio, unitário ou com diversos elementos). Devem ser criados, então, casos de teste com elementos representativos dos conjuntos válidos e inválidos, o que torna pertinente a utilização do critério.

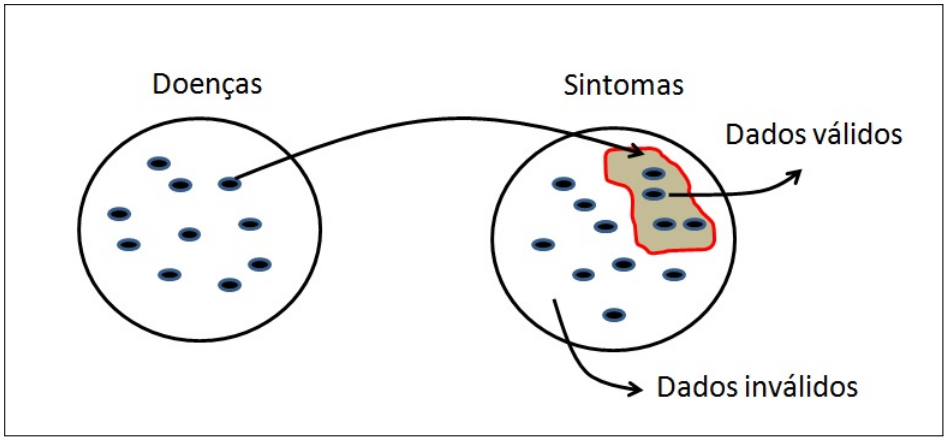

Figura 4.1: Exemplo de particionamento de equivalência.

4.2.2 Características dos Programas

Os programas que realizam consultas em ontologias formalizadas em OWL refletem as características dessa linguagem. OWL possui classes, classes agrupam recursos e recursos têm propriedades. Todos esses elementos devem ser levados em conta durante os testes.

A seguir são analisadas as características dos elementos representados na ontologia, que são: classes e relacionamentos.

a)Classes

As principais características de classes OWL relevantes para o teste são:

- Classes possuem indivíduos;
- Classes podem possuir subclasses; e
- Classes podem ser especialização de superclasses.

O que será testado depende de cada aplicação. O primeiro item da lista anterior aplica–se praticamente a todos programas. Nas situações em que é necessário recuperar todos os indivíduos de uma classe, os testes devem responder a duas perguntas:

1. O programa, para uma determinada classe, recupera todos os indivíduos?
2. O programa, para uma determinada classe, recupera somente os indivíduos pertencentes à classe?

Os mecanismos de consulta de Jena geralmente recuperam listas de elementos. Erros típicos ocorrem quando o programador supõe que apenas um valor vai ser retornado quando será uma lista de valores. Também é comum não se prever que o conjunto possa estar vazio ou não se percorrer totalmente uma lista.

No caso da segunda questão, erros acontecem quando indivíduos da superclasse são recuperados quando não deveriam ser ou quando indivíduos deixam de ser retornados nas situações que deveriam ser recuperados. A simples troca do método `listInstances(void)` pelo método `listInstances(boolean)` é suficiente para se obter resultados não esperados.

Se, no contexto da aplicação sob teste, for relevante recuperar as subclasses ou superclasses, as duas questões anteriores devem ser repetidas para cada caso.

b)Propriedades

Um conceito central em ontologias é o de "propriedade". Uma propriedade é o elemento que expressa os relacionamentos entre os indivíduos, relacionamentos esses que fornecem o conhecimento objeto da ontologia. Uma vez que o relacionamento liga um sujeito a um objeto, tem–se um indivíduo ligado a um conjunto de outros indivíduos ou dados.

Um pseudo-algoritmo para o teste de relacionamentos é apresentado na Figura 4.2, elaborado com base na tripla *sujeito - predicado - objeto*.

```
para cada classe C
   para cada propriedade P
      selecionar sujeito S entre indivíduos de C
      particionar o escopo de C em conjuntos válidos e conjuntos inválidos
      escrever casos de teste para exercitar os conjuntos válidos e inválidos
   fimpara
fimpara.
```

Figura 4.2: Pseudo-algoritmo para teste de relacionamentos.

O algoritmo visa exercitar os relacionamentos entre indivíduos de uma classe e os indivíduos do escopo de cada relacionamento. Para isso, seleciona–se um indivíduo da classe (sujeito) para o relacionamento sob teste. Para este indivíduo, divide–se os indivíduos do escopo da classe entre os que podem ser objeto do relacionamento e os indivíduos que não podem ser objetos desse predicado. Ao menos um caso de teste para cada conjunto deve ser elaborado. Além disso, um caso de teste adicional pode ser criado com um indivíduo fora do escopo.

Uma vez que as propriedades se dividem em propriedades de tipos de dados e propriedades de objeto, casos de teste devem ser criados para cada um desses tipos de propriedades.

4.2.3 Passos do Teste

Além de utilizar os procedimentos anteriormente descritos, o processo de teste, como toda atividade, deve seguir algumas etapas ou passos. Neste trabalho propõe–se os seguintes passos para teste de programas que manipulam ontologias:

1. Definir o que será testado: definir que elemento da ontologia será testado, como por exemplo, propriedades de objeto;

2. Identificar erros típicos: nesta etapa são levantados os erros comuns que podem ocorrer em relação ao elemento testado;

3. Identificar as restrições sobre os elementos: cardinalidade, disjunções e simetrias são exemplos dessas restrições;

4. Particionar os elementos de entrada e saída: os elementos são divididos em classes válidas e classes inválidas;

5. Escolher os dados de teste: devem ser selecionados dados representativos de cada conjunto para serem utilizados nos casos de teste;

6. Escrever os casos de teste: as entradas e saídas esperadas são especificadas; e

7. Executar os casos e examinar a saída obtida: por último, os casos de teste são executados e o resultado é examinado para verificar se o caso de teste "passou" ou "falhou".

Uma decisão fundamental para aplicação do particionamento de equivalência é determinar quais dados utilizar, como previsto no passo 4. Em se tratando de um critério de teste funcional, os requisitos de teste devem ser derivados da especificação funcional do software. No presente trabalho, as questões de competência serviram como especificação, pois essas questões explicitam qual conhecimento a aplicação deve fornecer. No entanto, diferentemente de aplicações convencionais, para as quais a especificação descreve o comportamento ou a resposta para uma dada entrada ou ação do usuário, as questões de competência não mostram a resposta esperada.

Para contornar o problema, no decorrer deste trabalho, foi utilizado o Protégé como "oráculo" de teste. As consultas foram primeiramente realizadas no Protégé e as respostas usadas como "resultado esperado" durante os testes. Para saber qual o resultado esperado, no critério Particionamento de Equivalência deve–se definir dois conjuntos de dados, os dados válidos e os dados inválidos. Com relação aos dados válidos não há dificuldade em obtê–los, porém determinar quais os dados inválidos é uma tarefa mais complexa, uma vez que o conjunto dos possíveis dados é muito grande. Se esta seleção não for cuidadosa, corre–se o risco de serem utilizados dados que não expressam a realidade. Assim, buscou–se nas próprias características da ontologia a resposta para o problema, como por exemplo, propriedades como disjunção e assimetria fornecem indicações sobre dados que não podem ser válidos para uma determinada consulta. Nesse sentido, sabendo–se que a classe `Carbamatos` é disjunta (*disjointWith*) em relação à classe `OrganoClorados`, é mais indicado utilizar indivíduos desta classe como candidatos a elementos inválidos durante os testes da classe `Carbamatos` do que outra classe qualquer.

4.3 Aplicação da Abordagem

Os procedimentos apresentados na Seção 4.2 foram utilizados no teste de uma parte do protótipo descrito no Capítulo 3. A seguir, são apresentadas características gerais das ferramentas utilizadas e descritos os testes realizados.

4.3.1 Ferramentas Utilizadas

O processo de execução de teste consiste, basicamente, em executar um caso de teste e examinar se a saída obtida confere com a saída esperada. Uma tarefa importante, nesse sentido, é determinar qual a saída esperada para uma determinada entrada. Utilizou–se, para isso, o próprio Protégé. Para saber qual a saída esperada para uma consulta, realizou–se a consulta primeiramente no Protégé.

Procurou–se, para os testes aqui apresentados, abordar dois problemas relacionados ao teste. O primeiro diz respeito à execução manual; o segundo, à necessidade de repetir os testes. Para isso, o *framework* JUnit [1] foi utilizado, integrado ao ambiente (IDE) NetBeans [2].

a) JUnit

JUnit é um *framework* que facilita a criação semiautomática de testes, a execução dos testes e exibição dos resultados. Sem a utilização de ferramentas desse tipo, geralmente, os programadores utilizam comandos como o `println( )` de Java para exibição de resultados. Isso tem a consequência de "sujar" o código, pois muitas vezes as instruções adicionais são esquecidas junto com o código do programa. Com JUnit o código do teste é independente do código da aplicação.

JUnit possui uma variedade de métodos *asserts* que permitem verificar automaticamente os resultados dos testes. Um resumo dos diversos *asserts* é apresentado na Tabela 4.1.

Tabela 4.1: Principais métodos *assert* de JUnit.

Método	**Descrição**
assertEquals	verifica se dois parâmetros são iguais
assertFalse	verifica se uma condição é falsa
assertTrue	verifica se uma condição é verdadeira
assertNotNull	verifica se um objeto não é nullo

Na Figura 4.3 é apresentado um exemplo de método de teste. O objetivo é testar o método `sintomaPertenceDoenca` da classe sob teste `Inferencia`. Na linha 3, um objeto é instanciado, na linha 4 é definido o resultado esperado (`expResult`) e na linha 6 são passados dois parâmetros, a doença e o sintoma. Como "Falta de apetite" é sintoma de "Depressão", o resultado esperado é o valor lógico `verdadeiro`. O método `assertEquals` (linha 7) verifica se as variáveis `expResult` e `result` são iguais.

[1] www.junit.org.
[2] www.netbeans.org

```
/* 1 */ Test
/* 2 */ public void testInferir() {
/* 3 */    Inferencia inf = new Inferencia();
/* 4 */    boolean expResult = true;
/* 5 */    boolean result = false;
/* 6 */    result = inf.sintomaPertenceDoenca("Depressão","Falta de apetite");
/* 7 */    assertEquals(expResult, result);
/* 8 */ }
```

Figura 4.3: Exemplo de método de teste.

A utilização de JUnit permite que casos de teste sejam mantidos para reutilização. Testes de regressão, por exemplo, podem ser executados com um custo pequeno.

b) NetBeans

JUnit é um *framework* que pode ser utilizado isoladamente ou então ser integrado a ambientes de desenvolvimento como NetBeans e Eclipse. A utilização em conjunto possibilita que a codificação e o teste sejam executados no mesmo ambiente. Os casos de teste são mantidos em diretório separado dos diretórios que contêm o código do programa, como pode ser visto na Figura 4.4. Quando ocorre a distribuição da aplicação, esses arquivos não precisam ser disponibilizados.

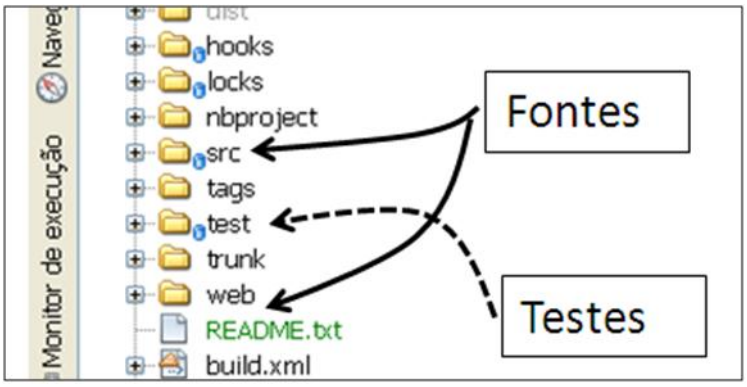

Figura 4.4: Diretório contendo os casos de teste e fontes.

4.3.2 Testes Realizados

A investigação de todas as classes e relacionamentos na Ontotox seria muito custoso em termos de tempo. Isso exigiu uma limitação na abrangência dos casos de teste, não sendo possível o teste de todas as questões de competência apresentadas na Seção A.2. Por isso,

decidiu–se utilizar a classe `Agrotoxico` e seus indivíduos para realizar a investigação. Essa classe é a que possui o maior número de relacionamentos.

A seguir são descritos testes realizados para classes e indivíduos seguindo a abordagem proposta.

a) Teste de Classes e Indivíduos

As aplicações que manipulam ontologias, geralmente, utilizam uma classe Java responsável por gerenciar as funcionalidades referentes à classe correspondente na ontologia. Uma das funcionalidades típicas é recuperar todos os indivíduos pertencentes à classe durante uma consulta. Por exemplo, para responder a questão de competência "Quais as doenças relacionadas ao metabolismo celular?", é necessário recuperar todos os indivíduos que são instâncias da classe `MetabolismoCelular`.

Seguindo os passos apresentados na Subseção 4.2.3, os testes foram elaborados como apresentado na Tabela 4.2.

Tabela 4.2: Teste do método `obterIndividuos()` para classe `MetabolismoCelular`.

Passo	**Descrição**
1. Definir o que será testado	Verificar se, para a classe `MetabolismoCelular`, o programa recupara todos os indivíduos durante uma consulta.
2. Identificar erros típicos	Uma vez que a consulta retorna uma coleção de objetos, deixar de percorrer completamente a coleção é um erro que pode ocorrer neste caso. Outro erro é recuperar indivíduos que não pertençam à classe.
3. Identificar as restrições sobre os elementos	Pode haver classes que, em determinado momento, não possuam indivíduos. Neste caso, é necessário criar caso de teste que verifique se o programa trata esta situação.
4. Particionar os elementos de entrada e saída	Para uma classe, o conjunto dos elementos válidos é constituído pelos indivíduos da classe; os inválidos, por indivíduos que não pertencem à classe.
5. Escolher os dados de teste	Selecionar um indivíduo do conjunto válido e um do conjunto inválido.
6. Escrever os casos de teste	As entradas e saídas esperadas são especificadas.
7. Executar os casos e examinar a saída obtida	Por último, os casos de teste são executados e o resultado é examinado para verificar se o caso de teste "passou" ou "falhou".

No exemplo apresentado na Tabela 4.2 é resumido o teste da consulta sobre os indivíduos da classe `MetabolismoCelular`. Uma vez que consultas em ontologias retornam coleções de indivíduos, o algoritmo deve tratar o resultado da consulta como um conjunto. Assim, deixar de percorrer a coleção é um erro que pode ocorrer nessas situações. Além disso, é preciso assegurar que o resultado da consulta contenha apenas os indivíduos pertencentes à classe sob consulta. Quanto às restrições, não foram identificadas para esse caso. O passo seguinte é particionar os elementos de saída em indivíduos válidos e inválidos. Considera–se como válido o elemento que é indivíduo da classe `MetabolismoCelular`, e como elemento inválido qualquer indivíduo que não pertença a esta classe. Ao serem escritos os casos de teste, ao menos um elemento do conjunto válido e um do conjunto inválido deve ser utilizado. No trecho de código apresentado na Figura 4.5 é mostrado o caso de teste correspondente. Na linha 1 é executado o método sob teste; na linha 3, é verificado se o resultado contém o indivíduo válido (`CancerDeProstata`); na linha 4, é verificado se não contém o indivíduo inválido (`Depressao`) e na linha 5, é verificado se a quantidade de indivíduos retornados corresponde ao total de indivíduos pertencentes à classe.

```
/* 1 */ ArrayList<String> result =
/* 2 */      instance.listarIndividuos("MetabolismoCelular");
/* 3 */ Assert.assertTrue(result.contains("CancerDeProstata"));
/* 4 */ Assert.assertFalse(result.contains("Depressao"));
/* 5 */ Assert.assertEquals(5, result.size());
```

Figura 4.5: Trecho de código do método `listarIndividuos()`.

b) Teste de Classes e Subclasses

Em algumas situações, é necessário recuperar as subclasses de uma classe durante uma consulta, por exemplo, para responder a questão de competência "Quais agrotóxicos estão relacionados a um determinado sintoma de intoxicação informado?". Nesse caso, é preciso obter as subclasses de `Agrotoxico`, uma vez que os sintomas estão relacionados com essas subclasses. Em outras palavras, um sintoma não está associado a um indivíduo, mas a um "grupo" de agrotóxico. Na Figura 4.6 é mostrado um fragmento do diagrama de classe da OntoTox, no qual é possível observar que há dois níveis na hierarquia de herança, sendo que para responder a questão de competência mencionada, apenas o primeiro nível deve ser considerado.

Na Tabela 4.3 estão apresentados os passos para o teste deste relacionamento de especialização.

Os casos de teste para esta consulta são mostrados na Figura 4.7. O teste verifica se todas as subclasses foram retornadas (linha 7) bem como se apenas as subclasses

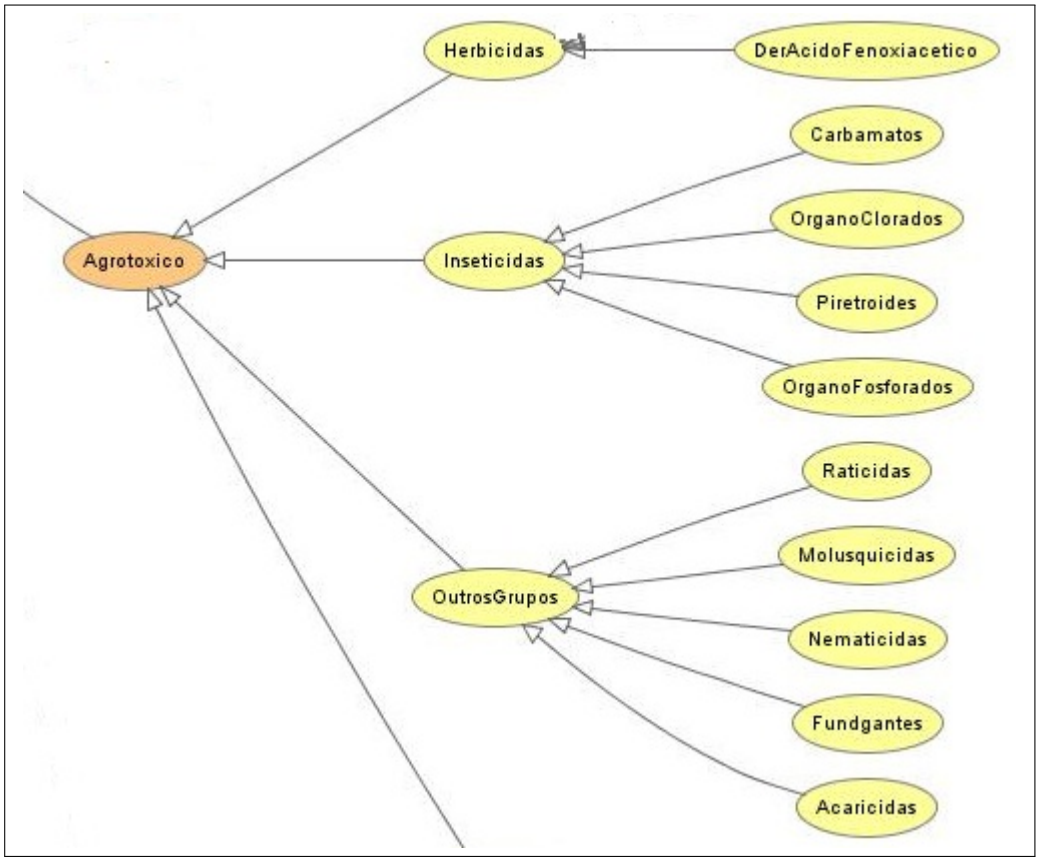

Figura 4.6: Subclasses da classe `Agrotoxico`.

esperadas foram obtidas. Nesse caso, foi possível verificar todos os elementos, uma vez que são apenas quatro. No caso de uma quantidade maior de elementos, deve–se utilizar amostragem.

```
/* 1 */Agrotoxico instance = new Agrotoxico();
/* 2 */ArrayList<String> expectedResult = new ArrayList();
/* 3 */expectedResult.add("Fungicidas");
/* 4 */expectedResult.add("Herbicidas");
/* 5 */expectedResult.add("Inseticidas");
/* 6 */expectedResult.add("OutrosGrupos");
/* 7 */ArrayList<String> result = instance.obterGrupos();
/* 8 */assertTrue(result.containsAll(expectedResult));
/* 9 */assertTrue(result.size(), 4);
```

Figura 4.7: Trecho de código do método `testObterGrupos()`.

Tabela 4.3: Teste do método `obterGrupos()` da classe `Agrotoxico`.

Passo	Descrição
1. Definir o que será testado	Verificar se, para a classe `Agrotoxico`, o programa recupera todas as subclasses durante uma consulta.
2. Identificar erros típicos	Neste caso, só interessa recuperar as subclasses do primeiro nível de herança; portanto, recuperar subclasses de níveis inferiores da árvore de herança é um erro.
3. Identificar as restrições sobre os elementos	Não devem ser recuperadas as subclasses das subclasses, em outras palavras, apenas o primeiro nível de subclasses.
4. Particionar os elementos de entrada e saída	O conjunto dos elementos válidos é constituído pelas classes que são herdeiras diretas de `Agrotoxico`. Os elementos inválidos dividem–se em dois conjuntos: as classes herdeiras indiretas de `Agrotoxico` e as classes não herdeiras.
5. Escolher os dados de teste	Selecionar um indivíduo do conjunto válido e um de cada conjunto inválido.
6. Escrever os casos de teste	Escrever o *script* para realização do teste (Figura 4.7).
7. Executar os casos e examinar a saída obtida	Por último, os casos de teste são executados e o resultado é examinado para verificar se o caso de teste "passou" ou "falhou".

c)Teste de Propriedades

Conforme exposto na Subseção 4.2.2, uma importante característica de ontologias refere–se às propriedades. Uma vez que as propriedades estão divididas em propriedades de objeto e propriedades de dados, os tipos serão analisados separadamente, os quais estão descritos a seguir.

c.1) Propriedades de Objeto

Propriedades de objeto relacionam indivíduos de uma classe a indivíduos da mesma ou de outra classe. Por exemplo, um indivíduo da classe `Agrotoxico` tem a propriedade `isAplicadoEm`, que relaciona o indivíduo a um conjunto de indivíduos da classe `ProdutosAgropecuarios`. Neste contexto, a classe `Agrotoxico` constitui o `domínio` do relacionamento, enquanto que a classe `ProdutosAgropecuarios` é o `escopo`.

Seguindo o critério Particionamento de Equivalência, dado um indivíduo i, para uma propriedade p, os objetos do relacionamento devem ser particionados em três classes: 1) indivíduos do escopo do domínio de i, que são objetos para o relacionamento

estabelecido por p, 2) indivíduos do escopo que não são objetos para o relacionamento estabelecido por p e 3) indivíduos que estão fora do escopo.

Para o teste das propriedade de objeto da classe `Agrotoxico`, levantou–se todos os relacionamentos possíveis. Isso significa listar cada propriedade de objeto que possa constituir um predicado em uma tripla em que um indivíduo da classe `Agrotoxico` seja sujeito. As propriedades estão relacionadas na Tabela 4.4.

Tabela 4.4: Propriedades dos indivíduos da classe `Agrotoxico`.

Domínio	**Propriedade**	**Escopo**
`Agrotoxico`	`hasTipoAgrotoxico`	`AgrotoxicoGrupo`
`Agrotoxico`	`hasIntoxicacaoRelacionada`	`Intoxicacao`
`Agrotoxico`	`hasDoencasRelacionadas`	`Doencas`
`Agrotoxico`	`hasSinonimoAgrotoxico`	`Sinonimos`
`Agrotoxico`	`isAplicadoEm`	`ProdutosAgropecuarios`
`Agrotoxico`	`hasEmpresaRegistrante`	`EmpresaRegistrante`
`Agrotoxico`	`hasIngredienteAtivo`	`IngredienteAtivo`

Em seguida, para cada propriedade de objeto, criou–se casos de teste para verificar se o método da classe Java correspondente fornece a resposta esperada. A seguir são mostrados dois exemplos de teste de propriedades.

Exemplo 1 - Propriedade `hasEmpresaRegistrante`

A propriedade (`hasEmpresaRegistrante`) estabelece o relacionamento de um indivíduo da classe `Agrotoxico` com um indivíduo da classe `EmpresaRegistrante`. A consulta desse relacionamento responde a questão de competência "Qual empresa comercializa um determinado agrotóxico informado? ". Os passos seguidos para a criação dos testes são mostrados na Tabela 4.5.

No trecho de código descrito na Figura 4.8 é mostrado um caso de teste (linhas 4 e 5) do qual se espera uma resposta verdadeira (`BASF` é a empresa registrante do `Piretroides38`). No outro caso (linhas 6 e 7), espera–se uma resposta negativa (`BASF` não é a empresa registrante do `Piretroides27`). Um caso de teste com um indivíduo que não pertence ao escopo também foi criado (linhas 8 e 9).

Exemplo 2 - Propriedade `hasIngredienteAtivo`

A propriedade (`hasIngredienteAtivo`) estabelece o relacionamento de um indivíduo da classe `Agrotoxico` com um indivíduo da classe `IngredienteAtivo`. Saber qual é o ingrediente ativo do agrotóxico é necessário porque as intoxicações, muitas vezes, são vinculadas ao composto químico e não ao nome comercial do produto.

Tabela 4.5: Teste da propriedade `hasEmpresaRegistrante` da classe `Agrotoxico`.

Passo	**Descrição**
1. Definir o que será testado	Verificar se, dado um indivíduo da classe `Agrotoxico`, o programa recupera a empresa registrante correspondente.
2. Identificar erros típicos	Para esta consulta simples, o principal erro possível é não retornar a empresa correspondente ao agrotóxico informado.
3. Identificar as restrições sobre os elementos	Só deve haver uma empresa registrante para cada indivíduo.
4. Particionar os elementos de entrada e saída	O conjunto válido é constituído pelo único indivíduo da classe `EmpresaRegistrante` que corresponde ao agrotóxico informado. Dois conjuntos inválidos podem ser estabelecidos: os indivíduos da classe `EmpresaRegistrante` que não são objeto do relacionamento e indivíduos de qualquer outra classe.
5. Escolher os dados de teste	Selecionar o indivíduo válido e um de cada conjunto inválido.
6. Escrever os casos de teste	As entradas e saídas esperadas são especificadas (Figura 4.8).
7. Executar os casos e examinar a saída obtida	Por último, os casos de teste são executados e o resultado é examinado para verificar se o caso de teste "passou" ou "falhou".

```
/* 1*/ public void testGetEmpresaRegistrante() {
/* 2*/    Agrotoxico agt = new Agrotoxico();
/* 3*/    String expectedResult = "BASF";
/* 4*/    String result = agt.getEmpresaRegistrante("Piretroides38");
/* 5*/    assertEquals(expectedResult, result);
/* 6*/    result = agt.getEmpresaRegistrante("Piretroides27");
/* 7*/    assertFalse(expectedResult.equalsIgnoreCase(result));
/* 8*/    result = agt.getEmpresaRegistrante("IngredienteAtivo89");
/* 9*/    assertFalse(expectedResult.equalsIgnoreCase(result));
/*10*/ }
```

Figura 4.8: Caso de teste para a propriedade `hasEmpresaRegistrante`.

Os passos seguidos para a criação dos testes são mostrados na Tabela 4.6.

No trecho de código descrito na Figura 4.9 é mostrado um caso de teste em que são consultados os ingredientes ativos para o agrotóxico "Piretroides46" (linha 6). Neste caso de teste é verificado se a resposta contém o ingrediente válido ("IngredienteAtivo14", linha 7) e não contém o ingrediente inválido ("IngredienteAtivo15", linha 8).

Tabela 4.6: Teste do método `getIngredienteAtivo` da classe `Agrotoxico`.

Passo	**Descrição**
1. Definir o que será testado	Verificar se, dado um indivíduo da classe `Agrotoxico`, o programa recupera os ingredientes ativos correspondentes.
2. Identificar erros típicos	O programa a)retornar os ingredientes ativos incorretos ou b)devolver apenas um ingrediente nos casos em que há mais de um.
3. Identificar as restrições sobre os elementos	Podem haver mais de um ingrediente ativo para cada agrotóxico.
4. Particionar os elementos de entrada e saída	O conjunto válido é constituído pelos indivíduos da classe `IngredienteAtivo` que correspondem ao agrotóxico informado. O conjunto de dados inválidos é constituído pelos indivíduos que não são ingredientes ativos do agrotóxico informado.
5. Escolher os dados de teste	Selecionar o indivíduo válido e um do conjunto inválido.
6. Escrever os casos de teste	As entradas e saídas esperadas são especificadas (Figura 4.9).
7. Executar os casos e examinar a saída obtida	Por último, os casos de teste são executados e o resultado é examinado para verificar se o caso de teste "passou" ou "falhou".

```
/* 1*/ public void testGetIngredienteAtivo() {
/* 2*/    Agrotoxico agro = new Agrotoxico();
/* 3*/    Ontotox instance = new Ontotox();
/* 4*/    OntModel ont = instance.getOntologia();
/* 5*/    Individual i = ont.getIndividual("Piretroides46");
/* 6*/    ArrayList<String> lista = agro.getIngredienteAtivo(i);
/* 7*/    assertTrue(lista.contains("IngredienteAtivo14");
/* 8*/    assertFalse(lista.contains("IngredienteAtivo15");
/* 9*/ }
```

Figura 4.9: Caso de teste para a propriedade `hasIngredienteAtivo`.

c.2) Propriedades de Tipos de Dados

Além das propriedades de objeto, indivíduos possuem propriedades de tipos de dados (`DatatypeProperties`). O teste das propriedades de dados deve ser realizado da mesma maneira que o teste das propriedades de objeto. Para cada classe, deve–se levantar todas as possíveis propriedades de dados que seus indivíduos podem ter e cada propriedade deve ser exercitada ao menos uma vez. Na Tabela 4.7 estão relacionadas todas as propriedades da classe `Agrotoxico`.

Tabela 4.7: Propriedades de tipos de dados da classe `Agrotoxico`.

Propriedade	**Escopo**
`Caracteristicas`	`string`
`ClasseToxicologica`	`string`
`Descricao`	`string`
`Formula`	`string`
`GrauDeIntoxicacao`	`string`
`Nome`	`string`
`NomeCientifico`	`string`

A seguir são apresentados dois exemplos de testes com propriedades de dados.

Exemplo 1 - Propriedade `Formula`

Na Tabela 4.8 são mostrados os passos para o teste da propriedade `Formula`. Esse teste foi um dos que revelaram erro no programa. Como nem todos os agrotóxicos possuem a fórmula cadastrada, o método de consulta gerava uma exceção (*nullPointerException*) quando um agrotóxico sem fórmula era utilizado como dado de teste. Isso ocorreu porque o programador não previu que, para alguns indivíduos, não havia informação correspondente a ser consultada.

Os casos de teste mostrados na Figura 4.10 cobrem as partições estabelecidas no passo 4. Assim, na linha 5 é testado o caso da fórmula correta; na linha 8, o caso inválido; na linha 11 o caso válido para um agrotóxico que não possui fórmula cadastrada (`Piretroides27`) e, na linha 14, o caso inválido para esse mesmo agrotóxico.

Tabela 4.8: Teste da propriedade `Formula` da classe `Agrotoxico`.

Passo	**Descrição**
1. Definir o que será testado	Verificar se, dado um indivíduo da classe `Agrotoxico`, o programa recupera a fórmula correspondente.
2. Identificar erros típicos	Para esta consulta simples, o principal erro possível é retornar uma fórmula inválida.
3. Identificar as restrições sobre os elementos	É possível haver agrotóxico sem fórmula catalogada.
4. Particionar os elementos de entrada e saída	Para o caso dos indivíduos que possuem fórmula catalogada, o conjunto válido é constituído pela *string* que corresponde à fórmula do agrotóxico informado; o conjunto inválido, por uma *string* que não é a fórmula do agrotóxico. Para o caso dos indivíduos que não possuem a fórmula catalogada, o conjunto válido é constituído por uma *string* "vazia"; o conjunto inválido, por uma *string* de comprimento maior que zero.
5. Escolher os dados de teste	Para cada um dos casos, selecionar o indivíduo válido e um inválido.
6. Escrever os casos de teste	As entradas e saídas esperadas são especificadas (Figura 4.10).
7. Executar os casos e examinar a saída obtida	Por último, os casos de teste são executados e o resultado é examinado para verificar se o caso de teste "passou" ou "falhou".

Exemplo 2 - Propriedade `ClasseToxicologica`

Na Tabela 4.9 são mostrados os passos para o teste da propriedade `ClasseToxicologica`. Essa informação é importante ao agente de saúde, pois a classe toxicológica indica as medidas de segurança contra riscos para a saúde humana.

Na Figura 4.11 são mostrados os testes realizados para o método Java responsável por obter a propriedade `ClasseToxicologica`. Os casos de teste cobrem tanto o resultado esperado (dado válido) na linha 8, quanto o resultado não esperado (dado inválido) na linha 10.

```
/* 1*/ public void testGetFormula(){
/* 2*/    Agrotoxico instance = new Agrotoxico();
/* 3*/    expectedResult = "C21H20Cl2O 3";
/* 4*/    result = instance.getFormula("Piretroides1");
/* 5*/    assertEquals(expectedResult, result);
/* 6*/    expectedResult = "20Cl2OC21H 3";
/* 7*/    result = instance.getFormula("Piretroides1");
/* 8*/    assertFalse(expectedResult == result);
/* 9*/    expectedResult = `` '';
/*10*/    result = instance.getFormula("Piretroides27");
/*11*/    assertEquals(expectedResult, result);
/*12*/    expectedResult = "C21H20Cl2O 3";
/*13*/    result = instance.getFormula("Piretroides1");
/*14*/    assertFalse(expectedResult == result);
/*15*/ }
```

Figura 4.10: Caso de teste para a propriedade `Formula`.

```
/* 1*/ public void testGetClasseToxicologica(){
/* 2*/    Agrotoxico agro = new Agrotoxico();
/* 3*/    Ontotox instance = new Ontotox();
/* 4*/    OntModel ont = instance.getOntologia();
/* 5*/    Individual i = ont.getIndividual("Piretroides46");
/* 6*/    expectedResult = "III - Azul (Medianamente Tóxico)";
/* 7*/    String classeTox = agro.getClasseToxicologica(i);;
/* 8*/    assertTrue(expectedResult.equals(classeTox);
/* 9*/    expectedResult = "II - Amarela (Altamente Tóxico)";
/*10*/    assertFalse(expectedResult.equals(classeTox);
/*11*/ }
```

Figura 4.11: Caso de teste para a propriedade `ClasseToxicologica`.

4.3.3 Relacionamentos Indiretos

Os casos apresentados realizam consultas em uma única classe ou envolvem apenas duas classes. Existem situações nas quais mais classes são necessárias para responder a uma consulta. Por exemplo, a questão de competência "Quais agrotóxicos estão relacionados a determinado sintoma de intoxicação?" é uma consulta mais complexa em relação às consultas realizadas para relacionamentos diretos. No fragmento do diagrama apresentado na Figura 4.12, gerado por meio do *plug-in* Jambalaya, são mostradas as classes necessárias para responder a essa questão.

Uma vez que não existe relacionamento direto entre `Agrotoxico` e `Sintomas`, é preciso recuperar, para um dado sintoma, qual a intoxicação relacionada e, a partir dela,

Tabela 4.9: Teste da propriedade `ClasseToxicologica` da classe `Agrotoxico`.

Passo	**Descrição**
1. Definir o que será testado	Verificar se, dado um indivíduo da classe `Agrotoxico`, o programa recupera a classe toxicológica correspondente.
2. Identificar erros típicos	O programa pode retornar uma classe toxicológica incorreta.
3. Identificar as restrições sobre os elementos	Cada agrotóxico pode ser classificado em uma classe.
4. Particionar os elementos de entrada e saída	O conjunto válido é constituído pela *string* que corresponde à classe toxicológica do agrotóxico informado; o conjunto inválido, por uma *string* que não é a classe toxicológica do agrotóxico.
5. Escolher os dados de teste	Selecionar o dado válido e um inválido.
6. Escrever os casos de teste	O caso de teste está descrito na Figura 4.11.
7. Executar os casos e examinar a saída obtida	Por último, os casos de teste são executados e o resultado é examinado para verificar se o caso de teste "passou" ou "falhou".

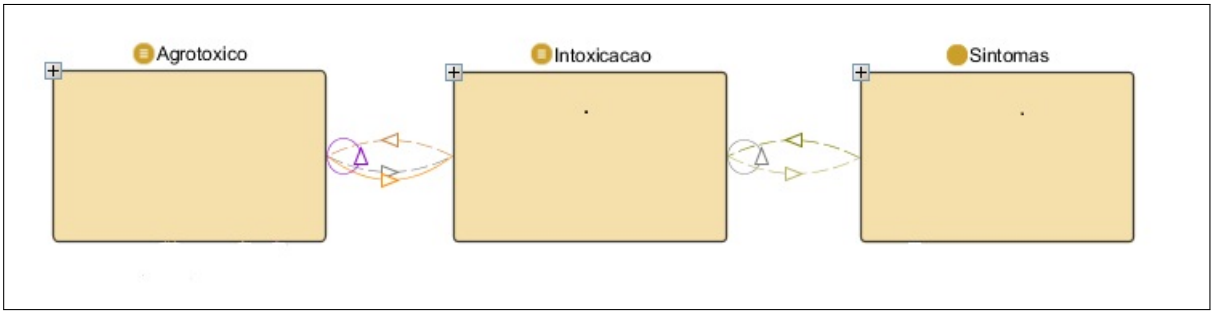

Figura 4.12: Relacionamento entre as classes `Sintomas`, `Intoxicacao` e `Agrotoxico`.

recuperar o agrotóxico associado. A questão que se colocou neste ponto foi se os testes desses relacionamentos poderiam ser realizados da mesma forma que os demais. Após a criação dos casos de teste conclui–se que os mesmos procedimentos podem ser seguidos, pois, em se tratando de testes funcionais, é necessário especificar a entrada e a saída, não importando qual o caminho seguido pelo algoritmo.

Na Tabela 4.10 são mostrados os passos para o teste da consulta que responde à questão de competência mencionada.

Na Figura 4.13 é mostrado o sintoma utilizado como entrada ("Formigamento nos lábios") na linha 5; a saída escolhida a partir da partição válida é especificada na

Tabela 4.10: Teste do relacionamento indireto entre `Agrotoxico` e `Sintomas`.

Passo	**Descrição**
1. Definir o que será testado	Verificar se, dado um sintoma de intoxicação, o programa recupera os agrotóxicos que podem ser os causadores desse sintoma.
2. Identificar erros típicos	Os dois principais erros que podem ocorrer são a)retornar um agrotóxico que não é causador do sintoma e b)não retornar todos os agrotóxicos.
3. Identificar as restrições sobre os elementos	Todo sintoma está relacionado a pelo menos um agrotóxico.
4. Particionar os elementos de entrada e saída	O conjunto de indivíduos válidos é constituído pelos agrotóxicos que são causadores do sintoma informado; o conjunto inválido, pelos agrotóxicos que não causam esse sintoma.
5. Escolher os dados de teste	Para cada um dos casos, selecionar o indivíduo válido e um inválido.
6. Escrever os casos de teste	A entrada e saídas esperadas são especificadas (Figura 4.13).
7. Executar os casos e examinar a saída obtida	Por último, os casos de teste são executados e o resultado é examinado para verificar se o caso de teste "passou" ou "falhou".

linha 7 ("Piretroides33" é causador do sintoma informado) e a saída inválida é mostrada na linha 8 ("Carbamatos4" não é causador do sintoma informado).

```
/* 1*/ public void testListarCausadorSintoma() {
/* 2*/    Sintoma sintoma = new Sintoma();
/* 3*/    Ontotox instance = new Ontotox();
/* 4*/    OntModel ont = instance.getOntologia();
/* 5*/    Individual i = ont.getIndividual("Formigamento nos lábios");
/* 6*/    ArrayList<String> lista = sintoma.listarCausadorSintoma(i);
/* 7*/    assertTrue(lista.contains("Piretroides33"));
/* 8*/    assertFalse(lista.contains("Carbamatos4"));
/* 9*/ }
```

Figura 4.13: Caso de teste do relacionamento entre `Agrotoxico` e `Sintomas` .

4.4 Resumo da Abordagem

Testes de software podem ser realizados manualmente, o que é pouco produtivo e sujeito a erros. As ferramentas de teste existentes são úteis, no entanto elas têm alto custo e a curva de aprendizado também é grande. Uma alternativa aqui apresentada é o *framework* JUnit. Com ele é possível criar casos de teste e mantê–los para serem utilizados durante a fase de manutenção, além de serem executados automaticamente.

Já a seleção de dados de teste, por sua vez, pode ser feita de forma aleatória ou utilizando–se algum critério. Os estudos realizados sugerem que o critério Particionamento de Equivalência é razoável para o tipo de programa aqui discutido.

Analisando–se as características dos programas, as recomendações para geração de casos de teste podem ser assim resumidas:

1. Para cada classe, crie casos de teste que verifiquem se o programa recupera corretamente todos os indivíduos;

2. Para cada classe, crie casos de teste que verifiquem se o programa trata adequadamente os casos em que não há indivíduos;

3. Se for relevante, crie casos de teste que verifiquem se o programa recupera as subclasses de uma dada classe, bem como para os casos em que não há subclasses;

4. Se for relevante, crie casos de teste que verifiquem se o programa recupera as superclasses de uma dada classe, bem como para os casos em que não há superclasses;

5. Para cada propriedade de objeto, crie casos de teste que exercitem a propriedade ao menos uma vez. Utilize objetos válidos e inválidos em casos de teste;

6. Para cada propriedade de dado, crie casos de teste que exercitem a propriedade ao menos uma vez. Utilize dados válidos; inválidos e nulos em casos de teste;

7. Para os recursos que possuem cardinalidade, crie casos de teste que verifiquem se a quantidade de objetos obtidos corresponde à cardinalidade esperada.

Essas recomendações devem ser adaptadas para cada programa e limitam–se aos programas que realizam apenas consultas. Devem ser estendidas, caso o programa realize inserções ou remoções de informações.

4.5 Resultados e Discussão

A motivação deste trabalho foi a necessidade de se ter uma abordagem diferenciada de teste para programas que manipulam ontologias. Os métodos de teste encontrados na literatura (Seção 2.9) objetivam avaliar e validar apenas a ontologia. A proposta que mais se aproxima da presente pesquisa é o trabalho de Porzel e Malaka (2004), que propõe a avaliação por meio da utilização da ontologia em uma aplicação. Mesmo assim, a proposta é testar a ontologia, e não o programa ou a integração entre os dois. Mesmo o trabalho de El-Korany (2007), que propõe um processo de teste, apenas enfatiza a necessidade de se manter um repositório de casos de teste para realizar os testes de regressão, sem mencionar como criar os casos de teste.

Entre os critérios de teste, o Particionamento de Equivalência mostrou–se adequado. Alguns dos outros critérios foram analisados e revelaram–se incompatíveis. O critério Análise de Valor Limite, por exemplo, aplica–se nos casos em que a entrada é um dado numérico. Esse critério propõe que se escolha dados nos limites da faixa válida para uma dada entrada. No caso de ontologias, a entrada de dados utilizada como parâmetro de uma questão de competência provavelmente não será um valor numérico. Outros critérios, como os baseados em modelos, são mais adequados para software orientado a objetos e não foi possível identificar se existe a possibilidade de aplicá-los na manipulação de ontologias.

Neste trabalho, o objetivo inicial foi buscar uma abordagem que revele defeitos na integração entre o programa e a ontologia. Em se tratando de uma estratégia funcional, deve–se utilizar uma especificação como fonte de geração de casos de teste. Na ausência de uma especificação sobre qual conhecimento a ontologia deveria fornecer, foram utilizadas as questões de competência.

Os *plugins* da ferramenta Protégé geram alguns diagramas automaticamente. Jambalaya, por exemplo, é útil na visualização da árvore de classes, bem como na visualização de relacionamentos *domínio* $\leftrightarrow$ *escopo*, minimizando o problema. No entanto, se possível, uma outra fonte mais detalhada deve ser utilizada.

Durante os testes, alguns erros foram encontrados. Esses erros podem ser divididos em duas categorias:

- engano por parte do programador, e
- ausência de especificação completa.

Na primeira categoria estão os erros gerados por um entendimento errado das funcionalidades que o software deve apresentar ou uma suposição errônea de algum

detalhe sobre como os dados devem ser manipulados. Por exemplo, ao ser testado o método `getIsIntoxicacaoOf` da classe `Intoxicacao`, esperava–se a lista dos possíveis agrotóxicos causadores do sintoma informado, porém, o programa retornou uma lista vazia. Isso ocorreu porque todas as propriedades na ontologia estavam escritas com a letra inicial minúscula e a propriedade sob teste estava escrita com a letra inicial maiúscula (`IsIntoxicacaoOf` ao invés de `isIntoxicacaoOf`). Isso não é um erro da ontologia, mas causou um erro na integração desta com o programa.

Na segunda categoria citada estão os erros oriundos da falta de especificação completa sobre os dados com os quais o programa vai operar. Um exemplo é o erro já comentado na Subseção 4.3.2, em que o método da classe não tratava o caso em que não havia valor cadastrado para uma determinada propriedade. Por isso, a consulta retornou um objeto nulo que gerava um erro em tempo de execução se aquela propriedade fosse consultada.

No decorrer deste trabalho foram criados 50 casos de teste por meio dos quais foi possível constatar que o critério particionamento de equivalência é aplicável ao tipo de programa abordado por esta pesquisa, conforme apresentados no Apêndice A.

4.6 Considerações Finais

Neste capítulo foram descritas algumas características de ontologias representadas em OWL e como essas características foram relevantes para o teste de programas. Também foi justificado o uso do critério Particionamento de Equivalência no teste de programas que manipulam conhecimento representado em ontologias e descrita uma abordagem de teste proposta para este tipo de programas. Neste capítulo foram apresentados exemplos de casos de teste criados para demonstrar a aplicabilidade da abordagem proposta. No próximo capítulo são apresentadas as conclusões do trabalho.

Capítulo

5

Conclusões

Testes são fundamentais para a qualidade do software. Existem duas maneiras de se realizar testes: de forma aleatória ou de forma sistemática. A primeira, tem pouca probabilidade de sucesso, enquanto que a segunda, por utilizar técnicas e critérios, pode ser mais eficiente. O objeto de estudo deste trabalho foi o teste de programas que manipulam ontologias expressas em OWL, com o objetivo de sistematizar o processo de teste deste tipo de programa. Embora existam diversos critérios de teste para software convencional, as características próprias da linguagem OWL foram analisadas e isso levou a adoção do critério de teste funcional Particionamento de Equivalência.

A linguagem OWL incorpora o modelo de dados RDF, que representa a informação por meio de declarações (*statements*). A estrutura *sujeito - predicado - objeto* desse modelo foi decisivo para a escolha do critério Particionamento de Equivalência. Assim, tem–se que, para um sujeito S e um dado predicado p, há classes de objetos (o's) válidas e classes inválidas. Considera–se um objeto o válido aquele pertencente ao escopo e que pode ser objeto para o sujeito S dado o predicado p. Os dados inválidos são aqueles que, pertencendo ou não ao escopo, não podem se constituir objetos para o sujeito S dado o predicado p. Uma vez particionados os elementos, casos de teste são criados selecionando–se dados de partições diferentes. Esse procedimento visa evitar testes redundantes e impedir que dados relevantes sejam deixados de lado. Desta forma, embora este trabalho tenha considerado apenas ontologias em OWL, os resultados aplicam–se a qualquer representação construída com base no modelo RDF.

A abordagem proposta neste trabalho define passos a serem seguidos na realização de teste de um programa que manipula ontologias em OWL, usando o critério particionamento de equivalência para seleção dos dados que serão utilizados. Esta abordagem consiste de duas partes: instrumentação dos testes e definição dos casos de teste. A necessidade de instrumentação é levantada no trabalho de El-Korany (2007), que pode ser resumida em manter um repositório de casos de teste para reutilização. *Frameworks* como JUnit servem a esse propósito e vão além, ao permitir a geração semiautomática e execução automática dos testes. O foco principal do trabalho, no entanto, foi a seleção dos dados de teste.

5.0.1 Contribuições

As principais contribuições deste trabalho são:

- Realização de testes em programas que manipulam ontologias utilizando estudos empíricos. Mesmo não sendo o objetivo do trabalho avaliar a eficácia dos testes aplicados, foi observado que a execução dos testes revelou defeitos. Isso comprova que a proposta é válida e confirma a hipótese levantada de que, mesmo para uma ontologia correta, é possível que o algoritmo que realiza as consultas esteja incorreto e leve a respostas erradas.

- Subsídios para identificação dos elementos da ontologia que devem ser testados (tais como classes e propriedades). Para classes devem ser criados casos de teste que verifiquem a relação entre classes e indivíduos e as relações de especialização e generalização. Para propriedades, devem ser criados testes que verifiquem se o programa recupera corretamente os relacionamentos.

- Definição de um conjunto de passos que podem ser seguidos pelo testador durante a fase de testes. Esses passos servem como roteiro para criação e execução dos casos de teste.

- Utilização do critério Particionamento de Equivalência como método viável para a criação de casos de testes para programas que utilizam o *framework* Jena. As consultas realizadas na ontologia têm a característica de sempre retornarem um conjunto de elementos e essa particularidade torna esse critério adequado para criação dos testes.

Assim, a principal contribuição do trabalho é o roteiro a ser seguido na realização dos testes sistemáticos em programas que manipulam ontologias.

O estudo encontrou algumas limitações, tais como a ausência de modelos ou especificações que servissem de referência. O teste funcional deve ser baseado em alguma especificação. Para suprir essa falta, foi utilizado o Protégé, que possui *plugins* que geram alguns modelos automaticamente. No entanto, esses modelos são parciais e não têm notação padronizada.

5.1 Trabalhos Futuros Sugeridos

Durante esta pesquisa, foi utilizado o critério de teste Particionamento de Equivalência e um tipo de ontologia. Algumas possibilidades de continuidade são:

- Exploração de outros critérios de teste, tais como o critério Grafo Causa–Efeito e os critérios baseados em modelos. Os modelos gerados pelas ferramentas de modelagem de ontologias podem ser utilizados para investigação; e
- Analise do teste de ontologias *fuzzy*, uma vez que este trabalho analisou apenas consultas para as quais as respostas são exatas. Ontologias *fuzzy*, por outro lado, lidam com limites imprecisos na representação do conhecimento e introduzem novos desafios para o teste.

Referências

Agarwal, B. B.; Tayal, S. P.; Gupta, M. *Software engineering & testing.* Boston: Jones and Bartlett, 2010.

Alavi, M.; Leidner, D. E. Knowledge management and knowledge management systems: Conceptual foundations and research issues. *MIS Quarterly*, v. 25, n. 1, p. 107–136, 2001.

Allemang, D.; Hendler, J. *Semantic web for the working ontologist.* Burlington: Elsevier Inc., 2008.

Beizer, B. *Software testing techniques.* New York: Van Nostrand Reinhold Co., 1990.

Bertolino, A. Software testing research: Achievements, challenges, dreams. In: *Future of Software Engineering (FOSE 07)*, Washington, DC, USA: IEEE Computer Society, 2007, p. 85–103.

Brank, J.; Grobelnik, M.; Mladenic, D. A survey of ontology evaluation techniques. In: *7th International Multi-conference on Information Society*, 2005, p. 216–224.

Brewster, C.; Alani, H.; Dasmahapatra, S.; Wilks, Y. Data driven ontology evaluation. In: *Proceedings of Int. Conf. on Language Resources and Evaluation*, 2004.

Burnstein, I. *Practical software testing : a process-oriented approach.* Chicago: Springer, 2003.

Chetty, D. *Tomcat 6 developer's guide.* Birmingham: Packt Publishing, 2009.

Colomb, R. M. *Ontology and the sementic web.* Amsterdan: IOS Press, 2007.

Delamaro, M. E.; Maldonado, J. C.; Jino, M. *Introdução ao teste de software.* Rio de Janeiro: Elsevier, 2007.

El-Korany, A. The engineering of expert systems testing process. In: *Proceedings of the 7th Conference on 7th WSEAS international Conference on Applied informatics and Communications*, 2007, p. 251–263.

Eschen, R. *Icefaces 1.8*. Birmingham: Packt Publishing, 2009.

Geary, D.; Horstmann, C. *Core javaserver faces*. Santa Clara: Addison Wesley, 2004.

Gil, A. C. *Como elaborar projetos de pesquisa*. 4 ed. São Paulo: Atlas, 2008.

Gruber, T. R. Toward principles for the design of ontologies used knowledge sharing. *International Journal of Human-Computer Studies*, v. 43, n. 5/6, p. 907–928, 1995.

Gruninger, M.; Fox, M. S. Methodology for design and evaluation of ontologies. In: *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI98)*, 1995, p. 61–72.

Hebeler, J.; Fisher, M.; Blace, R.; Perez-Lopez, A. *Semantic web programming*. Wiley, 2009.

Hepp, M. *Ontology management semantic web, semantic web services, and business applications* Springer, p. 17–22, 2008.

Howden, W. E. Functional program testing. *IEEE Transactions on Software Engeneering*, v. 6, n. 3, p. 162–169, 1980.

Knublauch, H.; Dameron, O.; Musen, M. A. Weaving the biomedical semantic web with the protégé owl plugin. In: *In International Workshop on Formal Biomedical Knowledge Representation*, 2004, p. 39–47.

Linhalis, F. *Mapeamento semântico entre unl e componentes de software para execução de requisições imperativas em linguagem natural*. Tese de Doutoramento, Instituto de Ciências Matemáticas e de Computação - ICMC, Universidade de São Paulo - USP, São Carlos - São Paulo, 2007.

Lozano-Tello, A.; Gómez-Pérez, A. Ontometric: A method to choose the appropriate ontology. *Journal of Database Management*, v. 15, n. 2, p. 1–18, 2004.

Maedche, A.; Staab, S. Measuring similarity between ontologies. In: *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management*, 2002, p. 251–263.

MARTIMIANO, L. A. F. *Sobre a estruturação de informação em sistemas de segurança computacional: o uso de ontologia.* Tese de Doutoramento, Instituto de Ciências Matemáticas e de Computação - ICMC, Universidade de São Paulo - USP, São Carlos - São Paulo, 2006.

MYERS, G. J. *Art of software testing.* 2 ed. New York: John Wiley & Sons, Inc., 2004.

NOY, N. F.; MCGUINNESS, D. L. Ontology Development 101: A Guide to Creating Your First Ontology. `protege.stanford.edu/publications/ontologydevelopment/ontology101.pdf/`, [Online; Acesso em: 08/03/2011], 2001.

OLIVEIRA, R. B. *Framework functest: Aplicando padrões de software na automação de testes funcionais.* Dissertação de Mestrado, Universidade de Fortaleza, Fortaleza, 2007.

PORZEL, R.; MALAKA, R. A task-based approach for ontology evaluation. In: *Proceedings of Workshop on Ontology Learning and Population (ECAI 2004)*, 2004, p. 161–166.

PREECE, A. Evaluating verification and validation methods in knowledge engineering. *MICRO-LEVEL KNOWLEDGE MANAGEMENT*, p. 123–145, 2001.

PRESSMAN, R. S. *Engenharia de software.* 6 ed. Porto Alegre: McGraw-Hill, 2006.

PROTÉGÉ What is protégé? Disponível em: <http://protege.stanford.edu/overview>. Acesso em: <15/05/2011>. [*On-line*].

SANTOS, C. R. J.; VALE, Z. A.; MARQUES, A. Verification & validation of power systems control centres kbs. In: *Proceedings of IASTED International Conference, Artificial Intelligence and Applications*, 2001, p. 39–52.

SCHREIBER, G.; AKKERMANS, H.; ANJEWIERDEN, A.; HOOG, R.; SHADBOLT, N.; VELDE, W. V.; WIELINGA, B. *Knowledge engineering and management. The CommonKADS methodology.* MIT Press, 455 p., 2000.

SOMMERVILLE, I. *Engenharia de software.* 8 ed. São Paulo: Addison-Wesley, 2007.

STUDER, R.; BENJAMINS, V. R.; FENSEL, D. Knowledge engineering: principles and methods. *Data and knowledge engineering*, v. 25, p. 161–197, 1998.

W3C. OWL Web Ontology Language Reference. `http://www.w3.org/TR/owl-ref/`, [Online; Acesso em: 08/03/2011], 2004.

von Wangenheim, C. G.; von Wangenheim, A. *Raciocínio baseado em casos.* Manole, 2003.

Zhu, H.; Hall, P. A. V.; May, J. H. R. Software unit test coverage and adequacy. *ACM Comput. Surv.*, v. 29, p. 366–427, 1997.

Apêndice

A

Ontologia OntoTox

A.1 A OntoTox

A construção de uma ontologia é indicada para manipular conhecimento específico de um domínio e altamente dependente de especialistas. A OntoTox[1] é uma ontologia que se propõe a organizar conhecimento relativo a agrotóxicos, intoxicações e doenças relacionadas para suporte ao profissional da saúde. Pouco se sabe ou é divulgado sobre as doenças e intoxicações causadas pelo uso indiscriminado de agrotóxicos. Essas doenças são, geralmente, câncer e degeneração do sistemas nervoso. Há também as manifestações imediatas de intoxicação, como vômitos e cefaléia. O tipo de intoxicação aguda (de curto prazo) causada por agrotóxico é diferente para cada produto com o qual o paciente teve contato. As doenças causadas pelas intoxicações também variam de acordo com o grupo ao qual pertence o agrotóxico.

A OntoTox possui, na versão utilizada, 39 classes, apresentadas na Figura A.1. Na figura pode-se observar, além das classes, os relacionamentos de generalização/especialização. Por exemplo, a classe `Agrotoxico` tem quatro subclasses: `Herbicidas`, `Inseticidas`, `Fungicidas` e `OutrosGrupos`.

A ontologia foi desenvolvida utilizando o ambiente Protégé. Este ambiente de desenvolvimento é uma plataforma que permite modelar ontologias por meio de frames ou por meio do editor OWL. Protégé é uma das ferramentas mais utilizadas na construção de

[1] A ontologia está sendo desenvolvida pela aluna de mestrado em Ciência da Computação Sandra Xavier de Macedo.

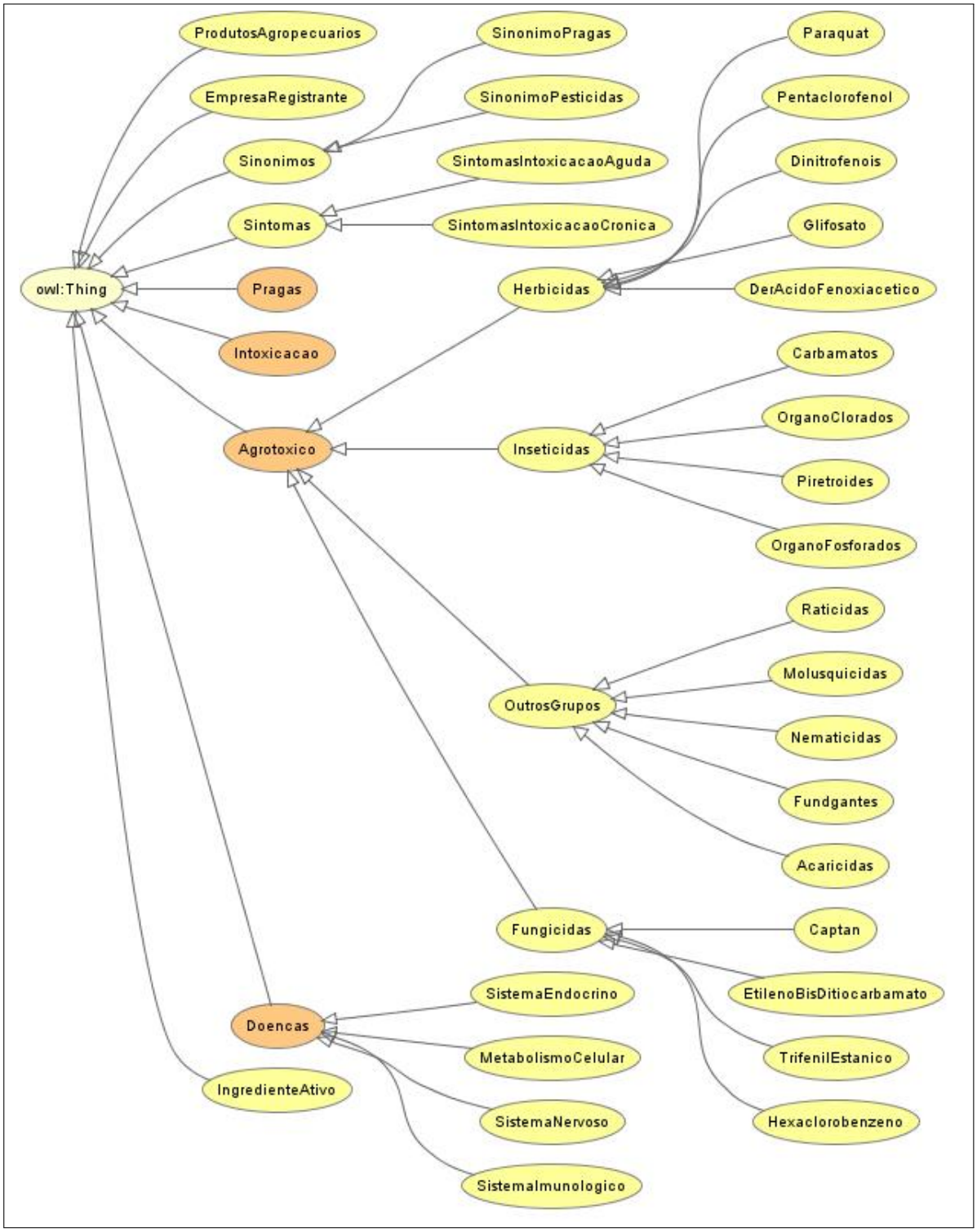

Figura A.1: Classes da OntoTox.

ontologias na área de saúde (Knublauch et al., 2004). O editor possibilita descrever classes, propriedades e instâncias. O software é livre e extensível por meio de *plugins* (Protégé, 2011). Na Figura A.2 é apresentado o diagrama de visão aninhada (*nested view*) gerado pelo plug-in Jambalaya. Este diagrama possibilita visualizar os relacionamentos entre as classes da ontologia.

A.2 Questões de Competência

As questões de competência são um método que definem quais perguntas a ontologia deve ser capaz de responder (Gruninger e Fox, 1995). Segundo Martimiano (2006), essas questões guiam a escolha dos conceitos representados na ontologia. Depois de construída, na fase de avaliação, as questões auxiliam a verificar a qualidade da ontologia. As principais questões de competência definidas para a OntoTox foram:

1. Quais as doenças relacionadas ao metabolismo celular?
2. Quais as doenças relacionadas ao Sistema Endócrino?
3. Quais as doenças relacionadas ao Sistema Nervoso?
4. Quais são as doenças cadastradas relacionadas ao uso de agrotóxicos?
5. Quais os sintomas de intoxicação por um determinado agrotóxico informado?
6. Quais os sintomas de determinada doença?
7. Quais produtos agrotóxicos têm um determinado ingrediente ativo informado?
8. Qual a empresa que comercializa um determinado agrotóxico informado?
9. Quais agrotóxicos são comercializados por uma determinada empresa?
10. Quais pragas são combatidas por um determinado agrotóxico?
11. Quais produtos agrícolas têm aplicação de determinado agrotóxico?
12. Quais pragas podem atacar determinado produto agrícola?
13. Quais agrotóxicos estão relacionados a um determinado sintoma de intoxicação informado? Ex.: O sintoma “vômitos” está relacionado à intoxicação por quais agrotóxicos?

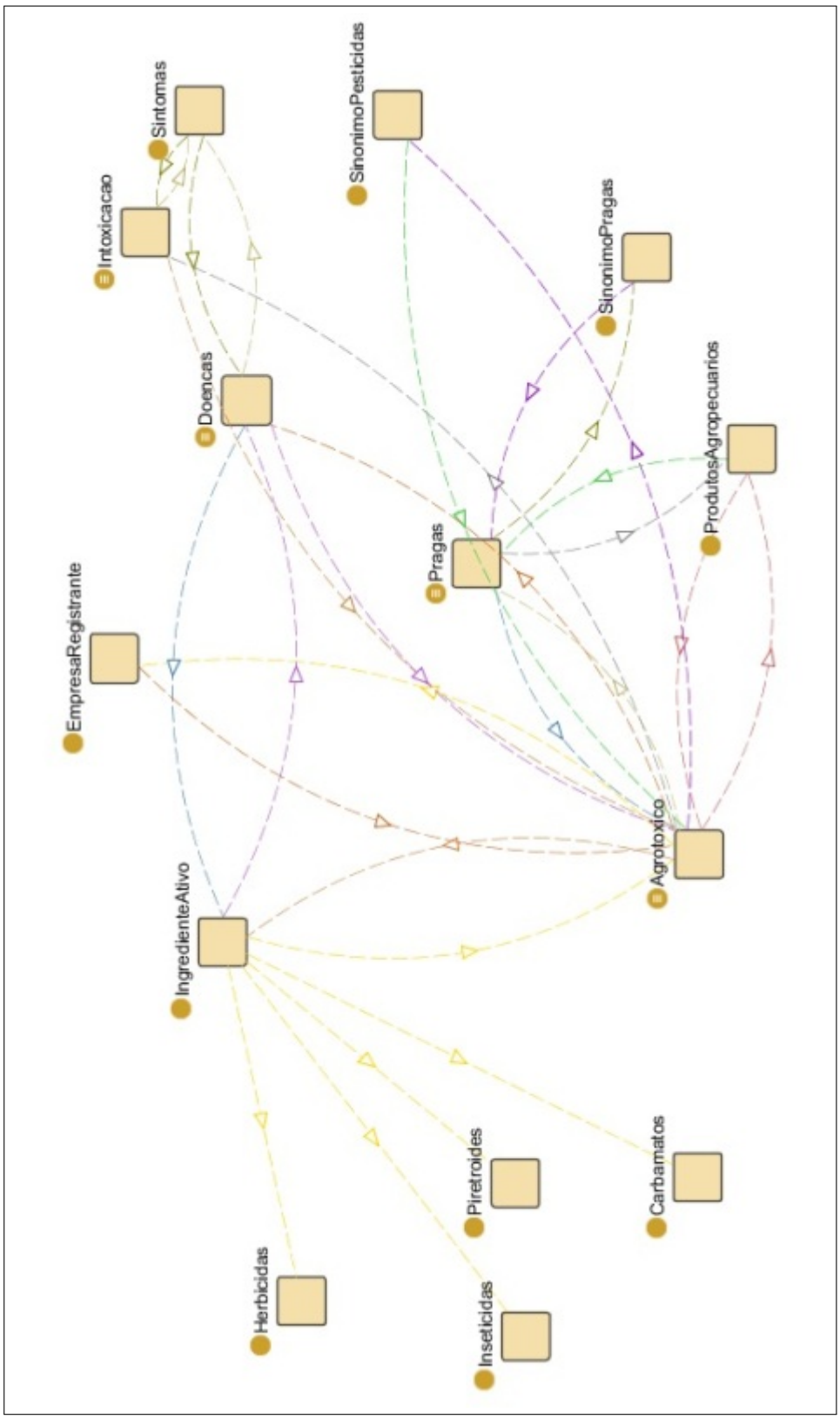

Figura A.2: Diagrama gerado pelo *plugin* Jambalaya.

14. Dados três sintomas, qual o provável grupo de agrotóxico causou a intoxicação?

Apêndice

B

Casos de teste aplicados

Neste apêndice são listados alguns casos de teste aplicados que foram criados utilizando JUnit.

B.1 Casos de teste para classe `Agrotoxico`

```
@Test
    public void testGetIngredienteAtivo(){
        System.out.println("getIngredienteAtivo");
        Agrotoxico agro = new Agrotoxico();
        Ontotox instance = new Ontotox();
        OntModel ontologia = instance.getOntologia();
        String tdNS = "http://www.owl-ontologies.com/Ontology1282915023.owl#";
        Individual i =
            ontologia.getIndividual("http://www.owl-ontologies.com/Ontology1282915023.owl#Piretroides_46");
        ArrayList<String> lista = agro.getIngredienteAtivo(i);
        Iterator it = lista.iterator();
        while (it.hasNext()){
            System.out.println("Ingrediente: "+it.next());
        }
    }

@Test
    public void testGetFormula(){
        Agrotoxico instance = new Agrotoxico();
        String expectedResult = "";
        expectedResult = "C21H20Cl2O 3^^http://www.w3.org/2001/XMLSchema#string";
        String result = instance.getFormula("Piretroides_1");
        assertEquals(expectedResult, result);
    }
```

```
@Test
    public void testGetClasseToxicologica(){
        System.out.println("getClasseToxicologica");
        Agrotoxico agro = new Agrotoxico();
        Ontotox instance = new Ontotox();
        OntModel ontologia = instance.getOntologia();
        String tdNS = "http://www.owl-ontologies.com/Ontology1282915023.owl#";
        Individual i =
            ontologia.getIndividual("http://www.owl-ontologies.com/Ontology1282915023.owl#Piretroides_46");
        String classeTox = agro.getClasseToxicologica(i);
        assertEquals("III - Azul (Medianamente Tóxico)", classeTox);
    }


@Test
    public void testObterGrupos(){
        Agrotoxico instance = new Agrotoxico();
        ArrayList<String> expectedResult = new ArrayList();
        expectedResult.add("Fungicidas");
        expectedResult.add("Herbicidas");
        expectedResult.add("Inseticidas");
        expectedResult.add("OutrosGrupos");
        ArrayList<String> result = instance.obterGrupos();
        assertTrue(result.containsAll(expectedResult));
    }


@Test
    public void testObterDatatypeProperty(){
        Agrotoxico instance = new Agrotoxico();
        String result = "";
        String expectedResult = "";
//        testanto individuo que possui formula
        expectedResult = "C21H20Cl2O 3^^http://www.w3.org/2001/XMLSchema#string";
        result = instance.obterDatatypeProperty("Piretroides_1", "Formula");
        assertEquals(expectedResult, result);
//        //testanto individuo que não possui formula
        expectedResult = "não definido";
        result = instance.obterDatatypeProperty("Carbaril", "Formula");
        assertEquals(expectedResult, result);
    }


@Test
    public void testListarNomesAgr(){
        ArrayList<String> agrotoxicos = new ArrayList();
        Agrotoxico instance = new Agrotoxico();
        agrotoxicos = instance.listarNomesAgrotoxicos();
        Iterator i = agrotoxicos.iterator();
        while (i.hasNext()){
            System.out.println("Nome: " + i.next().toString());
        }
        System.out.println("Quantidade: "+agrotoxicos.size());
    }
```

```
@Test
    public void testListarAgrotoxicos2() {
        System.out.println("listar agrotoxicos");
        Agrotoxico instance = new Agrotoxico();
        HashMap <String, String> hm = instance.listarAgrotoxicos();
        Set<String> chaves = hm.keySet();
        for (Iterator<String> iterator = chaves.iterator(); iterator.hasNext();)
        {
            String chave = iterator.next();
            System.out.println("Nome: "+ hm.get(chave));
        }
        System.out.println("\n\n\nQuantidade: "+ hm.size());
    }


 @Test
    public void testListarSinonimos() {
        System.out.println("listar propriedades");
        String nome = "Aldrin";
        Agrotoxico instance = new Agrotoxico();
        instance.isSinonimo(nome);
        ArrayList<String> result = instance.listarPropriedades(nome);
        Iterator r = result.iterator();
        System.out.println("Sinônimos de: "+ nome);
        while (r.hasNext()){
            System.out.println("Sinônimo: " + r.next().toString());
        }
    }


@Test
    public void testListarSintomasIntoxicacaoPor() {
        Agrotoxico instance = new Agrotoxico();
        HashMap<String, String> retorno = instance.listarSintomasIntoxicacaoPor("OrganoFosforados_57");
        Set<String> chaves = retorno.keySet();
        for (Iterator<String> iterator = chaves.iterator(); iterator.hasNext();)
        {
            String chave = iterator.next();
            System.out.println(retorno.get(chave));
        }
    }


@Test
    public void testGetEmpresaRegistrante(){
        Agrotoxico instance = new Agrotoxico();
        String expectedResult = "BASF";
        String result = instance.getEmpresaRegistrante("Piretroides_38");
        assertEquals(expectedResult, result);
        result = instance.getEmpresaRegistrante("Piretroides_27");
        assertFalse(expectedResult.equalsIgnoreCase(result));
    }
```

B.2 Casos de teste para outras classes

```
@Test
    public void testListarGrupos(){
        Doenca instance = new Doenca();
        int expectedResult = 4;
        int result = instance.listarGrupos().size();
        assertEquals(result, expectedResult);
    }


@Test
    public void testListarGrupos2(){
        Doenca instance = new Doenca();
        ArrayList<String> result = instance.listarGrupos();
        boolean expectedResult = result.contains("MetabolismoCelular"); //existe
        assertTrue(expectedResult);
        expectedResult = result.contains("SistemaRespiratorio");   //não existe
        assertFalse(expectedResult);
        Iterator it = result.iterator();
        while (it.hasNext()){
            System.out.println("Grupo: "+it.next().toString());
        }
    }

@Test
    public void testCarregarOntotox() {
        System.out.println("carregar Ontotox");
        Ontotox instance = new Ontotox();
        OntModel result = instance.carregarOntotox();
        assertNotNull( result);
    }


@Test
    public void testListarSubClasses(){
        System.out.println("listar sub classes");
        Ontotox instance = new Ontotox();
        ArrayList<String> lista = instance.listarSubClasses("Agrotoxico");
        boolean expectedResult = true;
        expectedResult = lista.contains("Fungicidas");
        Assert.assertTrue( expectedResult);
        expectedResult = lista.contains("SistemaNervoso");
        Assert.assertFalse( expectedResult);
    }
```

```
@Test
    public void testListarDoencasPorGrupo() {
        String nomeGrupo = "MetabolismoCelular";
        Iterator r;
        Doenca instance = new Doenca();

        System.out.println("Doenças do " + nomeGrupo);
        ArrayList<String> result = instance.listarDoencasPorGrupo(nomeGrupo);
        r = result.iterator();
        while (r.hasNext()){
            System.out.println("Doenca: " + r.next().toString());
        }

        nomeGrupo = "SistemaEndócrino";
        System.out.println("\n\nDoenças do " + nomeGrupo);
        result = instance.listarDoencasPorGrupo(nomeGrupo);
        r = result.iterator();
        while (r.hasNext()){
            System.out.println("Doenca: " + r.next().toString());
        }

        nomeGrupo = "SistemaImunológico";
        System.out.println("\n\nDoenças do " + nomeGrupo);
        result = instance.listarDoencasPorGrupo(nomeGrupo);
        r = result.iterator();
        while (r.hasNext()){
            System.out.println("Doenca: " + r.next().toString());
        }

        nomeGrupo = "SistemaNervoso";
        System.out.println("\n\nDoenças do " + nomeGrupo);
        result = instance.listarDoencasPorGrupo(nomeGrupo);
        r = result.iterator();
        while (r.hasNext()){
            System.out.println("Doenca: " + r.next().toString());
        }
    }
```