MAURO MORAIS DE MIRANDA

EXTENSÃO DO META-MODELO RAS PARA ADIÇÃO DE SUPORTE DE SISTEMAS MULTI-AGENTES A REDES DE COMPARTILHAMENTO DE COMPONENTES DE SOFTWARE

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2008

MAURO MORAIS DE MIRANDA

EXTENSÃO DO META-MODELO RAS PARA ADIÇÃO DE SUPORTE DE SISTEMAS MULTI-AGENTES A REDES DE COMPARTILHAMENTO DE COMPONENTES DE SOFTWARE.

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADO: 28 de fevereiro de 2008

Prof. Vladimir Oliveira Di Iorio
(Co-orientador)

Prof. José Luis Braga
(Co-orientador)

Prof. Íris Fabiana de Barcelos Tronto

Profª Mauro Nacif Rocha

Prof. Alcione de Paiva Oliveira
(Orientador)

”Eis que estou à porta, e bato: se alguém ouvir a minha voz e me abrir a porta, entrarei em sua casa e cearemos, Eu com ele e ele comigo.”

Ap 3:20

Agradecimentos

Primeiramente agradeço a Deus por ter me dado força, saúde e por ter colocado em meu caminho pessoas que me encorajaram a seguir em frente e sempre tiveram a certeza da conclusão deste trabalho. Muito Obrigado!!

Agradeço a minha esposa Auxiliadora pelo apoio, por assumir parte do meu papel de pai nas constantes ausências junto aos filhos, por estar sempre presente nos momentos mais difíceis e que não foram poucos, pela ajuda nas traduções e revisões da escrita. Que sempre incentivou, apoio e acreditou que conseguiria chega ao fim do túnel. Sempre companheira. Muito Obrigado!!

Agradeço aos meus filhos, Maria Claudia, Anna Paula e Mauro Junior, que sempre me incentivaram, por cobrarem a conclusão deste trabalho, sempre incentivando e tendo a paciência pelas ausências, mesmo quando estava fisicamente próximos deles. Muito Obrigado!!

Agradeço a minha mãe que sempre me incentivou, rezou, fez promessa, desculpas pela minha ausência. Aos meus irmãos e sobrinhos sempre me dando apoio e torcendo. Muito Obrigado!!

Agradeço especialmente aos amigos do Unileste, Vinícius, Débora e Deisy, incentivadores, amigos, companheiros de estrada, sempre a disposição e me cobrando para seguir em frente que tudo iria acabar bem. Sem vocês teria sido difícil chegar aqui. Muito obrigado!!

Agradeço aos meus colegas de equipe na PMI, que sempre me apoiaram, resolveram os problemas do dia a dia da equipe e que foram sempre fiéis companheiros na minha ausência. Tenho muito respeito e admiração por todos. Muito obrigado!!

Agradeço à Prefeitura Municipal de Ipatinga, na pessoa do secretário do Serviço Municipal de Dados-DATASERV, Sr. Ilton Câmara, que me liberou a presença às

atividades profissionais para as constantes viagens a Viçosa para a realização das aulas e reuniões. Muito Obrigado!!

Agradeço à UNILESTEMG pelo apoio financeiro, com a concessão de bolsa de estudo e redução da jornada de trabalho para a participação no mestrado.

Ao professor Antonio Machado Filho, coordenador do curso Bacharelado em Sistemas de Informação do UNILESTEMG, pelo apoio, incentivo, colaboração e sempre criando condições para seguirmos a caminhada. Muito Obrigado!!

Agradeço à Universidade Federal de Viçosa-UFV, que abriu as portas para que uma pessoa não ligada a instituição conseguisse ingressar no programa de mestrado. À coordenação do programa de mestrado e a todos os professores que sempre procuraram propiciar um ambiente acolhedor e favorável ao desenvolvimento do aprender. Muito Obrigado!!

Agradeço imensamente ao meu professor orientador Alcione de Paiva Oliveira, que sempre foi um incentivador, sempre a disposição para o atendimento e acreditando que poderia chegar a conclusão do programa. Muito Obrigado!!

Novamente, agradeço a Deus e a todos vocês, sempre serão muito importantes em minha vida e da minha família. Abraços a todos..... Muito Obrigado!!

Biografia

Mauro Morais de Miranda, filho de Felício Miranda Neto e Maria Perpetua de Araújo Miranda, brasileiro, nascido em 23 de junho de 1961 no município de Jaguaraçu, Estado de Minas Gerais.

No ano de 1981, após concluir o curso técnico em Metalurgia na cidade de Ipatinga, ingressou no curso de graduação de "Engenharia Industrial Elétrica" na Pontifícia Universidade Católica de Minas Gerais campus Coronel Fabriciano, onde se graduou no ano de 1989. Em 1985 ingressou na Prefeitura Municipal de Ipatinga como Programador de Computador passando em 1989 a atuar como Analista de Sistema. Em 1992 concluiu o curso de pós-graduação em Análise de Sistemas pela Faculdade Estácio de Sá-RJ. Em 1993 começou a trabalhar como professor de informática em escola de curso técnico na cidade de Coronel Fabriciano. Em 2002 concluiu a pós-graduação em Métodos e Ferramentas Computacionais pela Universidade Federal de Minas Gerais. Em 2003, após concluir a pós-graduação, começou a lecionar para o curso superior em um Centro Universitário da cidade de Coronel Fabriciano. Em 2005 foi aprovado na seleção do mestrado do Departamento de Informática - DPI, onde cursou o mestrado em Ciência da Computação na Universidade Federal de Viçosa - UFV, defendendo a sua dissertação em 28 de fevereiro de 2008.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Abreviaturas e Siglas

API	- *Application Programming Interface*
CCM	- *CORBA Component Model*
COM	- *Component Object Model*
CORBA	- *Common Object Request Broker Architecture*
CPF	- *Cadastro de Pessoa Física*
DAML	- *DARPA Agent Markup Language*
DBC	- *Desenvolvimento de Software Baseado em Componentes*
DCOM	- *Distributed component object model*
EJBs	- *Enterprise JavaBeans*
FIPA	- *Foundation for Intelligent Physical Agents*
FIPA-ACL	- *FIPA - Agent Communication Language*
HTML	- *Hyper Text Markup Language*
IDE	- *Integrated Development Environment*
J2EE	- *Java Enterprise Edition*
JADE	- *Java Agent Development Framework*
Java EE	- *Java Enterprise Edition*
JCP	- *Java Community Process*
JOE	- *Java Ontology Editor*
KIF	- *Knowledge Interchange Format*
OIL	- *Ontology Interchange Language*
OMG	- *Object Management Group*
Ontolingua	- *Linguagem de Compartilhamento de Ontologias*

ORB	- *Object Request Brokers*
OWL	- *Web Ontology Language*
RAS	- *Reusable Asset Specification*
RCCS	- *Rede de Compartilhamento de Componentes de Software*
RDF	- *Resource Description Framework*
RMI	- *Remote Method Invocation*
SEI	- *Software Engineering Institute*
SGBD	- *Sistema Gerenciador de Banco de Dados*
SGML	- *Standard Generalized Markup Language*
SQL	- *Structured Query Language*
UML	- *Unified Modeling Language*
URL	- *Uniform Resource Locator*
W3C	- *Word Wide Web Consortium*
WEB	- *Word Wide Web*
XML	- *Extended Markup Language*

Resumo

MIRANDA, Mauro Morais de, M.Sc.,Universidade Federal de Viçosa, Fevereiro de 2008. **Extensão do Meta-Modelo RAS para Adição de Suporte de Sistemas Multi-Agentes a Redes de Compartilhamento de Componentes de Software**. Orientador: Alcione de Paiva Oliveira. Co-Orientadores: Vladimir Oliveira Di Iorio e José Luis Braga.

O objetivo principal da utilização de componentes de software é o incremento do reuso, o que leva a um aumento da produtividade por usar componentes pré-existentes, com maior confiabilidade, maior facilidade de gerenciamento e a padronização do desenvolvimento. O processo de desenvolvimento de software baseado em componentes modifica sensivelmente o processo tradicional de desenvolvimento, introduzindo novas características. O foco passa a ser a integração de componentes, em que questões arquiteturais devem ser consideradas de forma mais rigorosa, requerendo um esforço em requisitos, testes e integração de componentes. O Object Management Group (OMG) entendeu que o primeiro passo a ser tomado seria definir a estrutura de um componente de software, não em termos tecnológicos, mas sim sob a ótica de meta-informações que ele deve possuir e como ela deve estar estruturada para que aquele componente seja facilmente utilizado, sugerindo o conjunto de meta-informações Reusable Asset Specification(RAS). O objetivo deste projeto é desenvolver uma ontologia sobre componentes de forma a estender o conjunto das meta-informações RAS, criando um repositório de componentes que permita especificar e desenvolver um sistema multiagente com mecanismos e padrões definidos onde seus agentes possam: gerenciar, catalogar, organizar e pesquisar componentes de software de forma a promover o reuso efetivo e aumentar a eficiência e produtividade do desenvolvimento de software.

Abstract

MIRANDA, Mauro Morais de Miranda, M.Sc.,Universidade Federal de Viçosa, Fevereiro of 2008. **Extensão do meta-modelo RAS para adição de suporte de Sistemas Multi-Agentes à Redes de Compartilhamento de Componentes de Software.** Advisor: Alcione de Paiva Oliveira. Co-Advisors: Vladimir Oliveira Di Iorio and José Luis Braga.

The main objective of the use of software components is the increase in reuse, which leads to increased productivity by using pre-existing components, with higher reliability, greater ease of management and the standardization of development. The process of developing software based components significantly modifies the traditional process of development, introducing new features. The focus is the integration of components, where architectural issues are considered in more rigorous ways, requiring an effort in requirements, testing and integration of components. The Object Management Group (OMG) felt that the first step to be taken would be to define the structure of a software component, not in technological terms, but from the perspective of meta-information it should have and how it should be structured so that component is easily used, suggesting the set of meta-information Reusable Asset Specification (RAS). The goal of this project is to develop a components ontology in order to extend the range of meta-information RAS, creating a repository of components enabling one to specify and develop a system with multiagente mechanisms and patterns, where its agents can manage, catalog, organize and search components of software in order to promote the effective reuse and increase the efficiency and productivity of software development.

Capítulo 1

Introdução

Segundo SAMETINGER (1997), componentes de software são artefatos autocontidos, claramente identificáveis, que descrevem ou realizam uma função específica e têm interfaces claras, documentação apropriada e um bom grau de reutilização. Cada componente pode ser reutilizado diversas vezes, em diferentes aplicações, e substituído por outros componentes com as mesmas especificações de interface e arquitetura.

Vários são os benefícios percebidos pela reutilização de software. AGRESTI and MCGARRY (1998) salientam os objetivos relacionados com o aumento de produtividade (porque usa componentes já existentes), aumento de confiabilidade (uma vez que usa componentes já testados), aumento de consistência (porque usa os mesmos componentes em vários lugares) e maior facilidade de gerenciamento (visto que usa componentes já compreendidos pelos desenvolvedores). AGRESTI and MCGARRY (1998) salientam ainda os benefícios obtidos pela padronização.

Este novo processo de Desenvolvimento de Software Baseado em Componentes (DBC), modifica sensivelmente o processo de desenvolvimento tradicional, introduzindo novas características. O foco passa a ser a integração de componentes, onde questões arquiteturais devem ser considerados de forma mais rigorosa, requerendo um esforço em: requisitos, testes e integração de componentes.

Nesta abordagem, componentes de software são projetados para o reuso. Para facilitar o reutilização, pesquisas tem sido realizadas para desenvolver repositórios de componentes, que possam auxiliar na busca de componentes de software com a

finalidade de aumentar a reusabilidade de componentes e a produtividade de desenvolvedores de software.

A especificação do Reusable Asset Specification (RAS) (KOZACZYNSKI et al., 2003),surgiu de um esforço da indústria de software, apoiada pelo Object Management Group (OMG), com a finalidade de apoiar e estimular o desenvolvimento de software baseado em componentes. O OMG entendeu que o primeiro passo a ser tomado seria definir a estrutura de um componente de software, não em termos tecnológicos mas sim sob a ótica de meta-informações que ele deve possuir e como ela deve estar estruturada para que aquele componente seja facilmente utilizado.

Os metadados RAS devem suportar a definição de um componente de software como um conjunto de meta-informações que ele deve possuir e como ela deve estar estruturada para que aquele componente seja facilmente utilizado.

Um componente precisa ser alcançável através de mecanismos de busca e recuperação, que vem sendo um dos grandes problema do DBC. Em alguns casos são utilizadas linguagens de consulta muito restritas com base no casamento de padrões, comparando palavras-chaves (da busca) com *strings* contidas na descrição do componente.

Nos últimos anos, pesquisas apresentam soluções com o uso de conteúdo semântico associado aos componentes descritos através de ontologia que descreve conceitos e relacionamento entre os mesmos. SUGUMARAN and STOREY (2003) propõem uma solução que se baseia no uso de ontologia e na criação de um conteúdo semântico para os componentes, possibilitando uma busca mais inteligente, pois facilita a criação de ferramentas que permitam buscas em linguagens naturais. Pode-se usar também uma máquina de inferência para saber se os componentes satisfazem a algumas premissas.

1.1 Objetivos de Trabalho

O conjunto das metas-informações contido no repositório utilizado em uma Rede de Compartilhamento de Componentes de Software (RCCS) deve conter informações suficientes para orientar os agentes de um sistema multi-agente na pesquisa de com-

ponentes.

O projeto de pesquisa visa desenvolver uma ontologia sobre componentes, que permita a extensão do conjunto das meta-informações do modelo RAS que permita definir um repositório de componentes, que dê suporte a definição de um componente de software como um conjunto de meta-informações que ele deve possuir de forma estruturada.

Acredita-se que esta extensão poderá facilitar a especificação e desenvolvimento de um sistema multiagente com mecanismos e padrões definidos, onde os seus agentes possam: gerenciar, catalogar, organizar e pesquisar os componentes de software de forma a promover o reuso efetivo e aumentar a eficiência e produtividade do desenvolvimento de software baseado na utilização de componentes.

O conjunto das metas-informações contido no repositório RAS não contempla informações suficientes para orientar os agentes na pesquisa de componentes. O objetivo passa a ser a elaboração de uma ontologia sobre componentes que permita uma extensão do meta-modelo RAS, capaz de fornecer informações suficientes para ajudar um sistema multiagente. Assim pode dar suporte ao desenvolvimento de sistemas baseado no reuso de componentes de software.

Neste contexto, foi realizado um estudo da especificação RAS da OMG e desenvolvida uma ontologia de componentes, que permita definir uma extensão do meta-modelo de informações do RAS, que auxilie a implementação de um sistema multiagente baseado no *framework* JADE que possa auxiliar na busca e recuperação de componentes de software. Com isto, pode levar a um aumento da reusabilidade de componentes e a produtividade de desenvolvedores de software.

Para a realização do trabalho, foi necessário alcançar os seguintes objetivos específicos:

- Estudar e estender a estrutura das meta-informações do RAS para otimizar a busca de componentes na RCCS.
- Produzir uma ontologia de componentes.
- Desenvolver repositórios de componentes sob a ótica de meta-informações que

ele deve possuir e não só como depósito de componentes. De forma a dar suporte ao desenvolvimento de um sistema multiagente que permita gerenciar, catalogar, organizar e pesquisar os componentes de software.

- Estudar e utilizar o *framework* JADE para implementação do protótipo do sistema multiagente.
- Disponibilizar os componentes em uma Rede de Compartilhamento de Componentes de Software.
- Aumentar a comunicação entre os produtores e os consumidores de componentes.
- Aumentar a reusabilidade de componentes de software.

1.2 Organização deste Documento

Este documento está organizado nos seguintes capítulos: no Capítulo 2, é feita uma revisão bibliográfica apresentando as principais característcas de Componentes de Software, Desenvolvimento Baseado em Componentes, especificação RAS, Ontologias de Desenvolvimento Baseado em Componentes e principais trabalhos relacionados; no Capítulo 3, é descrita a proposta da extensão do modelo RAS com a definição de uma ontologia para suporte a um Sistema multiagente para a pesquisa de componentes em uma rede de compartilhamento de componentes de software; eno Capítulo 4, são descritas as conclusões e indicadas propostas de trabalhos futuros desta dissertação.

Capítulo 2

Revisão da Literatura

2.1 Componentes

O conceito exato de componente em DBC ainda não é um tópico fechado. Cada grupo de pesquisa caracteriza, da maneira mais adequada ao seu contexto, o que seria um componente, assim, é possível encontrar na literatura, diversas definições de componentes.

SAMETINGER (1997) define componentes de software como artefatos **autocontidos**, **claramente identificáveis**, que descrevem ou realizam uma **função específica** e têm **interfaces claras**, **documentação** apropriada e um grau de **reutilização** definido. Cada componente pode ser reutilizado diversas vezes, em diferentes aplicações, e substituído por outros componentes com as mesmas especificações de interface e arquitetura. Componentes podem ser vistos como alguma parte do sistema de software que é identificável e reutilizável, ou como o estado seguinte de abstração depois de funções, módulos e classes. Portanto os componentes:

- São tratados como **artefatos**: por poderem assumir diferentes formas como, por exemplo, código, documentação, especificação de funcionalidades, entre outros produtos de todo o seu ciclo de vida.
- São **autocontidos**: significa que a função que o componente desempenha deve ser realizada por ele de forma completa e sem dependência de outros compo-

nentes. Por exemplo, uma função pode ser considerada autocontida se ela pode ser utilizada sem a necessidade de qualquer outra função, caso ela necessite de outra função para alcançar seus objetivos, então todo o conjunto de funções deve ser visto como um único componente de software.

- Devem ser **claramente identificáveis**: de forma que possa ser facilmente encontrado, seja qual for sua localização, ou seja, deve estar contido em um único módulo ou pacote. Em um sistema distribuído, é importante também que o componente seja identificado de maneira única sem ambigüidades, pois, atualmente, distribuição e interoperabilidade são características desejadas pelas aplicações.

- Têm uma **funcionalidade clara e específica**: de forma que possa ser facilmente identificável em relação a sua utilidade em um dado contexto. Por exemplo, um componente de validação de CPF para um sistema de controle financeiro.

- Possuir **interfaces**: que determinam como se dá essa dependência do componente em relação aos demais e ao ambiente que o cerca, determinam como esse componente pode ser reutilizado e interconectado com outros componentes, ocultando os detalhes que não são necessários para o reuso. As interfaces definem uma operação ou um conjunto de operações que estão disponíveis para um componente. Uma interface é um conjunto de assinaturas de operações que podem ser invocadas por um cliente.

- Ter uma boa **documentação** é também indispensável para a reutilização dentro de um determinado contexto. A documentação deve ser suficiente para que se possa recuperar um componente, facilitar o seu entendimento, a sua avaliação e fazer adaptações de modo a facilitar a integração aos demais componentes.

- Ter um **bom grau de reutilização**: Componentes devem ser mantidos em repositóriios de modo a preservar a reutilização sistemática, sendo que o grau de reutilização compreende diferentes informações tais como, por exemplo, quem é o proprietário, quem deve ser contatado em caso de problemas, qual é a situação de qualidade e quantas vezes e onde o mesmo foi reutilizado.

D'SOUZA and WILLS (1999) apresentam duas definições, uma para componentes de forma geral e outra para componentes na forma de código:

- "Um componente (em geral) é um pacote coerente de artefatos de software que pode ser independentemente desenvolvido e distribuído como uma unidade, e que pode ser composto, sem alterações, com outros componentes para construir algo maior. Componente é um pacote que pode conter vários artefatos, por exemplo: código executável, código fonte, projetos, especificações, teste, e documentação. E tem a capacidade de ser utilizado para a construção de algo maior, isto é, participar de composições".

- "Um componente (em código) é um pacote coerente de implementação que pode ser desenvolvido e distribuído independentemente, provê interfaces explícitas e bem especificadas, define interfaces que ele precisa de outros componentes, e pode ser combinado com outros componentes pela configuração de suas propriedades, sem a necessidade de modificação. Neste, o autor enfatiza aspectos relacionados à utilização do componente, à dependência contextual, isto é, que outros componentes são necessários (interfaces que o componente necessita)".

2.1.1 Classificação de Componentes quanto à Visibilidade

A Figura 2.1 mostra a classificação de componentes quanto à visibilidade do código fonte para os desenvolvedores de software:

- **Caixa-Branca**: Normalmente pertence à iniciativa de código aberto e de uso livre. Todo o código é livre para ser visualizado e alterado para implementar novas funcionalidades. Permite novas versões dos componentes.

- **Caixa-Preta**: Os desenvolvedores podem somente utilizar os serviços. Não podem visualizar o código. Reduzem a incompatibilidade quando são realizadas atualizações de versões, pois evitam que os desenvolvedores de software alterem o código do componente e assim quando houver atualização da versão pelo fornecedor haja sobreposição do código alterado.

- **Caixa-Cinza**: Os desenvolvedores podem utilizar somente parte de código referente à manipulação do componente, na realidade funciona quase como um componente caixa-preta.

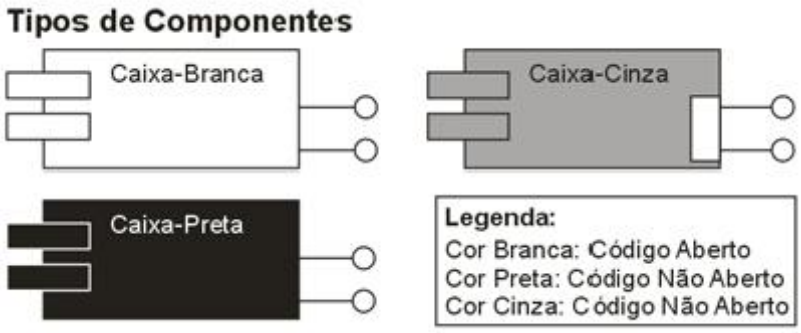


Figura 2.1: Classificação de Componentes quanto à Visibilidade.
Fonte: (SILVA, 2004)

2.1.2 Classificação de Componentes quanto à Aplicabilidade

Os componentes são classificados de acordo com a sua aplicabilidade, conforme a Figura 2.2 em:

Figura 2.2: Classificação de Componentes de acordo com a Aplicabilidade
Fonte: (SILVA, 2004)

- **Funcionais** ou **Verticais**: São componentes que implementam regras de negócios de uma determinada área com conhecimento bem específico.

- **Não funcionais** ou **Horizontais**: São componentes de uso geral que estão ligados a questões de infraestrutura em geral e/ou requisitos não funcionais, por exemplo para impressão de código de barras que pode ser incorporado em software das mais diversas áreas.

2.1.3 Interfaces de Componentes

A Figura 2.3 mostra o encapsulamento da implementação de um componente "X", que torna-se acessível apenas por meio da interface. Os pontos de invocação disponíveis na interface, servem como meios de fornecimento ("output") e consumo ("input") de serviços.

Figura 2.3: Encapsulamento Conceitual de uma Interface.
Fonte: (SILVA, 2004)

Um determinado componente não é completamente independente dos outros componentes e do ambiente. O que determina como se dá essa dependência do componente em relação aos demais e ao ambiente que o cerca, são suas **interfaces**. As interfaces de um componente determinam como esse componente pode ser reutilizado e interconectado com outros componentes, ocultando os detalhes que não são necessários para o reuso. As interfaces definem uma operação ou um conjunto de operações que estão disponíveis para um componente. Uma interface é um conjunto de assinaturas de operações que podem ser invocadas por um cliente.

A interface descreve o comportamento de um componente, que é obtido por

considerar somente as interações de suas interfaces e por ocultar todas as demais interações. Existem dois tipos de interfaces: **providas** (*provides*) e **requisitadas** (*requires*). Um componente suporta uma interface provida (define os serviços fornecidos pelo componente) se contém a implementação de todas as operações definidas pela interface. Por outro lado, um componente necessita de uma interface requisitada (especifica que serviços devem estar disponíveis a partir do sistema que está utilizando o componente) se solicita uma interação definida na interface e espera que algum outro elemento de software suporte essa interface.

2.1.4 Diagrama de Componentes

Diagrama de Componentes é um diagrama da *Unified Modeling Language* (UML) que pode representar as relações entre os componentes e subsistemas de uma aplicação por meio de chamadas as suas interfaces.

A Figura 2.4 mostra uma seta direcionada e conectada a interface "Y" do componente "X" com a finalidade de representar a relação de dependência entre o Subsistema "A" e o Componente "X".

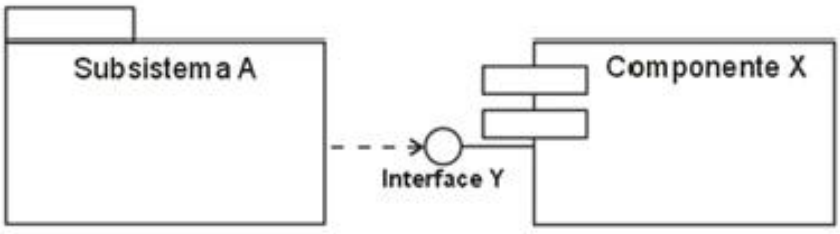


Figura 2.4: Diagrama de Componente com Interface Única.
Fonte: (SILVA, 2004)

A Figura 2.5 mostra um componente "X" com múltiplas interfaces. Os subsistemas "A" e "B" acessam a interface "Y" do componente "X" e um terceiro subsistema "C", acessa a interface "Z". Cada uma das interfaces pode apresentar um conjunto de serviços diferenciados e pode ser reutilizada por mais de um subsistema.

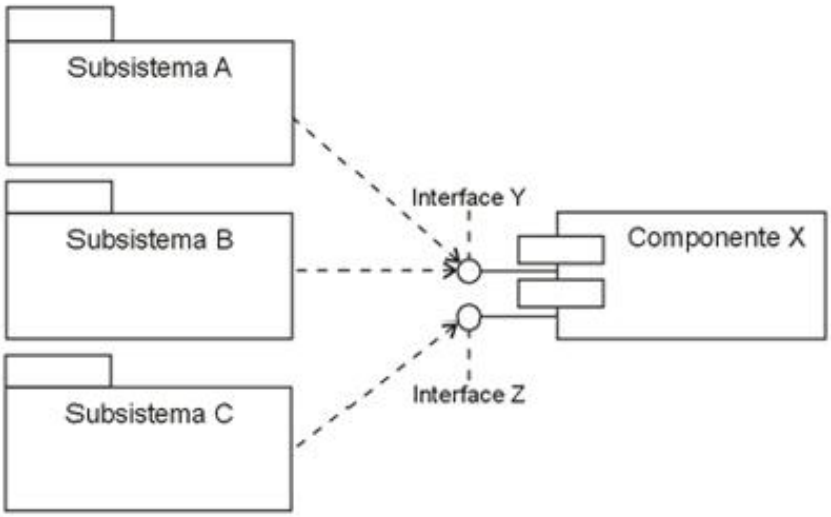


Figura 2.5: Diagrama de Componente com Interface Múltipla.

Fonte: (SILVA, 2004)

2.1.5 Tecnologias de Componentes

Segundo EMMERICH and KAVEH (2002) *apud* SILVA (2004) as Tecnologias de Componentes podem ser classificadas conforme mostrado na Figura 2.6:

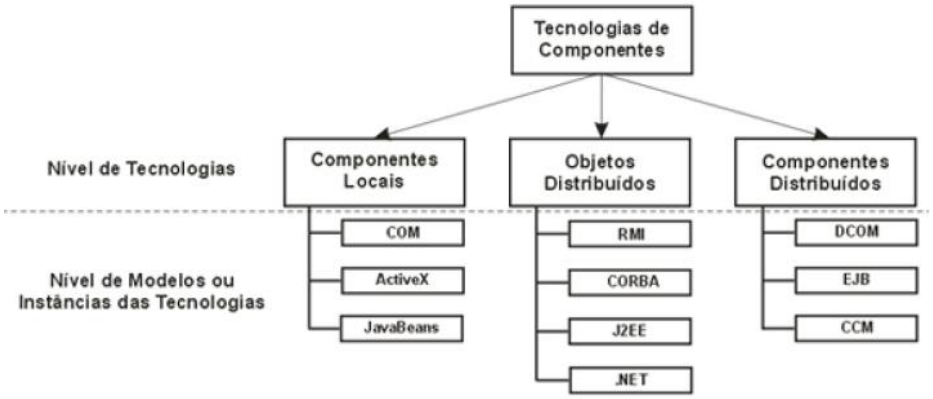


Figura 2.6: Tecnologia de Componentes e seus Modelos.

Fonte: (SILVA, 2004)

- **Componentes Locais**: Baseia-se em modelos para construção de componentes que irão executar e se comunicar num mesmo ambiente ou instância de sistema operacional. Seus principais modelos são:

- *Component Object Model*(COM): Modelo de componentes da Microsoft como tentativa de integrar aplicações *Windows* com as demais, posteriormente ampliado para COM+, visando facilitar o uso e a criação de componentes COM. É um padrão binário, que suporta a comunicação de componentes escritos em diferentes linguagens.

- *ActiveX*: Este modelo de componentes de controles da Microsoft baseia-se no modelo COM para se comunicar com outros tipos de componentes COM. Possuem funcionalidades especialmente projetadas para facilitar a distribuição de componentes em rede e prover a integração de recursos em navegadores *web* (funcionalidades para assinatura de código, permitindo que os usuários identifiquem os autores dos controles antes de autorizar ou negar sua execução local).

- *JavaBeans*: Modelo de componentes da Sun baseado na linguagem Java, que define um protocolo a ser implementado pelos componentes denominados beans. Tal protocolo consiste, principalmente, na definição de propriedades, métodos e eventos, que juntos, definem a interface do componente. Este protocolo preocupa-se, principalmente, em definir como deve ocorrer a comunicação entre dois ou mais beans, sem impor regras semânticas para troca de informações.

- **Objetos Distribuídos**: Os modelos têm por objetivo prover a comunicação entre múltiplos objetos residentes em diferentes ambientes. Tal comunicação entre objetos clientes e servidores ocorre, normalmente, por meio de invocações remotas de métodos, sobre a infra-estrutura de rede existente e em plataformas de desenvolvimento em camadas e integração de aplicações construídas em diferentes linguagens. Seus principais modelos são:

  - *Common Object Request Broker Architecture* (CORBA): Modelo de objetos que foi especificado pelo OMG e possui uma arquitetura para operação conjunta de componentes independentes de linguagem, plataforma de execução e localização física. A aplicação é encapsulada num objeto CORBA, dotado

de uma interface pública em diversas linguagens(Java, C++ e Smalltalk), bem como intermediários de comunicação entre objetos distribuídos denominados *Object Request Brokers* (ORB), que implementam os meios de comunicação entre componentes para cada uma dessas linguagens.

  - *Modelo de Aplicações Java* - Edição Corporativa (J2EE Application Model): Modelo de objetos distribuídos para o desenvolvimento de aplicações baseadas em Java, da Sun, define um padrão para o desenvolvimento de software em multi-camadas, visando simplificar a construção de aplicações e de componentes modulares, por meio de um conjunto de serviços responsáveis. O modelo prega o encapsulamento das camadas por tipos específicos de componentes, como por exemplo, EJBs (regras de negócio) e o J2EE (permite comunicações entre componentes, de forma transparente, por exemplo, via RMI). O modelo J2EE divide as aplicações corporativas em porções fundamentais: componentes, contêineres e conectores. Os componentes representam o foco da produção dos desenvolvedores de software nas empresas, enquanto que os fornecedores implementam os contêineres e os conectores que irão, respectivamente, suportar os componentes e permitir suas comunicações com outras plataformas.

  - *.NET*: Modelo de objetos distribuídos desenvolvido para a construção de sistemas de software na plataforma da Microsoft, que proporciona um ambiente de desenvolvimento com características de Orientação a Objetos, Multi-plataformas e Multi-linguagens. O .NET possui uma Máquina Virtual e um *Framework*, que permitem o desenvolvimento e a execução de aplicações tanto em ambientes *Desktop* e *Web*, quanto em ambientes móveis. Somente o Windows possui uma Máquina Virtual para .NET homologada. Atualmente, o Modelo .NET suporta linguagens como a C#, a VB.NET, a J# e a Delphi para .NET.

- **Componentes Distribuídos**: Modelos para construção de componentes que possam executar e se comunicar independentemente de sua localização física.

Combinam as características dos Componentes Locais e de Objetos Distribuídos, visando a obtenção de melhores definições de modelos e de infra-estruturas que possibilitem a criação de componentes reutilizáveis e comunicáveis, num ambiente corporativo distribuído. Seus principais modelos são:

- *COM Distribuído* (Distributed COM - DCOM): Este modelo utiliza-se da Tecnologia de Componentes Distribuídos e expande o modelo COM da Microsoft, habilitando a comunicação em rede de seus componentes, inclusive com *Applets Java* e componentes *ActiveX* baseados em COM. Tais características de objetos e componentes distribuídos, que se encontram implantadas pelo uso do modelo de objetos distribuídos RMI serviram de base, também, para a implantação do modelo de objetos distribuídos CORBA.
- *Enterprise JavaBeans - EJB* - Especificado pela *Java Community Process* (JCP), e é responsável pela camada de Regras de Negócio do J2EE de aplicações distribuídas para a porção servidor e introduz, por meio de seu *framework*, implementações que facilitam a inclusão de características não funcionais nos componentes, tais como segurança, suporte a transações, controle de concorrência e persistência. O modelo EJB apresenta também uma especificação mais formal do que a do JavaBeans, e ao contrário do que ocorre com RMI e CORBA, o gerenciamento de instâncias remotas não faz parte das responsabilidades do desenvolvedor.
- *Modelo de Componentes CORBA* (CORBA Component Model - CCM) - Este modelo especificado pela OMG, também se utiliza da Tecnologia de Componentes Distribuídos e, da mesma forma que o EJB, orienta a construção de aplicações distribuídas para a porção servidor. A principal diferença em relação ao EJB refere-se ao seu suporte a múltiplas linguagens, desde que a última camada da aplicação seja implementada em CORBA.

A Figura 2.7 mostra parte do código na linguagem Java para implementar uma regra de negócio de uma aplicação utilizada para Registrar Ordem de Serviço.

```
// INTERFACE
public interface ClienteDAO {
   dominio.Cliente getClientePorCPF(String cpf);
   dominio.Cliente getClientePorCPF(String cpf, Session sessao);
   dominio.Cliente getClientePorID(Integer id);
   dominio.Cliente insere(dominio.Cliente value, Session sessao);
}
// IMPLEMENTACAO
public class ClienteDAOImp implements ClienteDAO {
   public ClienteDAOImp() {}
   public Session getSession() {
      return util.HibernateUtil.currentSession();
   }
   public Cliente insere(Cliente value, Session sessao) {
      if (value==null) return null;
      Transaction tx =null;
      Session session = sessao;
      if (sessao == null){
         session = getSession();
         tx = session.beginTransaction();
      }
      try {
         Cliente aux = getClientePorCPF(value.getCpf(), session);
         if (aux!=null) return aux;
         session.save(value);
         if (tx!=null) tx.commit();
      } catch (HibernateException ex) {
         if (tx!=null) tx.rollback();
         System.out.println(ex.getMessage());
         return null;
      } finally {
         if (sessao == null)  util.HibernateUtil.closeSession();
      }
      return value;
   }
```

```
public Cliente getClientePorCPF(String cpf) {
   Cliente value = null;
   try {
      Session session = getSession();
      value = getClientePorCPF(cpf, session);
   } catch (HibernateException ex) {
      ex.printStackTrace();
   } finally { util.HibernateUtil.closeSession();}
   return value;
}
public Cliente getClientePorCPF(String cpf, Session sessao) {
   if (cpf==null || cpf.length()<10) return null;
   Session session = sessao;
   try {
      if (sessao == null) session = getSession();
      String consulta = "from Cliente as cl where cpf='"+ cpf + "'";
      List cls = session.createQuery(consulta).list();
      if (cls.size() == 0) {return null;}
        else {  return (Cliente) cls.get(0);
      }
   } catch (HibernateException ex) {
      ex.printStackTrace();
   } finally {  if (sessao == null)  util.HibernateUtil.closeSession(); }
   }
   return null;
}
public Cliente getClientePorID(Integer id) {
   Cliente value = null;
   try {  Session session = getSession();
         value = (Cliente) session.load(Cliente.class, id);
       } catch (HibernateException ex) {
   ex.printStackTrace();
   } finally { util.HibernateUtil.closeSession()
   }   return value;
} }
```

Figura 2.7: Código Java Implementando um Componente.

2.2 Desenvolvimento Baseado em Componentes

No Desenvolvimento de Software Baseado em Componentes (DBC) um software é criado conectando-se partes, chamadas componentes. Um componente é uma unidade de software bem definida e que pode ser utilizada em conjunto com outros componentes para formar grandes unidades de software. Cada componente tem funcionalidade própria para resolução de um problema especifíco que é oferecida através de uma interface que permite acessar os seus métodos.

Algumas definições em relação ao tema são encontradas em HOOPER and CHESTER (1991). É necessário distinguir o conceito de reutilização do conceito conexo de reusabilidade. Em geral, **reusabilidade** diz respeito à medida da facilidade em que o software pode ser usado (com ou sem adaptações) na solução de múltiplos problemas ou na solução de um problema diferente daquele para o qual ele foi originalmente desenvolvido. Já **reutilização** diz respeito à ação de sintetizar a solução para um problema, decompondo-o em subproblemas para os quais haja soluções pré-definidas. É o uso de conceitos ou produtos previamente adquiridos ou construídos em uma nova situação. Isso envolve a representação desses produtos em vários níveis de abstração, o armazenamento dos mesmos para futuras referências, a identificação de similaridades entre situações novas e antigas, a recuperação de produtos já desenvolvidos (ou parte deles) e sua adaptação na nova situação. Novos softwares são assim implementados a partir de outros softwares preestabelecidos.

O DBC requer princípios tais como a separação entre a especificação e implementação, que podem contribuir para melhorar a manutenabilidade dos sistemas desenvolvidos, uma vez que o desenvolvimento de software é um processo dinâmico em que desenvolvedores necessitam constantemente alterar os sistemas desenvolvidos à medida que novos requisitos ou mudanças aparecem. O reuso caracteriza-se pela utilização de produtos de software muitas vezes em uma situação diferente daquela para a qual estes produtos foram originalmente desenvolvidos. A reutilização de componentes de software em diferentes sistemas cria a interdependência que torna o processo de evolução destes sistemas mais complexos (HOOPER and CHESTER, 1991).

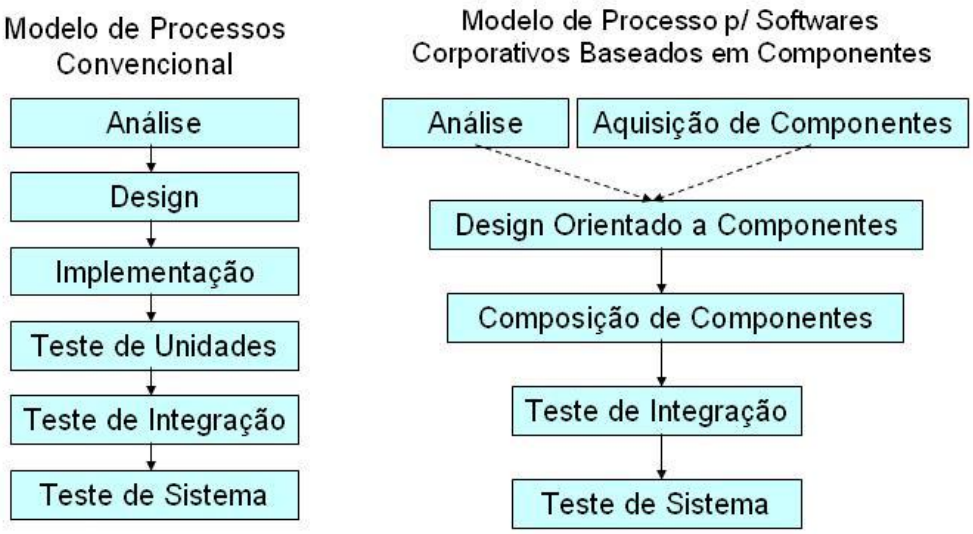


Figura 2.8: Desenvolvimento de sistema tradicional versus DBC.
Fonte: (AGRESTI and MCGARRY, 1998)

Este novo processo de desenvolvimento de software baseado em componentes modifica sensivelmente o processo de desenvolvimento tradicional, introduzindo novas características. A separação entre a especificação e a implementação de componente de software visa, fundamentalmente, permitir a divisão de duas atividades independentes: a produção de componente de software e a integração de sistemas. O foco passa a ser a integração de componentes, onde questões arquiteturais devem ser consideradas de forma mais rigorosa, requerendo um esforço em requisitos, testes e integração de componentes. Neste contexto, são fundamental importância os repositórios de componentes, onde os produtores de componentes depositam os artefados produzidos (especificações e implementações de componentes) e os consumidores (desenvolvedores) de componentes recuperam-nos para integrar a seus sistemas. A Figura 2.8 mostra um comparativo do método tradicional de desenvolvimento de software e o DBC.

A efetiva reutização de componentes em larga escala requer um processo de evolução sistemático de componentes de software que minimize os conflitos gerados por essas interdependências e que os repositórios de componentes de diferentes produtores, geograficamente distribuídos, possam interoperar para formação de uma ampla base de componentes disponíveis para desenvolvedores integradores de componentes.

Para que o processo DBC possa ser adotado em larga escala, um importante requisito é a disponibilidade de um conjunto expressivo de componentes nos diversos domínios de aplicação e com as mais variadas funcionalidades.

Componentes de software são projetados para o reuso. Para facilitar a reutilização, pesquisas têm sido realizadas para desenvolver repositórios, que possam auxiliar na busca de componentes de software com a finalidade de aumentar a reusabilidade e a produtividade de desenvolvedores de software.

Vários são os benefícios percebidos pela reutilização de software. Para AGRESTI and MCGARRY (1998) e bem como para a maioria dos autores, os principais objetivos da reutilização são claros e bem definidos. Dentre eles, podem ser destacados:

- **Aumento da Produtividade**: reutilizando-se artefatos, o esforço no desenvolvimento diminui gerando um produto final de igual ou superior qualidade do que um gerado sem reutilização, pois usa componentes já existentes;
- **Aumento da Qualidade**: como os artefatos foram previamente testados, a probabilidade de sua corretude é ainda maior. Se a parte tem qualidade, o todo também terá grande propensão a ter qualidade;
- **Redução dos custos**: o custo de construção de artefatos reutilizáveis é diluído entre os vários projetos onde será reutilizado;
- **Redução no tempo de entrega**: se o esforço no desenvolvimento diminui, consequentemente o tempo de entrega também diminui;
- **Padronização**: como os artefatos seguem uma padronização pré-definida, sua reutilização em um sistema causará conseqüente padronização;
- **Interoperabilidade**: a padronização garante que os sistemas se comportem de maneira comum, aumentando a interoperabilidade;
- **Previsibilidade/Confiabilidade/Redução de Riscos**: artefatos bem testados e reutilizados várias vezes têm alto grau de confiabilidade, diminuindo os riscos de erro;

- **Maior facilidade de gerenciamento**: visto que usa componentes já compreendidos pelos desenvolvedores.

Por viabilizar a disponibilidade de uma grande quantidade e variedade de componentes, repositórios de componentes têm sido apontados como um dos fatores críticos para o sucesso do DBC, permitindo o compartilhamento de componentes entre produtores e consumidores. Neste sentido, diversos sistemas de repositórios de componentes têm sido propostos.

Os atuais gerenciadores e repositórios de componentes, na sua maioria, permitem apenas o reuso de componentes *"black-box"*, ou seja, componentes empacotados onde não se tem acesso ao código fonte, o que dificulta tarefas de adaptação ou evolução. Para estimular a cultura do reuso, é necessário oferecer subsídios para reutilização de componentes *"white-box"*, ou seja, componentes com código fonte disponível. No entanto, sistemas de repositório não devem apenas armazenar componentes, mas também meta-dados que os descrevem.

2.3 Reusable Asset Specification (RAS)

2.3.1 Definições

Um componente está inserido dentro de um conjunto mais amplo de elementos, neste contexto denominado **ativo de software** (**asset** em inglês). Um ativo de software, segundo EZRAN et al. (2002) é qualquer artefato, passível de reutilização, produzido em qualquer fase de um processo de desenvolvimento de um sistema computacional, que deve ser utilizado seguindo certas regras de uso, sendo possível os artefatos possuírem algum ponto de customização. Assim, pode-se, considerar um ativo de software: artefatos de documentação, padrões de análise e projeto, normas de codificação, módulos executáveis, códigos e outros arterfatos gerados em todo o cilco de desenvolvimento do software. Componentes de software é apenas uma das categorias enquadrada como ativo de software.

Visando uma melhor organização e caracterização dos ativos de software para

fins de reuso, foi desenvolvido o padrão RAS (*Reusable Asset Specification*), resultante de um esforço da indústria de software apoiada pelo OMG. A primeira versão da especificação do RAS foi submetida pela OMG em agosto de 2003 com o objetivo de prover um conjunto de informações para descrição e empacotamento de ativos de software, determinando como ele deve estar estruturado de modo que este ativo possa ser facilmente localizado e utilizado. Atualmente, encontra-se na versão 2.2 (CARLSON et al., 2005) e pode ser baixada a partir do link <www.omg.org/technology/documents/formal/ras.htm>.

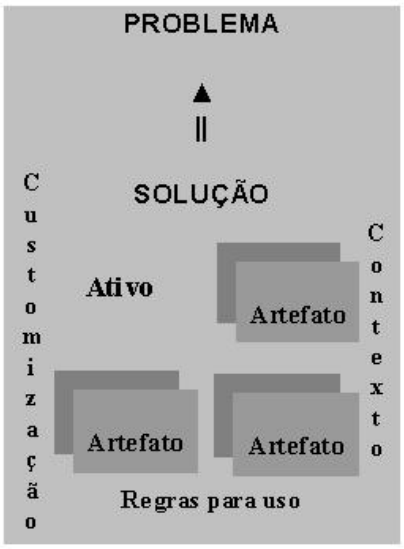


Figura 2.9: Definição Geral de um Ativo.
Fonte: (CARLSON et al., 2005)

O padrão RAS (CARLSON et al., 2005), conforme a Figura 2.9 apresenta, uma definição simples e objetiva: um **ativo reutilizável** oferece uma solução para um problema em determinado contexto e é composto por um conjunto de artefatos que devem ser utilizados seguindo certas regras de uso, sendo possível os artefatos possuírem algum ponto de customização.

Um ativo de software deve conter um arquivo *XML* que o descreva, chamado *manisfest file*, e um *XML Schema*, usado para validar o *manifest file*. Deve conter ainda pelo menos um artefato. O *RAS* é dividido em duas grandes categorias: *Core RAS* e os *profiles*, conforme mostrado na Figura 2.10.

Figura 2.10: Núcleo Ras e Perfil.
Fonte: (CARLSON et al., 2005)

Core RAS define os elementos fundamentais da especificação de ativos de software, que podem ser estendidos usando profiles. A especificação RAS define três *profiles* principais: *Default Profile*, *Default Component Profile* e *Web Service Profile*. O primeiro consiste no grupo principal de características; o segundo e o terceiro são extensões do *Default Profile* para representar características relacionadas a componentes e *Web Services*, respectivamente.

2.3.2 Reusable Software Asset Types

Um *reusable asset* fornece uma solução para um determinado problema dentro de um certo contexto (de negócio, documentação, desenvolvimento, teste, etc) e é composto

por um conjunto de artefatos, que devem ser utilizados seguindo certas regras de uso e podem ter um ponto de customização.

Um ativo de software pode ser descrito segundo três dimensões (CARLSON et al., 2005):

- **Granularidade**: que descreve se o ativo pode ser usado apenas para solução de um problema específico ou pode ser empregado em uma variedade de situações. Granularidade pequena, por exemplo, um algoritmo de cálculo de digito verificador, e granularidade grande, como por exemplo, um *framework*.

- **Variabilidade**: que descreve se um ativo pode ser alterado para ser adaptado a um problema ou se deve ser usado sem alterações. Esse aspecto está relacionado com a "Visibilidade" descrita na seção 2.1.1.

- **Articulação**: que descreve se o ativo oferece uma solução completa para o problema, com documentação, casos de testes, etc., ou apenas parcial.

Segundo essas dimensões, um componente de software é um ativo com baixa granularidade e pouca variabilidade. A Figura 2.11 ilustra a combinação dos três critérios apresentados.

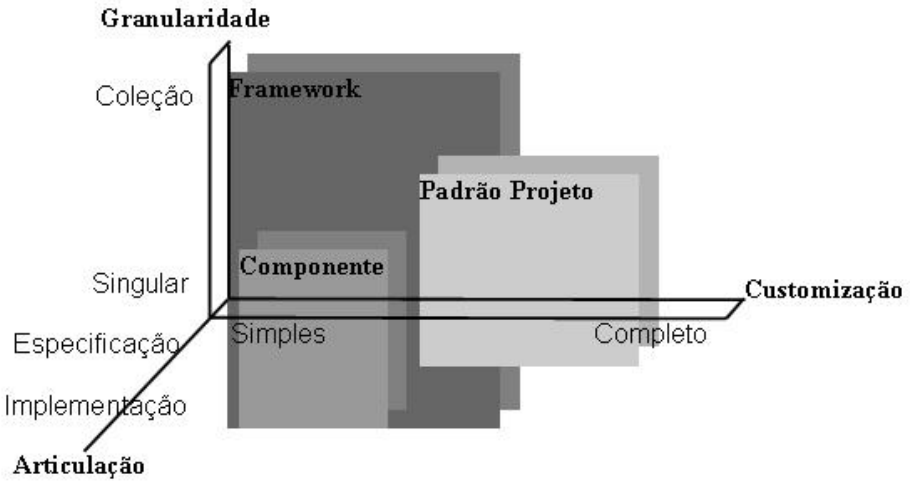


Figura 2.11: Tipos de Ativos de Software Reutilizáveis.
Fonte: (CARLSON et al., 2005)

2.3.3 Empacotamento de Ativos

Todo ativo tem que possuir um arquivo XML que o descreva, chamado *Manifest File* e um arquivo *XML Schema* usado para validá-lo e deve possuir pelo menos um artefato para ser considerado um recurso reutilizável válido.

Um pacote de um ativo é a coleção de arquivos dos artefatos mais o arquivo *Manifest File*. Há vários cenários para o empacotamento do ativo:

- Empacotamento em um arquivo: ativo, artefatos e *manifest file*;
- Artefatos podem permanecer em seu lugar de origem;
- Artefatos podem ser transferidos para outro local, quando empacotados.

A Figura 2.12 mostra o empacotamento de ativo em um único arquivo que posteriormente é compactado. Esta técnica pode ser usada em ambiente de desenvolvimento com times de desenvolvedores, como também em ambientes com um único desenvolvedor.

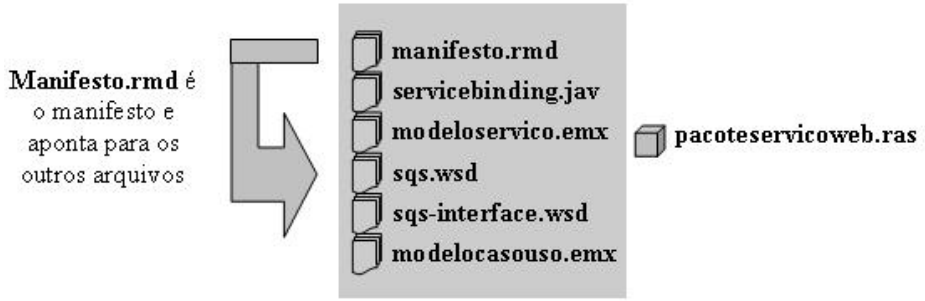


Figura 2.12: Empacotamento de Ativo em único arquivo.
Fonte: (CARLSON et al., 2005)

2.3.4 Core RAS

O *Core RAS* é o núcleo da especificação, comum a todos os *reusable assets*. Os *profiles* são uma extensão do *Core RAS* onde podem ser adicionadas livremente novas restrições e novos elementos, mas as restrições herdadas do *Core RAS* não podem ser relaxadas nos *profiles*. Os *profiles* representam os elementos fundamentais da especificação e também uma flexibilização do modelo proposto, pois se for necessário, um novo *profile* pode ser adicionado. Cada *reusable asset* deve estar associado a um *profile* e esta informação deve estar presente em seu *manifest file*. E deve existir um *XML Schema* associado a cada *profile*, pois são eles que definem a semântica do *asset*, refletida em seu *manifest file*.

O conjunto das meta-informações e dos artefatos que compõe o *reusable asset* é organizado em 4 seções, conforme a Figura 2.13.

Figura 2.13: Seções do Núcleo RAS.

Fonte: (CARLSON et al., 2005)

A Figura 2.14 mostra um diagrama de classe com a relação entre essas seções: **Classificação**, **Solução**, **Uso** e **Ativos Relacionados**, que serão descritas nas seções seguintes

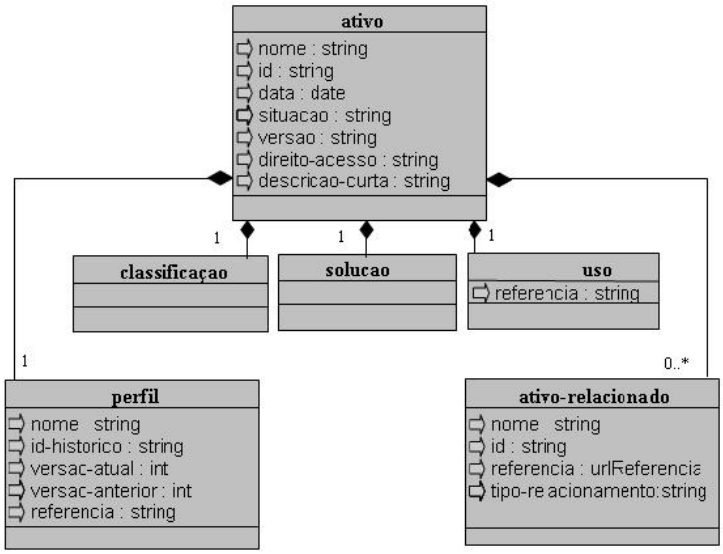


Figura 2.14: Relacionamento entre as Seções RAS.

Fonte: (CARLSON et al., 2005)

Classificação

Classificação é a seção onde se encontra uma descrição do contexto no qual ele é relevante e os descritores do ativo, tais como linguagem de programação utilizada, idioma, principal tecnologia, etc. Serve como um *container* para os elementos de descrição, onde os atributos podem ser definidos livremente. Para isto, a classe *Classification* pode ser relacionada com um contexto, por exemplo, o atributo "Tecnologia"só é relevante dentro do contexto "Desenvolvimento". A Figura 2.15 representa o diagrama de classe da seção Classificação.

Solução

Solução é a seção onde se descreve a lista dos artefatos que compõem o *asset*. Um artefato pode depender de outro artefato ou pode ainda ser composto de outros artefatos. Cada um pode possuir um ponto de customização, indicado pela classe

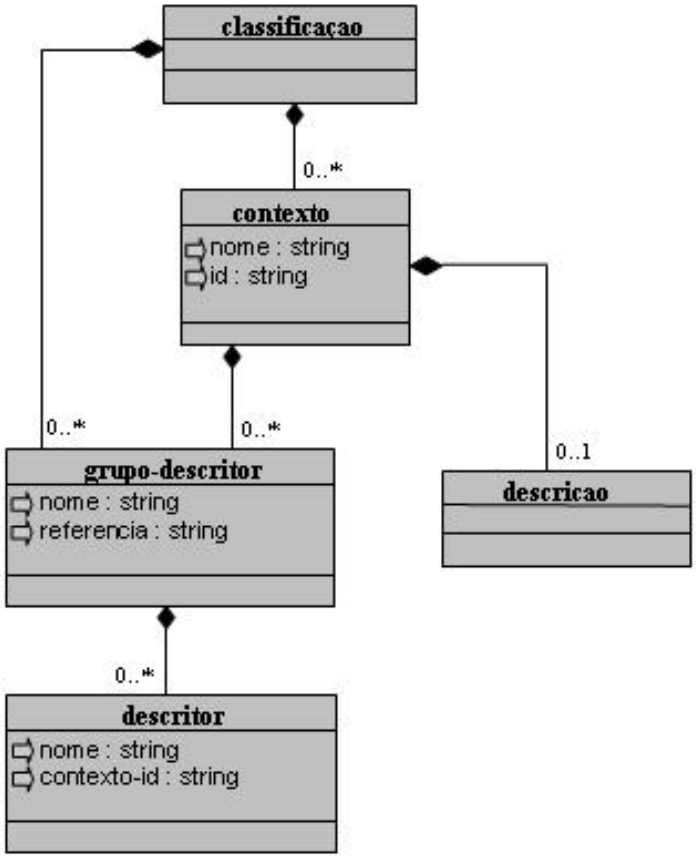


Figura 2.15: Seção de Classificação.

Fonte: (CARLSON et al., 2005)

VariabilityPoint e ser relevante no contexto específico. A Figura 2.16 representa o diagrama de classe da seção Solução.

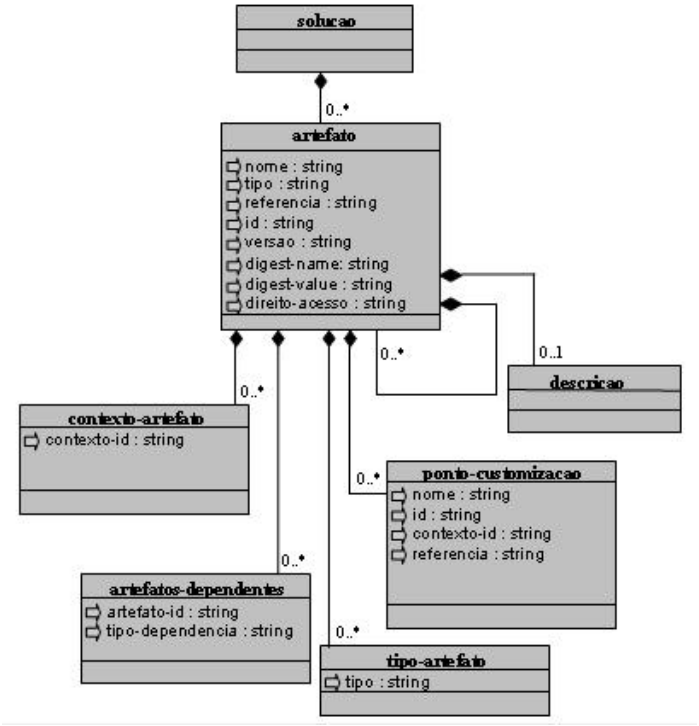


Figura 2.16: Seção de Solução.
Fonte: (CARLSON et al., 2005)

Uso

Uso é a seção onde se encontra as regras de utilização, instalação e customização. A classe *Usage* completa o conjunto de meta-informações, descrevendo a maneira de usar o ativo e seus artefatos, se há um ponto de variação em algum deles, como fazê-lo, ou seja, define as regras de utilização para que os artefatos que compõem a solução implementem corretamente o que está descrito na classificação do *asset*. A Figura 2.17 mostra a seção de Uso.

Dessa forma, a seção possui diversas classes que permitem que tanto o *asset* quanto seus artefatos possuam um conjunto de atividades, relacionadas ou não a um contexto particular. São elas: ArtifactActivity, AssetActivity, ContextRef, Activity, VariabilityPointBinding que são mostradas na Figura 2.18.

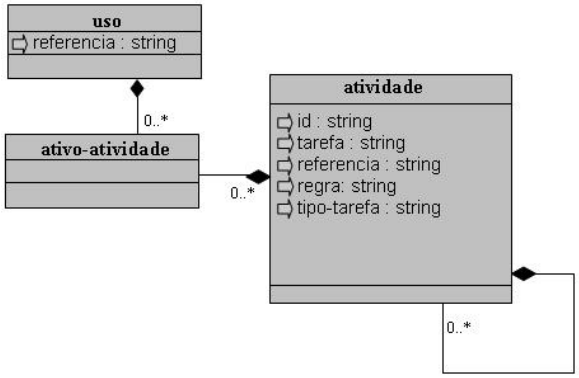


Figura 2.17: Seção de Uso.

Fonte: (CARLSON et al., 2005)

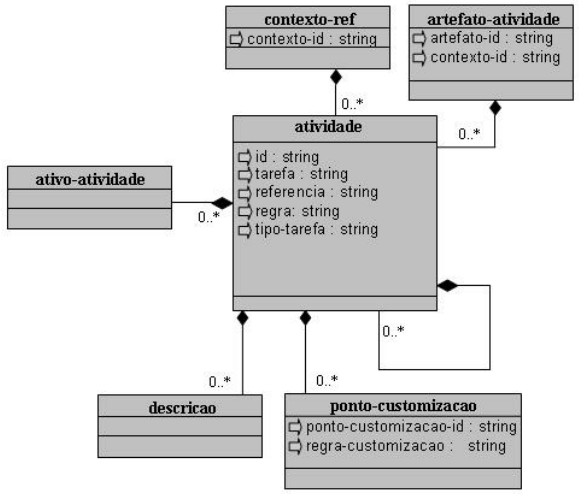


Figura 2.18: Diagrama da Classe Atividade.

Fonte: (CARLSON et al., 2005)

Ativos Relacionados

Ativos Relacionados é a seção que descreve a relação do *asset* com os outros. Os principais atributos da classe *RelatedAsset* são o nome e o tipo de relacionamento (similiar, dependency, agregation e parent). A Figura 2.19 mostra o diagrama de classe da classe.

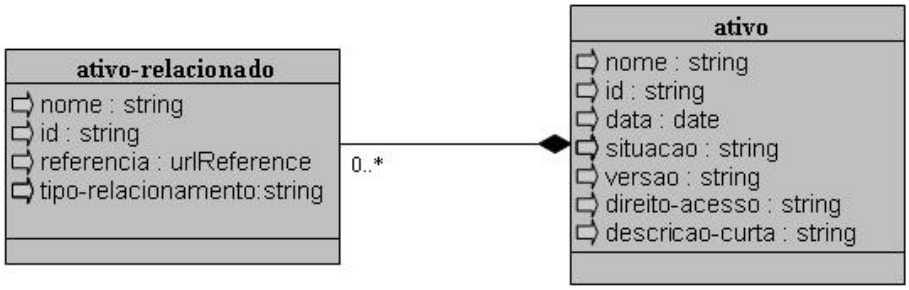


Figura 2.19: Seção de Ativos Relacionados.
Fonte: (CARLSON et al., 2005)

Algumas características e abordagens definidas pelos *profiles* do RAS possuem deficiências. Primeiramente, o *Default Component Profile* organiza os artefatos dos ativos apenas considerando o processo de desenvolvimento *UML Components*. Outra deficiência do *Default Component Profile* é não permitir o reuso de uma descrição de interface por mais de uma descrição de componente. Além disto, RAS também não descreve interfaces requeridas, propriedades, eventos e exceções, que são características presentes nos principais modelos de componentes. O RAS não provê meios para descrever modelos de negócios, evoluções entre versões de um ativo, requisitos não-funcionais e certificação. Embora RAS possua tais limitações, as mesmas podem ser contornadas com a definição de novos *profiles* que solucionam as limitações apresentadas.

2.3.5 XML

Extended Markup Language (linguagem extendida de marcação), abreviado XML, descreve objetos de dados chamados documentos XML e parcialmente descreve o com-

portamento de programas de computador que o processa. XML é definido como um perfil de aplicação do SGML. SGML é *Standard Generalized Markup Language* (linguagem de markup generalizada padronizada) definido pelo ISO 8879. Os documentos XML estão em conformidade com os documentos SGML.

Documentos XML são feitos de unidades de armazenamento chamadas entidades, que contém dados *parsed* (analisados gramaticalmente) e dados não *parsed*. Dados *parsed* são compostos de caracteres, alguns formam os dados em caracteres (*character data*), e alguns destes formam a marcação (*markup*). A marcação codifica uma descrição do leiaute do armazenamento e da estrutura lógica. XML provê mecanismos para impor restrições no leiaute de armazenamento e estrutura lógica. *Tags* são estruturas de marcação que consistem em breves instruções, tendo as marca de início e outra de fim. A Figura 2.20 mostra o documento XML com as informações do *asset*.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<asset access-rights="read/write" id="25875315"
       name="Jakarta.Commons.Logging_abstractComponent">
       <description>The Logging Wrapper Library  offers wrappers around
an extensible set of concrete logging implementations.</description>
     <classification>
         <descriptor-group>
             <descriptor name="Platform">Java</descriptor>
             <descriptor name="area">Logging</descriptor>
         </descriptor-group>
     </classification>
     <solution>
         <artifact name="Users Guide" reference="doc/usersGuide.doc"/>
         <artifact name="pagina do componente"
                reference="http://jakarta.apache.org/commons/logging"/>
     </solution>
</asset>
```

Figura 2.20: Exemplo XML para meta-dados de Ativo.

XML Schema

XML *Schema* é uma linguagem para a descrição da estrutura e das restrições relativas ao conteúdo de documentos XML, que inicialmente foi proposto pela Microsoft, mas tornou-se uma recomendação oficial do Word Wide Web Consortium (W3) em maio de 2001.

XML *Schema* consiste de componentes tais como definições de tipos e declarações de elementos. Estes podem ser usados para avaliar a validade de itens de informações de elementos e atributos, e ainda mais podem especificar adições para estes itens e seus descendentes.

O XML Schema define:

- elementos que podem aparecer em um documento.
- atributos que podem aparecer em um documento.
- quais são os elementos filhos.
- o número de elementos filhos.
- se um elemento está vazio ou se este pode incluir texto.
- tipos de dados para elementos e atributos.
- valores default e valores fixos para elementos e atributos.

2.4 Ontologia na DBC

2.4.1 Definições de Ontologia

A palavra ontologia tem a sua origem nos pensamentos filosóficos de Aristóteles. A disciplina de computação resgatou este termo, inserindo-o num novo contexto, apesar de próximo do significado original. Dentre as dimensões da metafísica, a ontologia trata do ser enquanto ser. Neste sentido, apesar dos múltiplos significados do ser, ele faz referência a um único princípio, que une todos estes múltiplos significados.

Uma ontologia é uma especificação explícita dos objetos, conceitos e outras entidades que assumimos existirem em uma área de interesse, além das relações entre esses conceitos e restrições expressados através de axiomas (GRUBER, 1993).

Segundo GUARINO (1998), as ontologias podem ser classificadas segundo o seu nível de generalidade, em: ontologias genéricas, ontologias de domínio, ontologias de tarefa e ontologias de aplicação.

Uma ontologia genérica descreve conceitos gerais, tais como, espaço, tempo, matéria, objeto, evento, ação, sendo independentes de domínio ou problema particulares.

Uma ontologia de domínio reúne conceitos e seus relacionamentos em um domínio particular, definindo restrições na estrutura e conteúdo do conhecimento desse domínio; por exemplo: o domínio jurídico, direito tributátio, microbiologia, etc.

Uma ontologia de tarefa expressa conceitos sobre resolução de problemas, independentemente do domínio em que ocorram, isto é, descrevem o vocabulário relacionado a uma atividade ou tarefa genérica; por exemplo, o acesso à informação.

Uma ontologia de aplicação descreve conceitos dependentes ao mesmo tempo de um domínio particular e de um conjunto de tarefas específicas. Estes conceitos freqüentemente correspondem a papéis desempenhados por entidades do domínio enquanto realizam certas atividades; por exemplo, o acesso à informação jurídica.

De forma simples, para elaborar ontologias, definem-se categorias para as coisas que existem em um mesmo domínio. Ontologia é um "catálogo de tipos de coisas" em que se supõe existir um domínio, na perspectiva de uma pessoa que usa uma

determinada linguagem (SOWA, 2000). Trata-se de "uma teoria que diz respeito a tipos de entidades e, especificamente, a tipos de entidades abstratas que são aceitas em um sistema com uma linguagem" (CORAZZON, 2002).

Os conceitos de uma ontologia de domínio ou de uma ontologia de tarefa devem ser especializações dos termos introduzidos por uma ontologia genérica. Os conceitos de uma ontologia de aplicação, por sua vez, devem ser especializações dos termos das ontologias de tarefas e das ontologias de domínio.

O uso de ontologias visa capturar o conhecimento declarativo do domínio e fornecer uma compreensão deste, possibilitando o reuso e o compartilhamento através de aplicações em grupos. "Ontologias fornecem um vocabulário comum de uma área e define - com níveis diferentes de formalismos - o significado dos termos e dos relacionamentos entre eles", (CORAZZON, 2002).

Segundo PEREZ and BENJAMINS (2006), uma ontologia de domínio é definida através de conceitos ("termos ou classes organizadas em taxonomias e partonomias, e seus domínios de valores"), relações, funções, axiomas e instâncias. Os princípios básicos para o construção de uma ontologia foram resumidos por (PEREZ and BENJAMINS, 2006):

- **Clareza e objetividade**: os termos devem ser acompanhados de definições objetivas e também de documentação em linguagem natural.
- **Completeza**: uma definição deve expressar as condições necessárias e suficientes para expressar um termo, indo além das necessidades circunstanciais de uma aplicação.
- **Coerência**: para permitir derivar inferências que sejam consistentes com as definições.
- **Extensibilidade monotônica**: para permitir a inclusão de novos termos sem revisão das definições existentes.
- **Mínimo compromisso ontológico**: para permitir que sejam definidas tão poucas suposições quanto possíveis sobre o mundo a ser modelado, permitindo

que as especializações e instanciações da ontologia sejam definidas com liberdade.

- **Princípio da distinção ontológica**: as classes definidas na ontologia devem ser disjuntas, sem superposição de conceitos.
- **Diversificação das hierarquias**: para aproveitar ao máximo os mecanismos de herança múltipla.
- **Modularidade**: para minimizar o acoplamento entre os módulos.
- **Minimização**: da distância semântica entre conceitos similares, de forma a agrupá-los e representá-los utilizando as mesmas primitivas;
- **Padronização**: dos nomes sempre que possível.

Segundo HINZ and PALAZZO (2006) *apud* (BORST, 1997) "Uma ontologia é uma especificação formal e explícita de uma conceitualização compartilhada". Nessa definição, "formal" significa legível para computadores; "especificação explícita" diz respeito a conceitos, propriedades, relações, funções, restrições e axiomas explicitamente definidos; "compartilhado" quer dizer conhecimento consensual; e "conceitualização" diz respeito a um modelo abstrato de algum fenômeno do mundo real. A conceitualização é formada por um vocabulário controlado que é arranjado hierarquicamente e através de relações entre conceitos, como nas taxonomias e tesauros. Uma conceitualização é uma visão abstrata e simplificada do mundo que se deseja representar.

A ontologia gerada por um sistema é o resultado da integração de processos representados através de uma árvore, onde cada nó representa um conceito da ontologia, e cada relacionamento entre os nós pode ser um relacionamento taxonômico, partonômico. O conhecimento nas ontologias é formalizado usando cinco tipos de componentes: classes, relações, funções, axiomas e instâncias, descritos abaixo:

- **Conceitos** representam qualquer coisa do domínio sobre a qual alguma coisa é dita; incluem os objetos do domínio, a descrição de uma tarefa, de uma função, ação, estratégia etc;

- **Relações** representam os tipos de interações entre os conceitos do domínio. São definidas formalmente como qualquer subconjunto de um produto de n conjuntos, ou seja: R:C1 x C2 x ,..., x Cn;

- **Funções** são relações especiais onde o n-ésimo elemento da relação é único para os n-1 elementos precedentes; formalmente, funções são definidas como F: C1 x C2 x ,..., x Cn-1 ? Cn; são exemplos de funções antecedente-de e causa, indicando que o valor do segundo componente da relação depende do primeiro;

- **Axiomas** modelam sentenças que são sempre verdadeiras. São classificados em estruturais e não estruturais;

- **Instâncias** representam elementos específicos da ontologia, ou seja, os próprios dados.

2.4.2 Vantagens do Uso de Ontologia

O uso de ontologias na área da Ciência da Computação apresenta uma série de vantagens, tais como:

- Ontologias fornecem um vocabulário para **representação do conhecimento**. Esse vocabulário tem por trás uma conceitualização que o sustenta, evitando ambigüidade de interpretações desse vocabulário.

- Ontologias permitem o **compartilhamento de conhecimento**. Uma ontologia que modele adequadamente certo domínio de conhecimento pode ser compartilhada e usada por pessoas que desenvolvam aplicações dentro desse domínio. Considere que exista uma ontologia para o domínio de livrarias, várias livrarias podem construir seus catálogos usando o vocabulário fornecido por essa ontologia sem a necessidade de refazer uma análise do domínio de livraria.

- Fornece uma **descrição exata do conhecimento**. Uma ontologia pode ser escrita em linguagem formal não deixando espaço para o gap semântico existente na linguagem natural. Isto melhora a comunicação entre duas pessoas, se essas

pessoas concordam em uma ontologia sobre um domínio qualquer, evitando um mal entendido.

- É possível fazer o **mapeamento da linguagem da ontologia** sem que com isso seja alterada a sua conceitualização, ou seja, uma mesma conceitualização pode ser expressa em várias línguas.

- Pode ser possível estender o **uso de uma ontologia genérica** de forma a que ela se adeqüe a um domínio específico. Por exemplo, se alguém precisa de uma ontologia sobre bicicletas para construir uma aplicação e só encontra uma ontologia sobre o domínio genérico de veículos, pode utilizar essa ontologia estendendo-a para o domínio específico da aplicação, no caso de bicicletas.

O uso de ontologias provê benefícios na análise e especificação de requisitos de um sistema citados em PEREZ and BENJAMINS (2006):

- **Na comunicação**: uma ontologia é uma estrutura de representação de conhecimento útil para ajudar as pessoas a se comunicarem sobre um determinado domínio. Pessoas envolvidas no processo se veêm diante de um desafio de explicar seu entendimento sobre o domínio em questão, o que as faz refletir e melhorar sua compreensão sobre esse domínio.

- **Na formalização**: o uso de notação formal na especificação de um domínio utilizando ontologias elimina possíveis contradições e inconsistências, diminuindo assim ambiguidades da especificação.

- **Na representação do conhecimento e reuso**: uma ontologia representa um vocabulário de consenso e especifica conhecimento de domínio de forma explícita no seu mais alto nível de abstração com enorme potencial para o reuso. As ontologias providenciam uma terminologia clara reduzindo as confusões terminológicas e conceituais de termos utilizados na análise de requisitos.

A Análise de Domínio evita que o processo de entendimento e a codificação do conhecimento seja refeito para toda aplicação construída num mesmo domínio. Desta

maneira evita a exposição a erros e inconsistências que já foram resolvidas. Evitando perda de tempo, esforço e conseqüentemente recursos.

Modelos de domínio, produtos da análise, são usados pela comunidade de reuso de software como especificações em alto nível de abstração. Estes modelos são uma formulação genérica o suficiente para representar um conjunto de problemas, conhecimentos ou atividades similares. Uma das formas de obter e representar esses conhecimentos baseia-se em Ontologias.

Segundo GRUBER (1993), "Uma ontologia é uma especificação formal e explícita de conceitos compartilhados". A utilização de ontologias nessa etapa do processo de desenvolvimento traz vantagens como: melhoria da comunicação entre as pessoas envolvidas, devido ao consenso sobre o vocabulário e os significados dos termos num domínio; formalização do conhecimento, já que a especificação do domínio em ontologias elimina contradições e inconsistências; e, principalmente, a representação do conhecimento para reuso, já que a ontologia descreve o conhecimento do domínio de forma explícita no seu mais alto nível de abstração, possibilitando especializar o conhecimento durante o desenvolvimento de diferentes aplicações num domínio, que tenham propósitos variados, sejam criados por equipes distintas e em diferentes momentos.

2.4.3 Exemplo do Uso de Ontologia

SOUSA and LEITE (2004) apresentam uma ontologia para a melhoria do processo de busca e recuperação de componentes de interface de usuário (IU), onde será possível classificar componentes de IU baseando-se também em aspectos de interação. É de fundamental importância o conhecimento do modelo de interação do componente de forma que o desenvolvedor possa prever aspectos de sua usabilidade. A Figura 2.21 mostra uma ontologia representando a interação entre o usuário e os componentes de IU.

A ontologia permitirá buscas por componentes de IU que ofereçam determinados tipos de interação. Por exemplo, o desenvolvedor de IU pode procurar componentes que tenham um campo de inserção de dados seguido de uma caixa de seleção

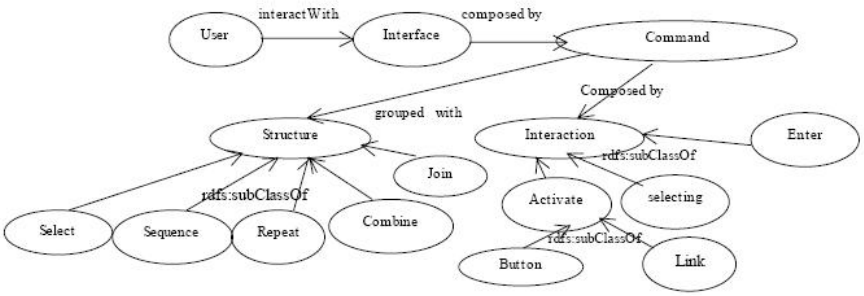


Figura 2.21: Ontologias representando a interação entre o usuário e os componentes de IU.

Fonte: (SOUSA and LEITE, 2004)

e de um botão.

A Figura 2.22 mostra que um pacote possui uma propriedade chamada *hasComponent* cujo valor é um componente. O componente, por sua vez, pode ter duas propriedades chamadas de *hasInterface* e *hasSignature*, onde estas têm como valor uma interface e a assinatura do componente respectivamente. Definimos que a assinatura de um componente possui métodos, eventos e atributos. A ontologia também descreve sinônimos para os conceitos. Por exemplo, o conceito *package* possui dois sinônimos que são *directory* e *library*. A descrição do modelo de interação através de

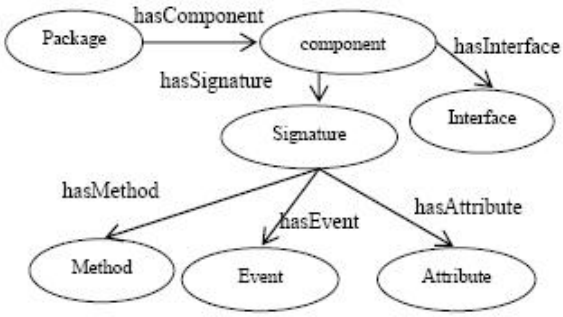


Figura 2.22: Ontologia das características estruturais dos componentes de IU.

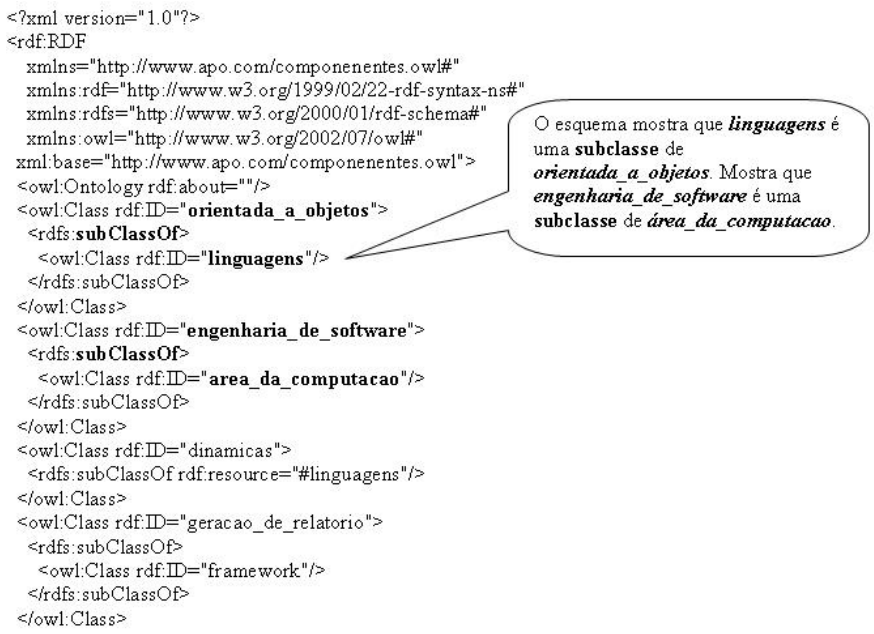
Fonte: (SOUSA and LEITE, 2004)

uma ontologia pode permitir a busca e a recuperação de forma mais eficiente, considerando as necessidades de interação e contribuindo para a usabilidade da aplicação. Por exemplo, podemos ter dois componentes de IU que tenham os mesmos métodos,

eventos e propriedades, mas que apresentem diferentes mecanismos de interação com o usuário.

A ontologia melhora o processo de busca, pois permite a recuperação utilizando aspectos semânticos relacionados à interação. Além disso, ela possibilita a criação de mecanismos de inferência que permite uma busca inteligente, verificando termos sinônimos e consultas em linguagem natural.

A Figura 2.23 mostra um outro exemplo onde se tem parte do esquema XML gerado pelo software Protégé descrevendo classificação hierárquica para a ontologia de componentes proposta neste trabalho de dissertação.




```
<?xml version="1.0"?>
<rdf:RDF
  xmlns="http://www.apo.com/componenentes.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
 xml:base="http://www.apo.com/componenentes.owl">
 <owl:Ontology rdf:about=""/>
 <owl:Class rdf:ID="orientada_a_objetos">
  <rdfs:subClassOf>
   <owl:Class rdf:ID="linguagens"/>
  </rdfs:subClassOf>
 </owl:Class>
 <owl:Class rdf:ID="engenharia_de_software">
  <rdfs:subClassOf>
   <owl:Class rdf:ID="area_da_computacao"/>
  </rdfs:subClassOf>
 </owl:Class>
 <owl:Class rdf:ID="dinamicas">
  <rdfs:subClassOf rdf:resource="#linguagens"/>
 </owl:Class>
 <owl:Class rdf:ID="geracao_de_relatorio">
  <rdfs:subClassOf>
   <owl:Class rdf:ID="framework"/>
  </rdfs:subClassOf>
 </owl:Class>
```

Figura 2.23: Esquema XLM classificação ontológica.

2.4.4 Ferramentas para Construção de Ontologias

Nesta seção são citadas suscintamente algumas ferramentas que podem auxiliar na construção de ontologias, o que permite a consistência e o aumento de produtividade durante o processo de elaboração. Enumeramos algumas ferramentas a seguir:

- **CODE4:** (Conceptually Oriented Description Environment) Ferramenta genérica que possui diferentes modos de herança e de inferências, interface gráfica, modo

de hipertexto para navegação e utilitários para leitura de documentos e gerenciamento léxico.

- **VOID:** Ambiente para navegação, edição e gerenciamento de ontologias. Através de simulações possibilita o estudo de questões teóricas como organização de bibliotecas de ontologias e tradução entre diferentes formalismos.

- **IKARUS** (Intelligent Knowledge Acquisition and Retrieval Universal System): Explora as capacidades cooperativas do ambiente Web. Utiliza uma representação hierárquica gráfica que permite herança múltipla. As declarações que contém a informação são representadas como predicados com sintaxe e semântica definidos ou como fragmentos sem estrutura.

- **Ontolingua**: Conjunto de serviços que possibilitam a construção de ontologias compartilhadas entre grupos. Permite acesso a uma biblioteca de ontologias, tradutores para linguagens e um editor para criar e navegar pela ontologia. Editores podem remotamente editar ontologias através de protolocos.

- **Ontosaurus**: Consiste de um servidor de ontologias que usa a linguagem LOOM para representação do conhecimento e um servidor de navegação por ontologias que cria páginas HTML dinamicamente, além de apresentar a hierarquia da ontologia.

- **GKB-Editor** (Generic Knowledge Base Editor): Ferramenta para navegação e edição de ontologias através de sistemas de representação baseados em frames. Oferece interface gráfica onde os usuários podem editar diretamente a base de conhecimento e selecionar a parte de seu interesse.

- **JOE** (Java Ontology Editor): Ferramenta para construção e visualização de ontologias. Proporciona gerenciamento do conhecimento em ambientes abertos, heterogêneos e com diversos usuários. As ontologias são visualizadas como um diagrama entidade-relacionamento, como em um gerenciador de arquivos, ou como uma estrutura em árvore.

- **APECKS** (Adaptive Presentation Environment for Collaborative Knowledge Structuring): É um servidor de ontologias que permite trabalho cooperativo através da criação de ontologias específicas por usuários. Estas ontologias podem ser comparadas com outras e é possível a discussão sobre as diferenças e similaridades entre elas.

- **OilEd**: Editor de ontologias de código aberto que permite construir ontologias utilizando a linguagem OIL. Não é um ambiente completo para desenvolvimento de ontologias. Verificação da consistência e classificação automática da ontologia podem ser executadas pela ferramenta FaCT.

- **OntoEdit**: Ambiente gráfico para edição de ontologias, que permite inspeção, navegação, codificação e alteração de ontologias. O modelo conceitual é armazenado usando um modelo de ontologia que pode ser mapeado em diferentes linguagens de representação. As ontologias são armazenadas em bancos relacionais e podem ser implementadas em XML, FLogic, RDF(S) e DAML+OIL.

- **OCM** (Ontological Constraints Manager): É uma ferramenta para verificar a consistência de ontologias em relação a axiomas ontológicos. É composto por duas ferramentas de edição que possibilitam verificar a ocorrência de conflitos.

- **Protegé 2000**: É um ambiente interativo de código aberto para projeto de ontologias que oferece uma interface gráfica para edição de ontologias e uma arquitetura para a criação de ferramentas baseadas em conhecimento. A arquitetura é modular e permite a inserção de novos recursos.

- **WebODE**: Ambiente para engenharia ontológica que dá suporte à maioria das atividades de desenvolvimento de ontologias. A integração com outros sistemas é possível através da importação e exportação de ontologias representadas em linguagens de marcação.

- **WebOnto**: Ferramenta que possibilita a navegação, criação e edição de ontologias representadas na linguagem de modelagem OCML. Permite o gerenciamento

de ontologias por interface gráfica, inspeção de elementos, verificação da consistência da herança e trabalho cooperativo. Possui uma biblioteca com mais de cem ontologias.

- **Ontomarkup Annotation Tool**: Ferramenta baseada em ontologias para incorporar informações semânticas em documentos através de anotações. Contém um componente de marcação que permite a navegação e a marcação de partes relevantes, um componente que aprende regras a partir de exemplos e um componente de extração da informação.

- **Onto Annotate**: É uma ferramenta de anotação semi-automática que permite a coleta de informações de documentos e páginas da Web, criando novos documentos com meta-dados. Permite a anotação em documentos de HTML estático, MS-Word e MS-Excel.

- **Asium** (Acquisition of Semantic Knowledge Using Machine learning method): Auxilia um especialista na aquisição de conhecimento e semântica de textos. Possui uma interface amigável que auxilia na exploração dos textos e no aprendizado da semântica que não está nos textos, como por exemplo de uma ontologia, que representa os conceitos estudados no domínio.

- **Text-to-onto**: Proporciona um ambiente para o aprendizado e construção de ontologias a partir de textos. Os textos podem ser em linguagem natural ou formatados em HTML. O sistema é composto por um módulo de gerenciamento de textos e um extrator de informações. Os resultados são armazenados em XML.

2.4.5 Linguagens para Ontologias

Nos últimos anos foram criadas várias linguagens para a representação de ontologias, sendo a maioria delas baseadas em Lógica de Primeira Ordem. Nesta seção são citadas algumas linguagens com as suas principais características.

Knowledge Interchange Format (KIF) é uma lógica de primeira ordem monotônica, possuindo sintaxe simples e com algumas extensões para suportar um raciocinador de relações.

Ontolingua é uma linguagem de compartilhamento de ontologias, desenvolvida para permitir o projeto e a especificação de ontologias com semânticas lógicas baseadas em KIF.

Ontology Inference Layer (OIL) é uma camada de inferência e representação baseada na Web, que combina a utilização de modelagem de primitivas provenientes das linguagens baseadas em frames com a semântica formal e, ainda com serviços de raciocinador provenientes de lógicas de descrição.

Resource Description Framework(RDF) é a recomendação do World Wide Web Consortium(W3C). Constitui-se em uma arquitetura genérica de metadados que permite descrever semanticamente recursos no contexto Web.

RDF fornece as primitivas básicas para a criação de ontologias, incluindo relacionamentos de generalização entre as classes. Três exemplos de categorias de tipos de informação são: *conteúdo*, *dados* e *fatos*; cada uma sendo processada de forma diferente:

- Informação de conteúdo: geralmente é processada como um todo, por exemplo: exibir uma página HTML, reproduzir uma música, etc;
- Dados: são processados em partes, ou blocos: inserir um registro no banco de dados, ordenar uma lista de objetos, etc;
- Fatos: são itens de dados em forma de instruções. Um exemplo de fato é a frase *A cerveja está quente*. RDF é uma linguagem que permite a representação explícita de fatos.

RDF Schema é uma extensão semântica do RDF e seu objetivo é fornecer uma linguagem para descrição de vocabulários a partir do RDF. O *RDF Schema* fornece um *framework* que permite definir classes e propriedades que possam ser utilizadas para descrever classes, propriedades e outros recursos.

DAML+OIL: este padrão é a união de DARPA *Agent Markup Language* (DAML) e OIL e define uma série de construções específicas para representação de ontologias em RDF. O Consórcio W3C está atualmente construindo um padrão para representação de ontologias, o OWL (*Ontology Web Language*), que é amplamente baseado no DAML+OIL e deverá manter grande parte das suas construções.

Ontology Web Language (OWL) é uma linguagem que foi criada pelo Consórcio W3C a partir da revisão da linguagem DAML+OIL, que por sua vez foi criada a partir destas outras duas linguagens separadas, DAML e OIL. A OWL foi desenvolvida com base nas necessidades da Web Semântica. Ontologias desenvolvidas em OWL possuem as seguintes características:

- Podem ser distribuídas através de diferentes sistemas;
- Escalabilidade conforme necessidades da Web;
- Compatibilidade com padrões Web quanto à acessibilidade e internacionalização.

OWL adiciona novas possibilidades de expressão de vocabulário nas classes e propriedades, a partir das já existentes na camada RDFS; logo, OWL depende das primitivas em camadas anteriores e se utiliza das mesmas. Alguns exemplos de expressões próprias da OWL são entre outros: disjunção (relacionamento entre classes), cardinalidade e igualdade.

Existem três tipos de sublinguagens da OWL, variando de forma crescente conforme o grau de complexidade e expressividade:

- **OWL Lite**: suporta ontologias simplificadas que precisam apenas de uma classificação hierárquica e restrições simples. Por exemplo, suporta restrição de cardinalidade, mas apenas 0 ou 1. É voltada para a migração de tesauros e outras taxonomias.
- **OWL DL**: suporta maior expressividade e poder computacional (garante a computabilidade de todas as conclusões). Contém todos os construtores da OWL, porém estes podem ser utilizados sob determinadas limitações, por exemplo: uma classe pode herdar de várias classes (herança múltipla), mas não pode ser

uma instância de outra classe. A Lógica de Descrição é responsável pela base formal da OWL.

- **OWL Full**: possibilita expressividade máxima aliada à liberdade da sintaxe do RDF, mas sem garantias computacionais. Por exemplo, uma classe pode ser tratada simultaneamente como um conjunto de indivíduos (instâncias) e como um indivíduo propriamente dito. OWL Full permite a uma ontologia expandir o significado do vocabulário pré-definido (RDF ou OWL).

2.4.6 Mecanismos de Inferência

Uma inferência permite extrair novos conhecimentos e novas conclusões válidas a partir de um conjunto de premissas, ou seja, da existência das regras e restrições da ontologia.

Para realizar inferência sobre uma ontologia, deve-se utilizar algum tipo de mecanismo. As ferramentas Jena e Pellet são utilizadas para fazer inferência. As principais caracterísitcas destas ferramentas são:

- **Jena** é um *framework* para a linguagem de programação Java e permite a criação de aplicações semânticas, manipulando informações em RDF e OWL. Jena também é considerada uma API (Application Programming Interface), pois disponibiliza uma série de classes e métodos que podem ser utilizados pelo programador para manipulação das ontologias.

  A ontologia é transformada em um modelo abstrato de dados orientado a objetos, fazendo com que seus termos possam ser manipulados como objetos, o que permite a utilização da programação orientada a objeto, como por exemplo, pela linguagem JAVA.

  A arquitetura do Jena é dividida em três módulos principais: *Ontology Model* que contém todas as classes necessárias para manipular a ontologia; *Reasoner* que é o Motor de Inferência responsável pela realização de inferências sobre a ontologia e a *Base RDF Graph* que gera as sentenças na sintaxe RDF.

- **Pellet** é um motor de inferência de código aberto para Java e voltado para a OWL DL. É baseado nos algoritmos de *tableaux*, desenvolvidos para Lógica de Descrição e suporta toda a expressividade da OWL DL.

  Pellet pode ser utilizado em conjunto com o Jena, como um motor de inferência externo, por ser mais completo.

2.5 Agentes no Contexto de DBC

Uma ontologia, dentro da Ciência da Computação, segundo MOREIRA et al. (2004) é um sistema de conceitos expresso em uma linguagem de representação de conhecimento. Esse sistema captura o significado de um conceito por meio de seu posicionamento dentro do sistema. Ontologias são utilizadas para diversos fins, tais como: representação, reuso, compartilhamento, aquisição e integração de conhecimento; processamento de linguagem natural; tradução automática; comunicação de informação entre sistemas, agentes, empresas ou pessoas, recuperação de informação e especificação de software (MOREIRA et al., 2004). No caso específico deste trabalho, a ontologia será utilizada pelo agente de busca para ampliar a consulta. O agente de classificação que será projetado futuramente utilizará a ontologia para classificar o componente.

Para os sistemas de inteligência artificial, tudo que existe pode ser representado. Quando o domínio do conhecimento é representado formalmente, o grupo de objetos e o relacionamento entre eles são refletidos em um vocabulário representacional. Assim, no contexto de Inteligência Artificial, pode-se descrever a ontologia de um programa definindo um grupo de termos de representações. Em uma ontologia, definições estão associadas aos nomes de entidades no universo em discurso: conceitos, relações, funções. Ontologias comuns descrevem tarefas para um conjunto de agentes de modo que estes possam se comunicar sobre um domínio do discurso sem necessariamente operar em uma teoria global compartilhada.

Segundo GRUBER (1993), um agente pertence a uma ontologia se suas ações observáveis são consistentes com as definições da ontologia. De forma real, uma

ontologia comum define o vocabulário comum e as afirmações que são trocadas entre os agentes. As tarefas da ontologia usam este vocabulário compartilhado de uma maneira coerente e consistente. Os agentes que compartilham o vocabulário não necessitam compartilhar de uma base de conhecimento; cada um tem suas próprias funções e seu conhecimento, e um agente que faz parte de uma ontologia não é requerido a responder todas as perguntas que podem se formular neste vocabulário compartilhado.

Segundo WOOLDRIDGE and JENNINGS (1995), agentes são *"sistemas computacionais situados em um determinado ambiente e que são capazes de executar ações autônomas neste ambiente para cumprir os objetos para os quais foram projetados"*. Por ações autônomas entende-se que os agentes são capazes de realizar ação sem a intervenção de elementos externos. Os agentes podem possuir capacidades de interação, permitindo que atuem de forma colaborativa/competitiva, formando sociedades de agentes denominadas de sistemas multi-agentes. Os sistemas multi-agentes podem ser vistos como um passo adiante na evolução dos métodos e linguagens para desenvolvimento de sistemas, sendo uma evolução do paradigma de desenvolvimento de sistemas orientados a objetos, diferenciando desses, fundamentalmente pela autonomia e pela capacidade de interação social. Como benefício do uso de sistemas multi-agentes, obtêm-se sistemas com um maior desacoplamento entre os módulos e com maior possibilidade de reuso. Por outro lado, as técnicas e ferramentas para desenvolvimento de tais sistemas ainda não estão maduras, tornando a tarefa de desenvolvimento ainda muito complexa. Portanto a utilização de tais sistemas está restrita a sistemas cujos benefícios compensem, claramente, o esforço de desenvolvimento.

Segundo WOOLDRIDGE (2002) um sistema baseado em agentes deve ser utilizado em ambientes dinâmicos, complexos e com dados e controle distribuídos. Estas características podem ser encontradas em um ambiente formado por um conjunto de repositórios de componentes distribuídos fisicamente. Esse tipo de ambiente pode incorporar agentes para atuar nas tarefas de catalogação e buscas de componentes. A catalogação e buscas dos componentes pode ser realizada com o apoio de um vocabulário controlado como uma ontologia. Cada repositório teria um agente para cada papel e os agentes em um repositório poderiam trocar informações com os agentes de

outros repositórios durante a tarefa de localização de componentes.

2.6 Repositórios

Repositório de bens de software reutilizáveis é um ponto centralizado de acesso e armazenamento de bens reutilizáveis, ou seja, de uma solução para um problema de desenvolvimento de software. Mais especificamente, o repositório de componentes é um sistema que apóia a pesquisa, o fornecimento e o gerenciamento de componentes para a construção de aplicações de negócios. Repositórios armazenam, registram e gerenciam todos os artefatos produzidos durante todo o ciclo de vida dos componentes.

As principais funcionalidades de um repositório são as pesquisas e a recuperação dos ativos de software. EZRAN et al. (2002) propõe funções adicionais que um repositório deve disponibilizar para permitir um melhor gerenciamento e recuperação dos ativos de software:

- **Identificação e descrição**: para se descrever um ativo é possível que a ele seja atribuído um conjunto de características tais como, nome, domínio, palavras-chave, dentre outras que os identificam e os diferenciam dos demais ativos que compõem esse mesmo repositório. Ativos classificados homogeneamente devem apresentar o mesmo conjunto de características, mas contudo podem valores diferentes para esse conjunto de características.

- **Inserção**: usuários autorizados podem insiram novos ativos. A inserção significa adicionar ao repositório o corpo e a descrição do ativo. É de suma importância que usuários autorizados tenham uma visão geral dos conjuntos de informações para evitar classificações incorretas e redundantes.

- **Exploração do catálogo**: usuários podem explorar o catálogo de ativos para que possam conhecer e analisar as características dos ativos disponíveis.

- **Pesquisa Textual**: um repositório deve permitir que seus usuários façam pesquisas mais específicas na descrição dos ativos. Como resultado da pesquisa

serão obtidos um ou mais ativos que satisfaçam as condições desejadas. Observados os resultados, pode-se decidir por um maior detalhamento ou generalização dos critérios anteriores.

- **Recuperação**: um repositório deve permitir que seus usuários recuperem ativos já classificados para que possam posteriormente utilizá-lo num processo de reuso.

- **Organização e pesquisa**: repositórios podem adotar novas formas de organizar as características dos ativos de modo a permitir que a pesquisa no repositório seja baseada em outros critérios a não se o nome.

- **Histórico**: é importante para o gerenciamento de um repositório que ele possa armazenar informações de uso, modificações, criação e exclusão de cada um dos ativos disponíveis. Essas informações devem montar uma base histórica que facilitará a análise e a reutilização dos mesmos.

- **Mensuração**: um repositório pode coletar estatísticas que facilitem o seu gerenciamento. Algumas das principais estatísticas que podem ser adotadas são: frequência de acesso ao repositório, quantidade de ativos disponíveis, taxa de recuperação, porcentagem de pesquisas bem-sucedidas, frequência com que um determinado ativo é analisado ou modificado, dentre outras.

- **Controle de acesso**: o repositório pode adotar uma política de segurança para que determinadas funcionalidades só estejam acessíveis a pessoas autorizadas. Por exemplo, pode-se definir para uma empresa específica uma política de segurança onde a pesquisa textual esteja disponível para todos os funcionários, mas a recuperação de ativos esteja disponível apenas aos funcionários da área de desenvolvimento.

- **Gerenciamento de versões**: um repositório pode conter várias versões de um mesmo ativo e, sendo assim, é recomendável que haja algum mecanismo para controlar essas versões e estabelecer o relacionamento entre elas.

- **Controle de modificações**: Um ativo é projetado para o reuso. É de suma

importância algumas funcionalidades para se fazer o gerenciamento de modificações dos ativos num repositório. Um repositório deve prover funções que notifiquem os seus usuários das modificações recentemente ocorridas tais como, inserção ou exclusão de ativos, alterações em documentos, disponibilização de novas funcionalidades, novas políticas de segurança, entre outras.

- **Quantidade de repositórios**: o repositório pode ser único ou pode existir um conjunto deles. O **único** é indicado para organizações pequenas onde os tipos de ativos não são completamente distintos e são compreendidos por todos os seus usuários. **Vários** repositórios são aconselhados para organizações maiores ou geograficamente distribuídas, cada um relacionado a uma área de atuação da empresa. É necessário um gerenciamento centralizado e eficiente para evitar inconsistência nas duplicações existentes entre alguns repositórios.

- **Acesso pela rede**: um requisito típico é que o repositório possa ser acessado de qualquer ponto numa rede, uma vez que seus usuários estão distribuídos geograficamente. O que normalmente se tem observado na implementação dos repositórios é a utilização de modelos cliente-servidor em 3 (três) ou mais camadas, distribuindo as atividades em cada uma das camadas, tais como: uma camada é a responsável pelo armazenamento dos ativos, a outra pelo processamento das principais funcionalidades do sistema (pesquisa e recuperação) e a terceira corresponde à aplicação cliente, por onde o repositório poderá ser acessado.

Lucrédio (LUCREDIO et al., 2004) cita alguns requisitos essencias para a especificação de um mecanismo de busca e recuperação de componentes de software:

- **Elevada precisão e recuperação**. Elevada precisão significa que a maioria dos componentes recuperados são relevantes.

- **Segurança**. Somente pessoas autorizadas podem acessar o repositório.

- **Formulação de Consultas**. Existe uma lacuna entre o problema e a solução, já que componentes são descritos em termos de sua funcionalidade (**"como"**)

e as consultas em termos de solução (**"o que"**), assim o mecanismo de busca provê meios a auxiliar o usuário na formulação das consultas.

- **Descrição do componente**. O mecanismo de busca é responsável por identificar os componentes relevantes para o usuário de acordo com a consulta formulada e executada em cima das descrições dos componentes.

- **Familiaridade no repositório**. O mecanismo de busca deve auxiliar o usuário a explorar e recuperar componentes familiares ao que era o alvo inicial, podendo expandir o nível de busca inicial, facilitando buscas e estimulando a concorrência entre os fornecedores de componentes.

- **Interoperabilidade**. O mecanismo de busca que funciona neste cenário deve ser baseado em tecnologias que facilitam sua futura expansão e integração com outros sistemas operacionais e repositórios.

- **Desempenho**. É usualmente medido em termos de tempo de resposta. Sistemas centralizados envolvem variáveis relacionadas ao poder de processamento e algoritmos de busca. Em sistemas distribuídos as variáveis como: controle de tráfego da rede, distância geográfica e o número de componentes disponíveis devem ser consideradas.

2.6.1 Classificação e Recuperação

Repositórios podem classificar componentes através de seus recursos e objetivos. Diversos mecanismos são utilizados para a classificação, dentre os quais, pode-se destacar a utilização de palavras-chave, casamento de assinaturas, introspecção, hierarquia de tópicos e "ranqueamento", em que componentes mais usados são prioritariamente recuperados. Em relação às pesquisas, existem basicamente dois modelos: o passivo, em que o usuário inicia a busca; e o ativo, quando o ambiente de desenvolvimento inicia a busca dos componentes.

2.7 Trabalhos Relacionados

AGORA (SEACORD et al., 1998): é um repositório desenvolvido pelo Software Engineering Institute (SEI), cujo objetivo é facilitar a busca por componentes, através de uma base de dados distribuída, gerada e indexada automaticamente, e de amplitude mundial. Agentes de software vasculham na internet para descobrir e coletas recursos, buscando URL da localização dos mesmos.

SCM RAGNAROK (CHRISTENSEN, 1999): é um sistema de gerência de configuração, com suporte a elementos de arquitetura, onde o componente é a unidade central. O usuário entende o repositório como um conjunto de componentes ao invés de um conjunto de arquivos e diretórios.

Os processos de adição de artefatos e recuperação de artefatos têm um comportamento influenciado pela arquitetura do software alterado. Ao incluir uma nova versão de um artefato, a arquitetura é considerada, e todos os componentes relacionados são por sua vez versionados. Verificações de consistência de alterações precedem o check-in de uma nova versão.

O Ragnarok se preocupa mais com a evolução da arquitetura e não oferece algumas funcionalidades importantes para DBC como: pesquisas de componentes, possibilidade de integração com IDE ou um modelo de metadados apropriado para componentes.

WREN(LER and ROSENBLUM, 2001): é um protótipo de ambiente para DBC, baseado em Java e JavaBeans, com um repositório integrado. No ambiente WREN, os componentes contêm a sua própria descrição, evitando problemas como: perda de informações contidas em arquivos textos e dificuldade em atualizar e sincronizar manualmente descrições e implementações. A descrição do componente não inclui somente um texto, mas também parte da especificação do componente. Detalhes sobre suas interfaces requeridas e providas são armazenados, como por exemplo: sintaxe, semântica, qualidade de serviço, e descrição da interface. Estas informações estão disponíveis de forma estruturada para que o ambiente consiga utilizá-las para realizar verificações e validações em configurações arquiteturais de componentes. Todos

os componentes contém classes que provêem métodos que fornecem estas informações.

Outro problema abordado pelo ambiente WREN é a identificação de interface de componentes. Interfaces devem apresentar nomes distintos e globais de modo que não haja ambigüidade em suas referências. Para este problema é sugerido o uso de "espaços de nomes" globais, que garantem que o nome completo da interface seja único. Em Java, quando se utiliza a convenção de nomear pacotes com o inverso do domínio da Internet, o nome do pacote concatenado com o nome da interface é um exemplo de "espaço de nome" global.

Ao utilizar um componente recuperado de um repositório, o WREN permite que seja mantida uma referência para o repositório de origem do componente. Esta referência é utilizada para monitorar as evoluções do componente recuperado. Quando o componente é alterado no repositório original, o WREN automaticamente recupera o componente novamente. Este tipo de relacionamento entre o ambiente DBC e o repositório é chamado de reuso por referência. Uma das vantagens é que existe apenas uma cópia mestra do componente no repositório de origem.

As interfaces WREN são versionadas, mas o versionamento é linear: não existem versões concorrentes. O repositório WREN permite buscas usando palavras-chave em linguagem natural. O ambiente WREN não oferece suporte a um formato de metadados padronizado.

CODE BROKER (YE, 2001): é um repositório para componentes de propósito geral, com busca ativa de componentes, na qual buscas são iniciadas automaticamente pelo ambiente de desenvolvimento.

A indexação e busca de componentes no sistema CodeBroker baseia-se em textos livres, tais como artigos de jornais e livros. Nestas tarefas são utilizados dois modelos: um modelo probabilístico e um modelo baseado na semântica de associações.

O CodeBroker baseia-se em buscas ativas, que estão integradas a um ambiente de desenvolvimento. Entretanto, o CodeBroker não dá suporte a integração com sistemas GCS.

OSCAR (BOLDYREFF et al., 2002): é um repositório com a finalidade de armazenar e gerenciar artefatos de software de propósito geral. Ele é um componente

da plataforma GENESIS (GEneralized eNvironment for procESs management in cooperatIve Software engineering), que tem como objetivo prover suporte a questões relacionadas ao desenvolvimento distribuído de software, oferecendo gerenciamento de *workflow* e suporte a processos para equipes localizadas em diferentes organizações. Suporte a controle de versões através da integração com o sistema GCS-CVS. É um repositório distribuído. Os metadados do OSCAR não seguem um formato padronizado, que facilitaria uma integração com outros repositórios e ferramentas de desenvolvimento.

CKBR-P (VITHARANA et al., 2003): é um protótipo de um repositório que aborda principalmente problemas relacionados à classificação e busca de componentes.

O repositório implementa um esquema que divide as informações dos componentes em identificadores estruturados, os quais classificam informações bem conhecidas, e identificadores semi-estruturados que classificam diversas facetas do componente, como funcionalidade, regras de negócio, etc.

As facetas propostas são:

- Sinônimos de nomes de componentes, métodos, etc.
- Papel (Role).
- Regras de negócio.
- Funcionalidade / Tarefa.
- Elemento / Parte (composição do componente).
- Ação / Evento.
- Usuário (usuários do componente).

Os identificadores estruturados são armazenados em um banco de dados, ao passo que os identificadores semi-estruturados são armazenados em arquivos XML. Dados semiestruturados são dados com uma organização que pode mudar ou que apresenta uma irregularidade.

A pesquisa de componentes acontece em dois estágios. No primeiro estágio, a pesquisa utiliza apenas os parâmetros estruturados, devido a sua eficiência, para fazer uma pré-seleção de resultados. No estágio seguinte, acontece o refinamento da pesquisa a partir dos resultados selecionados no estágio anterior. Os parâmetros semi-estruturados permitem um maior nível de detalhe na busca de componentes.

O repositório é dividido em vários sub-repositórios mutuamente exclusivos de acordo com identificadores estruturados, para otimizar a busca de componentes.

A pesquisa de componentes é implementada de modo a apoiar um processo incremental de seleção de componentes. Inicialmente, a tendência é encontrar um grande número de componentes, através da adição de novos critérios de seleção, o resultado da pesquisa é cada vez mais refinado, selecionando-se um número menor de componentes.

Questões como interoperabilidade e padronização no empacotamento de bens não são abordadas pelo CKBR-P.

Odyssey-SCM O Odyssey (MURTA, 2004) é uma ferramenta que utiliza técnicas de Engenharia de Domínio, Linha de Produtos e DBC. Linha de produtos é um conjunto de sistemas compartilhando características gerenciáveis comuns, que satisfazem às necessidades específicas de um segmento de mercado em particular e que são desenvolvidos sistematicamente, a partir de um conjunto comum de artefatos.

O Odyssey é composto de vários sub-projetos, sendo que entre eles, o Odyssey-SCM, é o sub-projeto do Projeto Odyssey que mais está relacionado às funcionalidades de um repositório de componentes.

Odyssey-SCM tem como objetivo fornecer uma abordagem de GCS customizada para apoiar DBC.

A abordagem Odyssey-SCM é composta por cinco sub-abordagens:

- Odyssey-SCMP (SCMP: Software Configuration Management Process): Processos, normas, procedimentos, políticas e padrões de GCS para o contexto de DBC.

- Odyssey-CCS (CCS: Change Control System): Sistema de controle de modifi-

cações configurável e extensível.

- Odyssey-VCS (VCS: Version Control System): Sistema de controle de versões baseado em políticas com suporte aos diversos níveis de abstração dos artefatos que descrevem componentes, interfaces e conectores.

- Odyssey-BRCS (BRCS: Build and Release Control System): Sistema de controle de construções e liberações orientado a arquitetura de componentes.

- Odyssey-WI (WI: Workspace Integration): Integração dos espaços de trabalho de GCS e DBC.

Rigel: PINHO (2006) apresenta um repositório de componentes, denominado de Rigel, baseado nas extensões do modelo RAS propostas por GUERRA et al. (2005). As extensões RAS propostas por Pinho são bem abrangentes, mas não apresentam metadados para o registro de informações relacionadas com estatísticas de uso e satisfação do usuário. Estas informações são importantes para construção de ferramentas eficazes de ranqueamento de componentes. Além disso, o interfaceamento com o repositório Rigel baseia-se no protocolo de invocação remota da linguagem Java (RMI), o que restringe a interação com outras ferramentas. A inexistência de uma ontologia também limita o uso de ferramentas, que pode recuperar conjuntos de ativos de software relacionados conceitualmente.

Capítulo 3

Proposta do uso de Agentes em Desenvolvimento Baseado em Componentes

3.1 Introdução

A proposta deste trabalho é desenvolver um conjunto de meta-informações para um repositório de ativos de software, tendo um enfoque principalmente em componentes de softwares, que possam disponibilizar seus ativos em uma Rede de Compartilhamento de Componentes de Software (RCCS).

As meta-informações contidas no repositório RAS não contemplam informações suficientes para orientar os agentes do sistema na pesquisa de componentes e, em vista disto, este trabalho visa desenvolver uma ontologia sobre componentes, que permita a extensão do conjunto das meta-informações dos repositórios, de forma a suportar a definição de um componente de software como o conjunto de meta-informações que ele deve possuir de forma estruturada.

O conjunto das meta-informações extendido permitirá, em trabalhos futuros, a especificação e o desenvolvimento de sistema multiagente com mecanismos e padrões bem definidos, onde os seus agentes possam: gerenciar, catalogar, organizar e pesquisar os componentes de software de forma a promover o reuso efetivo e aumentar a eficiên-

cia e produtividade do desenvolvimento de software baseado na utilização de componentes.

Partindo do repositório RAS que é o resultado do esforço da OMG em apoiar o Desenvolvimento Baseado em Componentes de Software. Do estudo de trabalhos relacionados que foram descritos sumariamente na Seção 2.7 e da experiência do mestrando, foram levantados alguns requisitos para a definição do meta-dados do repositório proposto. Meta-dados este, que é uma base de dados comum, no qual as informações dos componentes de software reutilizáveis estão armazenadas para compartilhamento.

3.2 Levantamento de Requisitos do Repositório

3.2.1 Missão

O Repositório de Ativos de Software visa oferecer um *conjunto de meta-informações dos ativos de software* que dê suporte para o desenvolvimento de sistema multiagente com mecanismos e padrões definidos, onde os seus agentes possam: gerenciar, catalogar, organizar e pesquisar os ativos de software. O objetivo é promover o reuso efetivo e aumentar a eficiência e produtividade no ambiente de desenvolvimento de software baseado nem componentes.

3.2.2 Atores

Neste novo contexto de Desenvolvimento de Software Baseados em Componentes encontramos novos papéis dos profissionais envolvidos no processo de desenvolvimento. A Tabela 3.1 faz uma definição desses profissionais que chamados de atores. Um mesmo profissional físico pode desempenhar mais de um papel, por exemplo um Desenvolvedor de Componente pode agir também como um Desenvolvedor de Software.

Tabela 3.1: Atores no contexto de desenvolvimento DBC.

Ordem	Ator	Definição
1	Desenvolvedor de Componentes	São responsáveis por fazer o levantamento de requisitos e pelo desenvolvimento dos componentes com enfoque em reuso. Disponibilizam para o Administrador do Repositório os componentes e suas versões para que o mesmo faça a catalogação e atualização do repositório.
2	Administrador do Repositório	É responsável pela classificação e catalogação dos componentes, pela validação dos novos componentes e versões submetidas, pela manutenção dos grupos de usuário e usuários, divulgação e permissão de acessos aos ativos de software e também pela liberação de acesso a funcionalidades do aplicativo.
3	Desenvolvedor de Software	Responsável por levantar requisitos do software e fazer a pesquisa por componentes que possam ser reusados no desenvolvimento. Podem também sugerir componentes para serem catalogados no repositório.

3.2.3 Benefícios

A Tabela 3.2 apresenta uma lista de benefícios levantados com o uso do repositório de ativos proposto na dissertação. Este levantamento se faz necessário para uma análise do custo e benefício dos investimentos na disponibilização de funções que atendam a estes benefícios e também na priorização dos requisitos funcionais a serem atendidos pelo repositório.

Tabela 3.2: Lista de Benefícios do uso Repositório.

Ordem	Benefício	Valor
1	Restrição de Acesso aos ativos de software.	Essencial
2	Maior agilidade no cadastro de ativos de software.	Essencial
3	Melhor divulgação de novos ativos ou versões, ou de problemas detectados.	Desejável
4	Melhor classificação dos ativos de software.	Essencial
5	Eliminação de duplicidades no cadastro.	Essencial
6	Maior agilidade na recuperação de ativos.	Essencial
7	Maior agilidade no atendimento das solicitações de adaptações dos ativos.	Desejável
8	Maior agilidade em liberar o acesso ao ativo para os usuários.	Desejável
9	Análise de Estatística de acesso de usuários e ativos mais utilizados e atualizações.	Desejável
10	Centralização dos ativos de software da corporação.	Essencial
11	Acesso aos ativos em rede ou web, ou seja, descentralizado.	Essencial

3.2.4 Casos de Uso

Os casos de uso representam funções completas que o repositório deve prover, ou seja, realiza um aspecto maior da funcionalidade que podem gerar um ou mais benefícios

Tabela 3.3: Lista de Casos de Uso do Repositório.

Ordem	Caso de Uso	Descrição
1	Gestão de Usuários	Controle do cadastro de grupos de usuários e usuários que manipular o repositório.
2	Gestão de Componentes	Processamento de inclusão, alteração e exclusão de ativos. Responsável também pela classificação.
3	Gestão de Recuperação	Recuperação das informações dos ativos cadastrados no repositório.
4	Gestão de Solicitação	Processamento de inclusão, alteração e exclusão de solicitação de adaptações dos ativos existentes e das anormalidades detectadas no uso do ativo.
5	Gestão de Estatística	Emissão de dados estatísticos de manipulação dos ativos de software e acessos dos usuários.
6	Gestão de Liberação de Ativos	Controle dos níveis de permissão dos grupos de usuários e usuários sobre o ativo.
7	Gestão de Acesso a Funções	Controle dos níveis de permissão que cada grupo de usuários e usuários têm sobre as funções dos aplicativos que manipulam o repositório.
8	Gestão de Divulgação	Divulgação de novos ativos presentes no repositório e/ou de novas versões dos ativos disponíveis para os usuários do repositório.

para os usuários do repositório. A Tabela 3.3 mostra a lista dos casos de uso.

3.2.5 Diagrama de Contexto

O Diagrama de Contexto é um importante Diagrama de Caso de Uso da UML, pois representa a interação dos atores com os casos de uso, ou seja, indica a comunicação existente entre os mesmos. O propósito primário dos casos de uso é o de descrever os requerimentos funcionais do sistema de maneira consensual entre os usuários e os desenvolvedores do sistema e fornecer uma descrição consistente e clara sobre as responsabilidades que devem ser cumpridas pelo sistema.

A Figura 3.1 representa o papel dos novos profissionais na abordagem DBC(Atores).

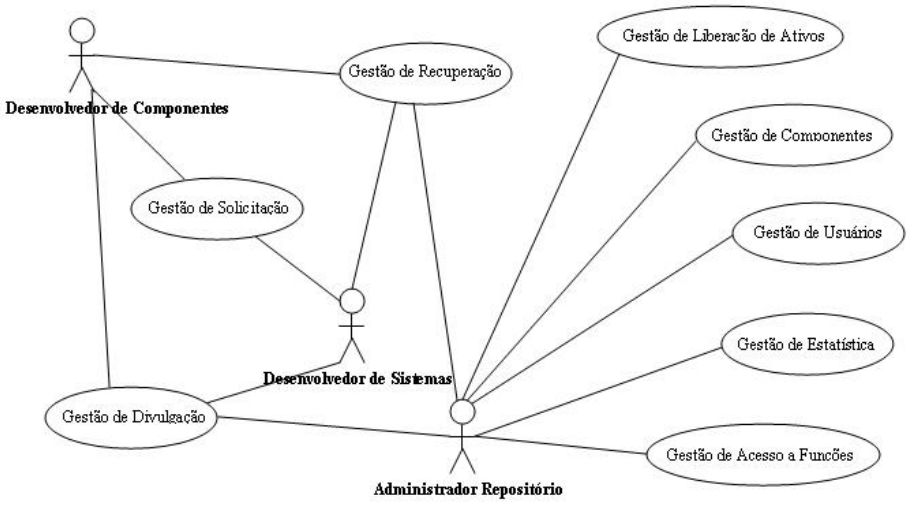


Figura 3.1: Diagrama de Contexto.

3.2.6 Lista de Funções

A Tabela 3.4 apresenta a lista das funções que o repositório deve prover. Estas funcionalidades são derivadas da lista de requisitos. A lista de requisitos deve ser contemplada por requisitos de alta qualidade, claros, completos, sem ambigüidade, implementáveis, consistentes e testáveis.

Tabela 3.4: Lista das Funções.

Num	Tipo	Nome Função	Descrição da Função
RF001	Cad	Cadastro Grupo Usuários	Permite a inclusão, alteração, exclusão e desativação de grupo de usuários.
RF002	Cad	Cadastro Usuários	Permite a inclusão, alteração, exclusão e desativação de usuários
RF003	Con	Vínculo de Usuários a Grupo Usuários	Permite a vinculação de usuários ao grupo de usuários.
RF004	Cad	Cadastro de Grupo Descritores	Permite a inclusão, alteração e exclusão de grupo descritores.
RF005	Cad	Cadastro de Descritores	Permite a inclusão, alteração e exclusão de descritores.
RF006	Cad	Cadastro de Ativo	Permite a inclusão, alteração e exclusão ativos
RF007	Cad	Cadastro de Versão de um Ativo	Permite a inclusão, alteração e exclusão de versão de ativo.
RF008	Rel	Recuperação de um Ativo	Permite a pesquisa de ativo
RF009	Con	Liberação acesso a funções aplicativo	Permite vincular permissão de acesso ao repositório a grupo de usuários.
RF010	Con	Liberação acesso a um ativo.	Permite a inclusão, alteração e exclusão de permissão de acesso a um determinado ativo.
RF011	Cad	Registro de solicitação adequação de ativo	Permite a inclusão, alteração e exclusão da solicitação de adequação de um ativo
RF012	Rel	Divulgação Ativo ou Versão	Permite divulgação de novo ativo ou alteração de versão de ativo do repositório

Abreviaturas: **Cad** - Cadastro, **Con** - Controle, **Rel** - Relatório

3.3 Ontologia

No caso específico deste trabalho a ontologia será utilizada pelo agente de busca para ampliar o escopo da consulta. O agente de classificação que será projetado futuramente utilizará a ontologia para classificar o componente.

Como foi citado na Seção 2.4.5, existem várias linguagens para a representação de ontologias, sendo que atualmente destaca-se a **OWL** (*Web Ontology Language*) (SMITH et al., 2004) que é uma linguagem baseada nas linguagens **OIL** (*Ontology Interchange Language*) e **DAML** (*DARPA agent markup language*) e é a linguagem recomendada pela **W3C1**. Por ser um padrão, existem diversas ferramentas para auxiliar a criação de ontologias no formato **OWL**. No caso deste trabalho a ferramenta utilizada foi o software *Protégé* (versão 3.3.1).

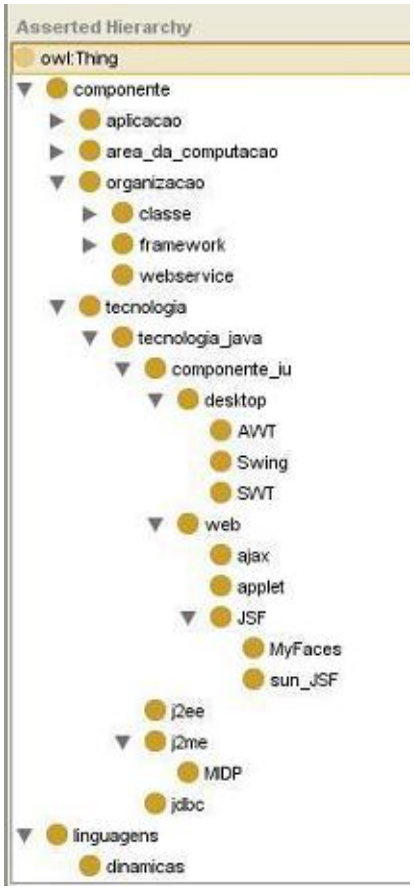


Figura 3.2: Segmento da Ontologia de Componentes Desenvolvida.

A Figura 3.2 mostra um segmento da ontologia de componentes proposta neste

trabalho. A ontologia apresenta um caráter multi-facetado de acordo com a proposta de (DIAZ, 1991) *apud* (LUCREDIO et al., 2004). Isto significa que o componente pode ser classificado sob diferentes ângulos. Por exemplo, um componente para algoritmo genético pode ser classificado tanto como um componente da área de inteligência artificial como um componente da área de otimização.

Um dos benefícios de se usar padrão como **OWL** é poder contar com diversos mecanismos de inferência sobre lógica de descrições para raciocinar sobre a ontologia. O *Protégé* fornece uma **API** para comunicar com alguns desses raciocinadores, como o **RACER** (www.sts.tu-harburg.de/ r.f.moeller/racer/), o **FACT++** (owl.man.ac.uk/factplusplus) e o raciocinador **Pellet** (pellet.owldl.com/). Neste trabalho, foi utilizado o raciocinador *Pellet* (SIRIN et al., 2007) por ser de código aberto e escrito em *Java*, o que permite uma fácil integração com o sistema desenvolvido. *Pellet* é um raciocinador para lógica de descrições baseado em algoritmos de *tableaux*, e suporta toda a expressividade da **OWL** DL.

A Figura 3.3 mostra o arquivo da ontologia especificada na dissertação, gerada pelo *Protégé*, em formato OWL. Ela mostra a hierarquização das classes e o caracter multi-facetado onde a classe **algoritmos-geneticos** é uma classe que é subclasse da classe **programacao-evolutiva** e também da classe otimizacao. Portanto, quando um componente for classificado como sendo da classe **algoritmos-geneticos** e fizermos uma expansão da recuperação e/ou pesquisa de componentes das classes de nível superior, serão selecionados todos os componentes da classes **programacao-evolutiva** e **otimizacao**.

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns="http://www.apo.com/componenentes.owl#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://www.apo.com/componenentes.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="orientada_a_objetos">
   <rdfs:subClassOf>
    <owl:Class rdf:ID="linguagens"/>
   </rdfs:subClassOf>
  </owl:Class>
  </owl:Class>
  <owl:Class rdf:ID="dinamicas">
   <rdfs:subClassOf rdf:resource="#linguagens"/>
  </owl:Class>
  <owl:Class rdf:ID="geracao_de_relatorio">
   <rdfs:subClassOf>
    <owl:Class rdf:ID="framework"/>
   </rdfs:subClassOf>
  </owl:Class>
          ●
          ●
          ●
<rdfs:subClassOf rdf:resource="#linguagens"/>
  </owl:Class>
  <owl:Class rdf:ID="algoritmos_geneticos">
   <rdfs:subClassOf rdf:resource="#programacao_evolutiva"/>
   <rdfs:subClassOf rdf:resource="#otimizacao"/>
  </owl:Class>
  </owl:Class>
  <owl:Class rdf:ID="MVC">
   <rdfs:subClassOf rdf:resource="#aplicacao"/>
</rdf:RDF>
          ●
          ●
          ●
<!-- Created with Protege (with OWL Plugin 3.2.1, Build 365)
http://protege.stanford.edu -->
```

Figura 3.3: Parte do arquivo OWL da Ontologia Componentes Desenvolvida

3.4 Extensão do RAS

Algumas características e abordagens definidas pelos *profiles* do modelo RAS possuem limitações, tais como: o *Default Component Profile* que organiza os artefatos dos ativos apenas considerando o processo de desenvolvimento *UML Components* e o do *Default Component Profile* em não permitir o reuso de uma descrição de interface por mais de uma descrição de componente. O RAS também não descreve as interfaces requeridas, as propriedades, os eventos e as exceções, que são características presentes nos principais modelos de componentes. O RAS também não dispõe de meios para descrever os modelos de negócios, evoluções entre versões de um ativo, requisitos não-funcionais e certificação. Estas informações poderiam auxiliar a um sistema multiagente na busca e recuperação de componentes.

Embora o RAS possua tais restrições, as mesmas podem ser contornadas com a definição de novos *profiles* que solucionam as limitações apresentadas.

Considerando-se os requisitos levantados na Seção 3.2, são propostas a inclusão de novas classes ao meta-modelo RAS para incorporar informações complementares no repositório de ativos de software com a finalidade de atender os requisitos levantados. Na próxima Seção 3.4.1, são descritas as novas classes propostas, agrupadas por funcionalidade propostas pelos casos de uso da Seção 3.2.4.

3.4.1 Extensão para uso da ontologia de classificação de componentes

A ontologia de componentes de software apresenta um caráter multi-facetado e foi criada utilizando-se o software Protégé, que se baseia na linguagem OWL, conforme é descrito na seção 3.3.

A extensão especificada na dissertação baseia-se na ontologia de componentes e após ser feito o mapeamento da ontologia *OWL*, foram acrescidos ao modelo *RAS* as seguintes classes: *ClassOntology*, *ClasssubOntology* e *AssetClassontology*. A Figura 3.4 mostra o diagrama de classe da extensão.

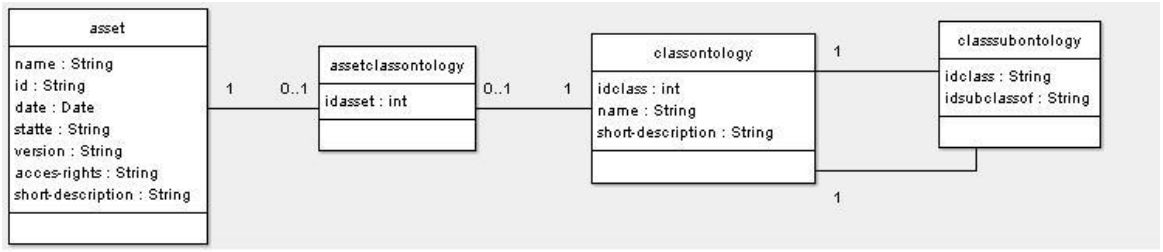


Figura 3.4: Diagrama de Classe da Extensão Proposta.

A classe *ClassOntology* instancia todas as classes da classificação ontológica. Possui os atributos: *idclass*, *name* e *short-description*. O atributo *idclass* identifica univocamente cada uma das classes. O atributo *name* identifica o nome da classe em poucas palavras. O atributo *short-description* descreve mais detalhadamente a funcionalidade da classe.

A classe *Classsubontology* associa uma classe a todas as suas subclasses da hierarquia de classificação. Possui os atributos *idclasse* e *idsubclassof*. O atributo *idclasse* identifica uma classe e o atributo *idsubclassof* identifica uma classe da qual a classe é uma subclasse. Uma classe pode ser subclasse de uma ou mais classes.

A classe *AssetOntology* associa um ativo a todas as classes a que pertence segundo a classificação ontológica. O atributo *idasset* identifica o ativo. O atributo *idclass* identifica a classe associada a que o ativo pertence. Esta associação possibilita que ao ser localizado um determinado componente através de uma consulta

simples ao seu nome ou com alguns outros critérios, possamos através de inferência expandir a busca por outros componentes da mesma categoria, ou das categorias imediatamente superiores da categoria do componente solicitado (expandindo a busca) ou ainda das categorias imediatamente inferiores da categoria do componente solicitado (restringindo a busca). A Figura 3.5 mostra um formulário de um aplicativo cliente-servidor, que através de comandos SQL faz uma busca à base de dados relacional do repositório e que permite a busca semelhantemente à utilização de uma máquina de inferência.

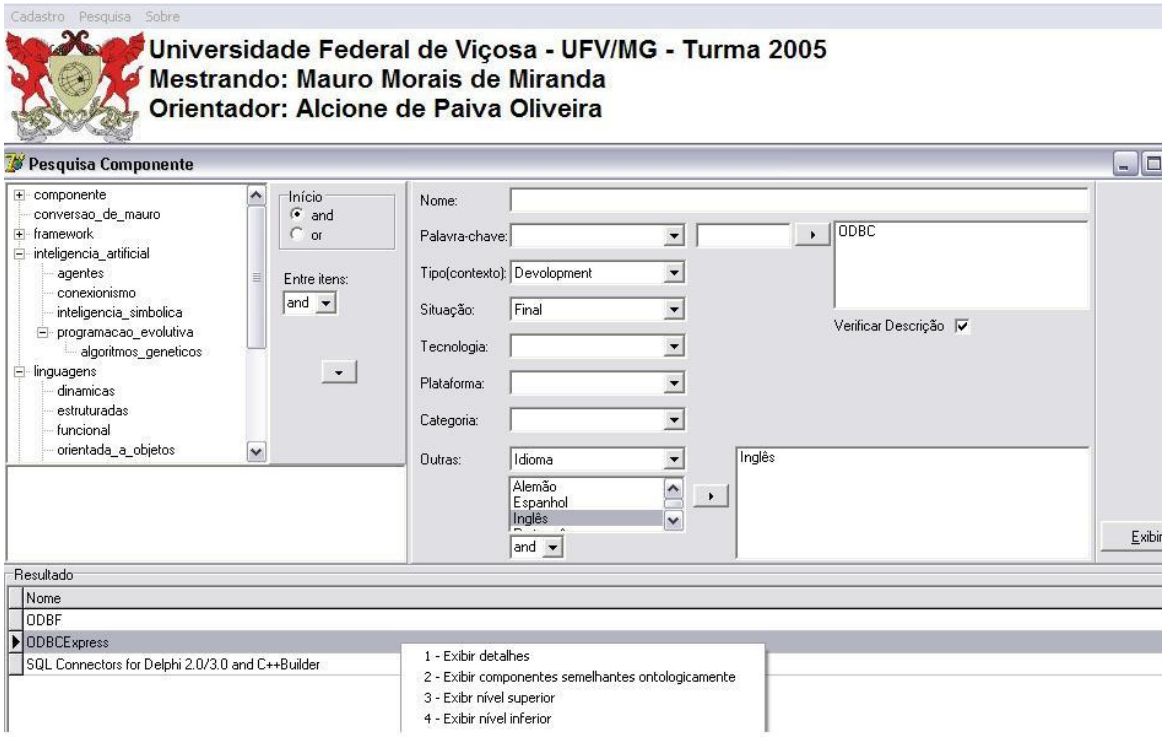


Figura 3.5: Formulário para Pesquisa de Componentes.

3.4.2 Extensão para padronização de termos na classificação

A boa classificação do ativo é de suma importância para determinar um bom processo de busca e recuperação de ativos de software em um repositório.

O RAS utiliza-se da classe *Grupo Descritor* que é relacionada diretamente com a classe *Classificação*, *Contexto* e com a *Descritor* para formar um par (atributo, valor). Por exemplo, pode ser (**Tecnologia, JAVA**), onde o atributo **Tecnologia** é definido pela classe *Grupo Descritor* e o valor **Java** corresponde ao da classe *Descritor*. O valores assumidos pelo par (atributo, valor), são livres no modelo RAS. Este método funciona semelhantemente ao método de *classificação por facetas*.

O método de classificação por facetas (DIAZ, 1991) é baseado na análise de um assunto em função de seus termos básicos para caracterizá-lo através de facetas ou classes elementares básicas. Facetas são consideradas como perspectivas, visões ou dimensões de um domínio de aplicação em particular.

No caso da classificação em facetas, o método de busca consiste na seleção de termos das facetas para se chegar aos ativos de software que satisfaçam esta seleção. Desta forma, as facetas fornecem as listas pré-definidas de termos que são apresentadas aos usuários para continuar a formulação da pesquisa. Este par atributo e valor representa uma classificação por facetas. A partir de estudo da literatura, propomos, neste trabalho, as seguintes facetas:

- **Palavra-Chave**: conjunto de palavras que descrevam a finalidade do ativo de software, que tipo de problema ele resolve. Por exemplo: ODBC, Report, Banco de Dados e etc.

- **Categoria**: especifica a área que é atendida pelos serviços oferecidos pelo ativo software. Por exemplo: *Business Component*, *Código de Barras*, *Infra-Estrutura*, *Serviço de Negócios*, *Sistema Operacional*.

- **Aplicação**: especifica qual a finalidade dos serviços oferecidos dentro do domínio do componente, como por exemplo, dentro do domínio de hotelaria, pode-se considerar, reservas, check-in, check-out, ou alocação de quartos.

- **Distribuição**: Tipo de distribuição do ativo. Por exemplo: Comercial, *Open Source*, Demonstração, *Shareware*, etc.

- **Tecnologia Principal**: especifica o ambiente tecnológico em que o componente de software pode ser utilizado. Por exemplo: .NET, CORBA, J2EE, JAVA e etc.

- **Plataforma**: a plataforma na qual o ativo de software pode ser utilizado. Por exemplo: Bea WebLogic 6.1,Bea WebLogic 7.1, IBM WebShere 4.x,IBM WebShere 5.x, IBM WebShere 6.x, JBoss 3.0.4, Tomcat 4.0.x, Tomcat 4.1.x, .NET, etc.

- **Tipo de Ativo**: especifica o tipo do componente de software, ou seja, subrotina, classe, pacote, framework ou sistema;

- **Idioma**: Inglês, Português, Alemão, etc.

- **Linguagem de Programação**: Linguagem em que foi desenvolvido o componente. Por exemplo: Java, Delphi, Cobol, C++ e etc.

Para permitir uma padronização dos termos básicos que atendem ao método de classificação por facetas e também a padronização de alguns termos que podem ser atribuídos como valores válidos dos atributos, novas classes são adicionadas ao meta-modelo RAS. A Tabela 3.5 apresenta uma relação das novas classes acrescentadas ao meta-modelo do repositório.

Estas classes adicionais permitem uma melhor classificação do componente devido à padronização de termos e conseqüente facilidade do mecanismo de busca dos artefatos do repositório. A Figura 3.5 mostra uma interface que se baseia na seleção de uma ou mais facetas como critério de recuperação do ativo de software.

Tabela 3.5: Tabelas - Padronização de Termos do Repositório.

Nome Classe	Descrição da Classe
extdiraces	Classe onde estão todos os valores de tipos de direito de acesso permitidos para o atributo access-rights da classe asset.
extstasset	Classe onde estão todos os valores permitidos para o atributo status da classe asset.
extdepenartef	Classe onde estão todos os valores de tipos de dependência permitidos para o atributo dependency-type da classe artifact-dependency.
extdescritor	Classe onde estão todos os valores permitidos de descritores pertencentes a um determinado grupo descritor. Cada descritor é associado a um grupo descritor. Por exemplo: grupo descritor Idioma (Inglês, Português, Alemão, etc).
extgrdescritor	Classe onde estão todos os valores permitidos para grupo descritor. Exemplo: Idioma, Tecnologia, Categoria, etc.
extcontexto	Classe onde estão todos os valores permitidos para contexto.
extsoluctpdoc	Classe onde estão cadastrados todos os tipos de documentos.
extpoartefato	Classe onde estão cadastrados todos os tipos de artefatos.

3.4.3 Extensão para Controle de Acesso

A Figura 3.6, mostra a extensão do meta-modelo RAS para a Gestão de Controle de Acesso às versões de ativos de software no repositório.

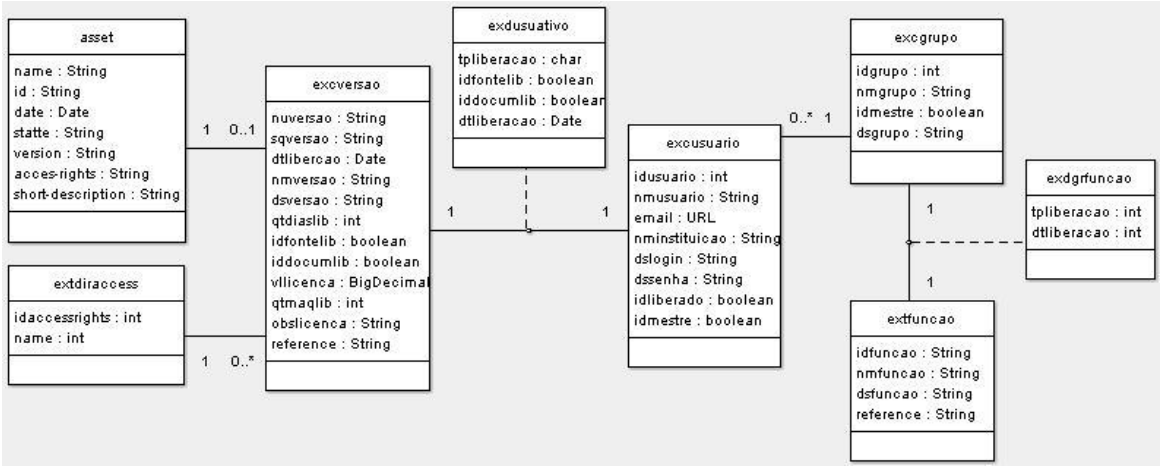


Figura 3.6: Diagrama de Classe para Controle de Acesso

O repositório deve adotar um **Controle de Acesso** com **Política de Segurança** para que determinadas funcionalidades só estejam acessíveis a pessoas autorizadas. Pode-se, por exemplo, definir para uma pessoa específica, uma política de segurança onde a pesquisa a um ativo de software por palavra-chave ou outro critério de seleção esteja disponível, mas a recuperação e/ou download de ativos não esteja disponível. Mas pode ser adotada outra política de segurança para pessoas que pertencem a equipe de desenvolvimento, onde também a recuperação e ou dowload de ativos esteja disponível. Uma outra política de segurança pode ser adotada para o administrador do repositório, que possui permissão de acesso a todas as funcionalidades do aplicativo e também permissão total a todos os ativos de software.

A Tabela 3.6 mostra a relação das novas classes para Gestão de Controle de Acesso.

Tabela 3.6: Relação das Classes para Gestão de Controle de Acesso.

Nome Classe	Descrição da Classe
excgrusuario	Classe grupos de usuários. Um grupo é um conjunto de pessoas que possuem o mesmo nível de permissão de acesso.
excusuario	Classe dos usuários que tem permissão de acesso ao repositório.
extfuncao	Classe das funções do aplicativo ou link's.
exdgrfuncao	Classe que associa todas as funções e/ou link's a um grupo de usuário.
exdusuativo	Classe que associa um usuário a todas as versões de ativos de software que tem permissão de acesso e o nível de acesso que o usuário possui sobre o ativo de software.

Um grupo de usuários é um grupo de pessoas que possui a mesma política de acesso e permissão. A criação do grupo de usuários facilita a implementação de segurança, pois ao liberar a permissão para o grupo, todos os usuários do mesmo grupo recebem a mesma permissão. A Tabela 3.7 corresponde relação de atributos da classe de grupo de usuários (excgrusuario).

Todas as pessoas que quiserem acessar o repositório, terão previamente que realizar o cadastro como usuário, quando automaticamente receberão o nível de permissão padrão. Este nível permite acesso a determinados ativos de software com determinado nível de permissão.

Tabela 3.7: Relação de atributos da classe Grupo de Usuários (excgrusuario).

Campo	Descrição	Tipo	Obrig
idgrupo	Identificador do grupo de usuários	inteiro	S
nmgrupo	Nome do grupo de usuários	string	S
idmestre	Status Mestre (S,N)	string	S
dtreginc	Data inclusão do registro	timestamp	S
idusuinc	Identificação do usuário incluiu registro	inteiro	S
idativo	Status do registro ativo(S,N)	string	S
dtaltreg	Data de alteração do registro	timestamp	S
idusualt	Identificação do usuário que alterou o registro	inteiro	S

Usuário mestre tem acesso completo ao repositório de ativos e às funcionalidades dos aplicativos que manipulam o repositório. A Tabela 3.8 contém todos os atributos da classe usuário (excsuario).

O cadastro de usuários contém informações suficientes para identificação e para que o módulo de *Gestão de Divulgação* possa fazer contato. O usuário pode ser classificado como mestre, neste caso tem acesso completo aos ativos do repositório e a todas as funcionalidades dos aplicativos que o manipulam.

A *Gestão de Liberação de Acesso* é responsável pelo cadastro das funcionalidades do aplicativo que acessa o repositório e a associação do nível permissão ao grupo de usuários, permitindo assim uma restrição de acesso às funcionalidades do aplicativo (inclusão, alteração, exclusão, consulta ou pesquisa). Uma função pode ser uma opção de menu ou um link em uma página web. Toda função que tiver algum nível de restrição de acesso, obrigatoriamente deve ser cadastrada. Para uma função sem restrição, não é obrigatório o cadastro, mas é aconselhável para o gerenciamento.

Tabela 3.8: Relação dos atributos da classe Usuário (excsuario).

Campo	Descrição	Tipo	Obrig
idusuario	Identificador do usuário	inteiro	S
nmusuario	Nome do usuário	string	S
idgrupo	Identifica o grupo do usuário	inteiro	S
email	Email do usuário	string	S
nminstituicao	Nome da Instituição	string	S
dslogin	Login do Usuário	string	S
dssenha	Senha do Usuário	string	S
idliberado	Status Liberação (S,N)	string	S
idmestre	Status Mestre (S,N)	string	S
dtreginc	Data inclusão do registro	timestamp	S
idusuinc	Identificação do usuário que incluiu registro	inteiro	S
idativo	Status do registro ativo(S,N)	string	S
dtaltreg	Data alteração do registro	timestamp	S
idusualt	Identificação do usuário que alterou registro	inteiro	S

A Tabela 3.9 apresenta a relação dos atributos da classe Funções do Aplicativo (extfuncao).

Tabela 3.9: Relação dos atributos da classe Funções do Aplicativo (extfuncao).

Campo	Descrição	Tipo	Obrig
idfuncao	Identificador da função ou link	inteiro	S
nmfuncao	Nome do função ou link	string	S
dsfuncao	Descrição da Função ou link	string	S
dtreginc	Data inclusão do registro	timestamp	S
stacesso	Status de Acesso Livre(S,N)	string	S
idusuinc	Identificação do usuário que incluiu registro	inteiro	S
idativo	Status do registro ativo(S,N)	string	S
dtaltreg	Data alteração do registro	timestamp	S
idusualt	Identificação do usuário que alterou registro	inteiro	S

A *Gestão de Liberação de Funções* é responsável pelo vínculo do grupo de usuários a todas as funcionalidades do aplicativo às quais o grupo possui permissão.

A Tabela 3.10 mostra a relação dos atributos da classe associativa que faz a associação do grupo de usuários as funcionalidades do aplicativo (exdgrfuncao). Mesmo que um usuário possa ter **acesso às funcionalidades** do aplicativo, o mesmo poderá ter **restrições de acesso** a uma determinada versão de um ativo de software. A *Gestão de Liberação de Ativo* vincula ao usuário quais os ativos de software que podem ser acessados, bem como o nível de permissão. O Admininstrador do Repositório é o responsável pela Gestão de Liberação de Ativos.

A Tabela 3.11 mostra os atributos da classe que associa ao Usuários aos Ativos (exdusuativo).

Tabela 3.10: Relação dos atributos da classe (exdgrfuncao)

Campo	Descrição	Tipo	Obrig
idfuncao	Identificador da função ou link	inteiro	S
idgrupo	Identificador do grupo de usuários	inteiro	S
tpliberacao	Nível de Permissão Função ou Link	string	S
dtreginc	Data inclusão do registro	timestamp	S
idusuinc	Identificação do usuário que incluiu registro	inteiro	S
idativo	Status do registro ativo(S,N)	string	S
dtaltreg	Data alteração do registro	timestamp	S
idusualt	Identificação do usuário que alterou registro	inteiro	S

Tabela 3.11: Relação de atributos da Classe associa Ativos aos Usuários (exdusuativo).

Campo	Descrição	Tipo	Obrig
idusuario	Identificador do usuário	inteiro	S
idasset	Identifica o Ativo de Software	inteiro	S
nuversao	Identifica a versão Ativo de Software	string	S
tpliberacao	Nível de Permissão sobre o ativo	string	S
idfontelib	Status do Fonte Liberado Usuário(S,N)	string	S
Iddocumlib	Status Documentação Liberado Usuário(S,N)	string	S
dtreginc	Data inclusão do registro	timestamp	S
idusuinc	Identificação do usuário que incluiu registro	inteiro	S
idativo	Status do registro ativo(S,N)	string	S
dtaltreg	Data alteração do registro	timestamp	S
idusualt	Identificação do usuário que alterou registro	inteiro	S

3.4.4 Extensão para Controle de Modificações do Ativo e Histórico

Um ativo é projetado para o reuso. São de suma importância algumas funcionalidades para se fazer o gerenciamento de modificações dos ativos num repositório. Um repositório deve prover funções que notifiquem os seus usuários das modificações recentemente ocorridas, tais como inserção ou exclusão de ativos, alterações em documentos, disponibilização de novas funcionalidades, novas políticas de segurança, entre outras.

A Figura 3.7, mostra o diagrama de classe para extensão do meta-modelo RAS para que dê suporte ao Controle de Modificações do Ativo e Histórico de acessos realizados por usuários autorizados ao repositório de ativos de software.

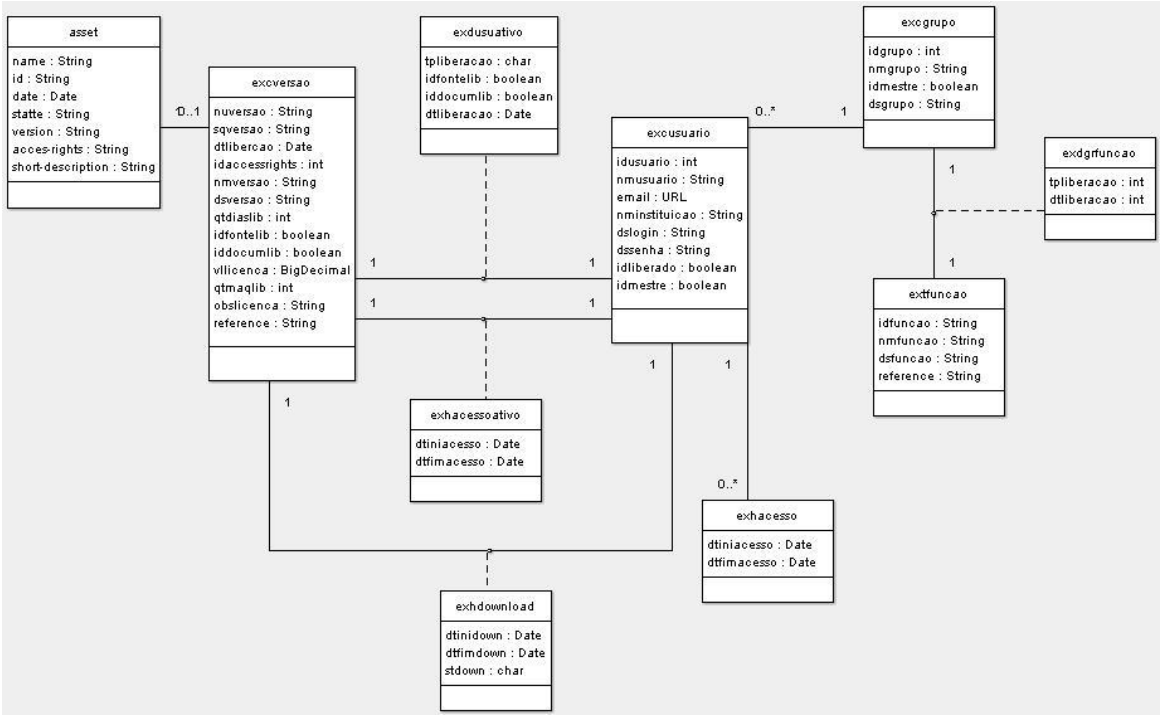


Figura 3.7: Diagrama de Classe Gestão de Controle de Modificações Ativo e Histórico.

A Tabela 3.12 mostra a descrição das novas classes para a Gestão de Controle de Modificações do Ativo e Histórico.

Tabela 3.12: Classes para Controle de Modificações do Ativo e Históricos.

Nome Classe	Descrição da Classe
exhacesso	Classe histórico dos acessos do usuário ao repositório.
exhdownload	Classe Histórico de download de ativos de software realizados por determinado usuário.
exhativo	Classe Histórico de alterações das versões do ativo de software.
exhartefato	Classe Histórico das alterações de dos artefatos do ativo de software.
excsolicita	Classe Solicitações de alteração e/ou reclamação sobre um ativo de software.

Quando um ativo é disponibilizado para uso, o nível de utilização aumenta bastante e consequentemente o seu nível de teste, pois são diversos usuários que estão utilizando o ativo, o que pode provocar manutenções corretivas e/ou adaptativas para adequar a novos requisitos e funcionalidades. Quando for registrada alteração de alguma versão de ativo de software, funcionalidade de *Gestão de Divulgação* se encarrega de fazer a comunicação aos usuários que efetuaram download do ativo de sofware sobre a atualização, para que o mesmo fique atento às atualizações dos ativos.

A Tabela 3.13 mostra os atributos da classe cadastro de histórico de versões de ativo(exhativo). Essas informações devem fazer parte de uma base histórica que facilitará a análise e a reutilização dos mesmos.

Tabela 3.13: Relação dos atributos da classe Histórico das alterações versões ativos (exhativo).

Campo	Descrição	Tipo	Obrig
idasset	Identificador do ativo de software	inteiro	S
nuversao	Identificador da versão do ativo software	string	S
sqversao	Identifica a seqüência da versão do Ativo	string	S
dtalterac	Data e Hora Alteração	timestamp	S
dtlibversao	Data da Liberação Versão	timestamp	S
nmversao	Nome da versão	string	S
dsversao	Descrição da versão	string	S
cdtipolic	Tipo da licença	inteiro	S
qtdiaslib	Quantidade de dias liberados na licença	inteiro	S
idfontlib	Status de fonte liberado	string	S
iddocumlib	Status de documentação liberado	string	S
vllicenca	Valor da liberação da licença	moeda	S
qtmaqlib	Quantidade máquinas liberadas na licença	inteiro	S
obslicenca	Observação da licença	string	S
dtreginc	Data inclusão do registro	timestamp	S
idusuinc	Identificação do usuário que incluiu registro	inteiro	S
idativo	Status do registro ativo(S,N)	string	S
dtaltreg	Data de alteração do registro	timestamp	S
idusualt	Identificação do usuário que alterou registro	inteiro	S

Todo acesso ao repositório feito por um usuário é registrado pelas funcionalidades do *Gestão de Estatística*. A Tabela 3.14 mostra a relação dos atributos da classe Histórico de Acesso ao repositório (exhacesso).

Tabela 3.14: Relação dos atributos classe Histórico de Acesso Repositório(exhacesso).

Campo	Descrição	Tipo	Obrig
idusuario	Identificador do usuário	inteiro	S
dtiniacesso	Data e hora do início do acesso	timestamp	S
dtfimacesso	Data e hora do fim acesso	timestamp	S
idusuinc	Identificação do usuário que incluiu registro	inteiro	S
idativo	Status do registro ativo(S,N)	string	S
dtaltreg	Data de alteração do registro	timestamp	S
idusualt	Identificação do usuário que alterou registro	inteiro	S

Todo acesso a uma versão do ativo de software do repositório, feito por um usuário é registrado também pelas funcionalidades do *Gestão de Estatística.* O Registro dos acessos é de suma importância, pois isto pode auxiliar um sistema multiagente executar um processo de busca e recuperação de um componente de software, por exemplo, qual o componente mais acessado por usuários, quais foram realizados mais download.

A Tabela 3.15 mostra relação dos atributos da classe Histórico acesso a uma versão de ativo de software (exhacessoativo).

Tabela 3.15: Relação dos atributos classe Histórico acesso a Ativo Software (exhacessoativo).

Campo	Descrição	Tipo	Obrig
idusuario	Identificador do usuário	inteiro	S
idasset	Identificador do ativo de software	inteiro	S
nuversao	Identificador da versão do ativo software	string	S
dtiniacesso	Data e hora do início do acesso	timestamp	S
dtfimacesso	Data e hora do fim do acesso	timestamp	S
idusuinc	Identificação do usuário que incluiu registro	inteiro	S
idativo	Status do registro ativo(S,N)	string	S
dtaltreg	Data de alteração do registro	timestamp	S
idusualt	Identificação do usuário que alterou registro	inteiro	S

Toda recuperação de ativo que é seguida por uma operação de download de versões de ativos, deve ser registrada pelas funcionalidades do *Gestão de Estatística*. A Tabela 3.16 mostra a relação dos atributos da classe Histório de Download de ativos de software (exhdownload).

A *Gestão de Solicitação* é responsável pelo registro de todas as solicitações, que podem ser enquadrar em duas categorias:

- **Adaptativas**: Os desenvolvedores de sistemas que utilizam o ativo de software podem solicitar adaptação de novas funcionalidades. O Administrador do Repositório deverá analisar juntamente com o Desenvolvedor do Componente a viabilidade da solicitação e responder ao solicitante.

- **Corretivas**: Apesar dos ativos passarem por um exaustivo processo de testes, quando passam a ser utilizados em diversos contextos e/ou ambientes, podem apresentar erros durante o seu uso. Estas solicitações de correção devem ser registradas e posteriormente, quando são corrigidas, devem ser disponibilizadas

Tabela 3.16: Relação dos atributos da classe História de Download (exhdownload).

Campo	Descrição	Tipo	Obrig
idusuario	Identificador da função ou link	inteiro	S
idasset	Identificador do ativo de software	inteiro	S
nuversao	Identificador da versão do ativo software	string	S
dtinidown	Data e Hora do Início Download	timestamp	S
dtfimdownl	Data e Hora do Fim Download	timestamp	S

no repositório. A *Gestão de Divulgação* deverá comunicar a todos os usuários que utilizam o ativo sobre a atualização de versão.

A Tabela 3.17 apresenta as informações utilizadas para o registro do cadastro de solicitações. Mostra a relação dos atributos da classe Solicitações de Alterações (exh-solicita).

Tabela 3.17: Relação dos atributos classe Solicitações de Alterações (exhsolicita).

Campo	Descrição	Tipo	Obrig
idsolicita	Identificador da solicitação	inteiro	S
dtsolicita	Data Solicitação	timestamp	S
idasset	Identificador do ativo de software	inteiro	S
nuversao	Identificador da versão do ativo software	string	S
stsolicita	Status solicitação (A, S, E)	string	S
dssolicita	Descrição Solicitação	string	S
dsresposta	Descrição da Resposta	string	S
dtresposta	Data da Resposta	timestamp	S
idusuresp	Identifica Usuário Resposta	inteiro	S
dtreginc	Data inclusão do registro	timestamp	S
idusuinc	Identificação do usuário incluiu registro	inteiro	S
idativo	Status do registro ativo(S,N)	string	S
dtaltreg	Data alteração do registro	timestamp	S
idusualt	Identificação do usuário alterou registro	inteiro	S

3.4.5 Extensão para Gestão de Controle de Versões

Conforme citado na Seção 3.4.4, um ativo de software pode sofrer manutenções adaptativas ou corretivas. Para se ter uma boa funcionalidade da *Gestão de Componentes*, é de fundamental importância o registro das atualizações das diversas versões dos ativos de software, pois podem existir usuários trabalhando com mais de uma versão.

A Figura 3.8 mostra o diagrama de classe que representa a extensão do modelo RAS para suporte a Gestão de Controle de Versões.

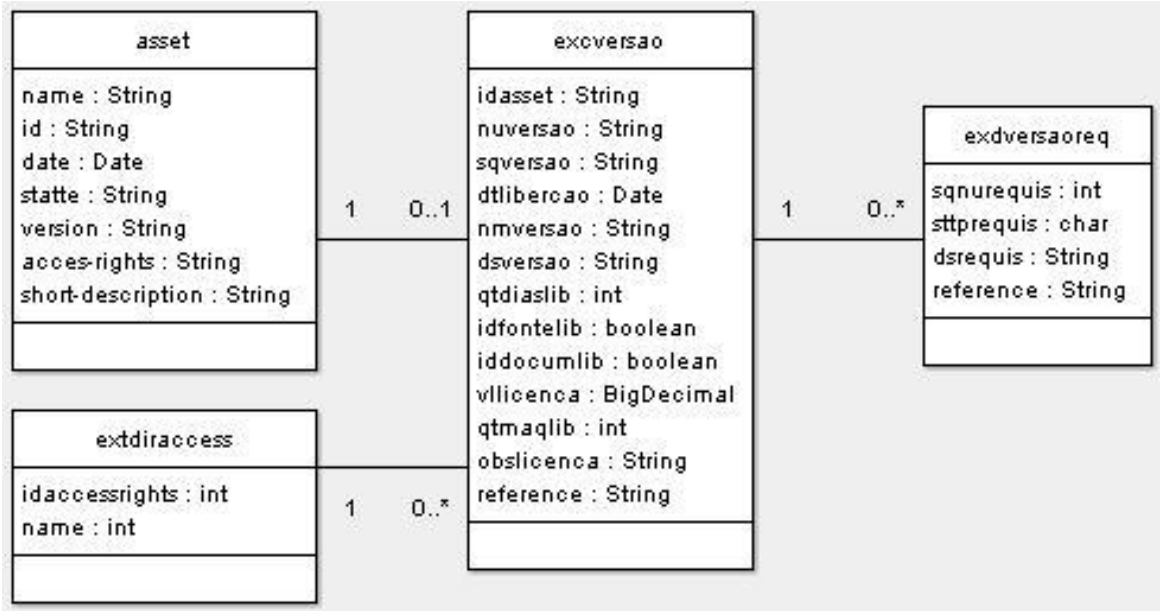


Figura 3.8: Diagrama de Classes para Controle de Versões.

A Tabela 3.18 exibe a relação dos atributos da classe Versões Ativos de software (excversao).

Um fator importante para a recuperação de um ativo de software é o conhecimento de seus requisitos, tanto os requisitos funcionais como os não funcionais. A Seção **solução do modelo RAS** pode apresentar os requisitos por uma referência a um documento externo através do atributo *reference* da classe *artifact*(classe da seção solução descrita no modelo RAS). O repositório de ativo de software propõe uma classe para registro dos requisitos do ativo, para facilitar o processo de recuperação. A Tabela 3.19 apresenta relação dos atributos da classe Requisitos da Versão

Tabela 3.18: Relação dos atributos classe Versões Ativos (excversao).

Campo	Descrição	Tipo	Obrig
idasset	Identifica o Ativo de Software	inteiro	S
nuversao	Identifica a versão Ativo de Software	string	S
sqversao	Identifica a seqüência da versão do Ativo	string	S
dtlibversao	Data da Liberação Versão	timestamp	S
nmversao	Nome da Versão	string	S
dsversao	Descrição da Versão	string	S
cdtipolic	Tipo da Licença	inteiro	S
qtdiaslib	Quantidade de dias Liberado Licença	inteiro	S
idfontlib	Status de Fonte Liberado	string	S
iddoculib	Status de Documentação Liberado(S,N)	string	S
vllicenca	Valor da Liberação Licença(S,N)	moeda	S
qtmaqlib	Quantidade Máquinas Liberada Licença	inteiro	S
obslicenc	Observação da Licença	string	S
dtreginc	Data inclusão do registro	timestamp	S
idusuinc	Identificação do usuário que incluiu registro	inteiro	S
idativo	Status do registro ativo(S,N)	string	S
dtaltreg	Data alteração do registro	timestamp	S
idusualt	Identificação do usuário que alterou registro	inteiro	S

(exdversaoreq), cadastro de requisitos funcionais e não-funcionais de uma versão de ativo de software.

O atributo *sttiporeq* (FU, NF) identifica categoria do requisito. A categoria pode assumir dois valores: Requisitos Funcionais (FU) e Requisito Não-Funcionais (NF). O atributo *reference* é o endereço de um documento e/ou uma referência de uma URL que contém um detalhamento maior sobre o requisito. Um ativo de software pode ter um ou mais requisitos.

Tabela 3.19: Relação dos atributos classe Requisitos da Versão do Ativo (exdversaoreq).

Campo	Descrição	Tipo	Obrig
idasset	Identifica o Ativo de Software	inteiro	S
nuversao	Identifica a versão Ativo de Software	string	S
sqversao	Identifica a seqüência da versão do Ativo	string	S
sqnurequi	Número do Requisito	inteiro	S
sttiporeq	Identifica Tipo Requisito (FU,NF)	string	S
dsrequisi	Descrição Requisito	string	S
reference	Nome da Versão	string	S
dtreginc	Data inclusão do registro	timestamp	S
idusuinc	Identificação do usuário que incluiu registro	inteiro	S
idativo	Status do registro ativo(S,N)	string	S
dtaltreg	Data da alteração do registro	timestamp	S
idusualt	Identificação do usuário que alterou registro	inteiro	S

3.5 Implementação do Repositório

O Repositório de Componentes de Software foi implementado utilizando o Sistema Gerenciador de Banco de Dados (SBGD) *MySQL Server verão 5.0 (Sun).* O MySQL (http://dev.mysql.com/downloads/) é um SGBD com código fonte aberto que tem sido utilizado como solução por desenvolvedores de sistema, provedores de serviços, aplicações Enterprise e também para suportar aplicações livres.

O MySQL foi escolhido para implementação do repositório por apresentar uma solução confiável e baixo custo para aplicações críticas, com grande volume de dados e que necessitam de uma alta disponilidade da aplicação. Ele apresenta diversas características de um bom SGBD, tais como: facilidade de uso, desempenho, confiabilidade, suporte a usuários, boa documentação, independência de arquitetura e sistema operacional, suporte para processadores de 32 e 64-bits, filosofia de compatibilidade total com o padrão ANSI/SQL (desde que não haja comprometimento de desempenho e confiabilidade). Além disso, ele suporta transações ACID (atomicidade, concorrência, isolamento e durabilidade), *Triggers* (Gatilhos), *Stored Procedures* (Procedimentos), *Views* (Visões), implementação de integridade referencial, ferramentas de administração (*Backup*, *Restore*, Replicação, Segurança), entre outras facilidades.

Um outro recurso do MySQL é o *full text search*, que permite realizar pesquisas textuais em textos longos utilizando palavras-chave. Este recurso é importante, pois elimina o uso do comando SQL(Structured Query by Example) **like *padrão***, que normalmente possui um alto custo de recuperação. Possui também recurso OPENGIS, que dá suporte para dados espacias que permitem trabalhar com geoprocessameto. Também possui um mecanismo de *Cluster*, que permite a utilização de banco de dados distribuídos. Os sistemas distribuídos permitem a aplicação acessar dados como se estivessem em um único servidor, apesar dos dados estarem em servidores separados.

Em síntese, o MySQL foi escolhido porque é robusto, apresenta bom desempenho e facilidade de instalação, configuração e manipulação. Possui suporte para várias plataformas, sistemas operacionais, além de ser livre e possuir uma empresa por trás do produto e toda uma comunidade que disponibiliza ferramentas gratuítas

para administração e manipulação das informações.

Foi utilizada uma ferramenta/aplicativo com interface gráfica bem interativa para a administração e criação da base de dados do repositório. A Figura 3.9 mostra interface do MySQL - Front versão 2.5(http://www.freedownloadscenter.com/Best/mysql-front-2$_3$.*html*).

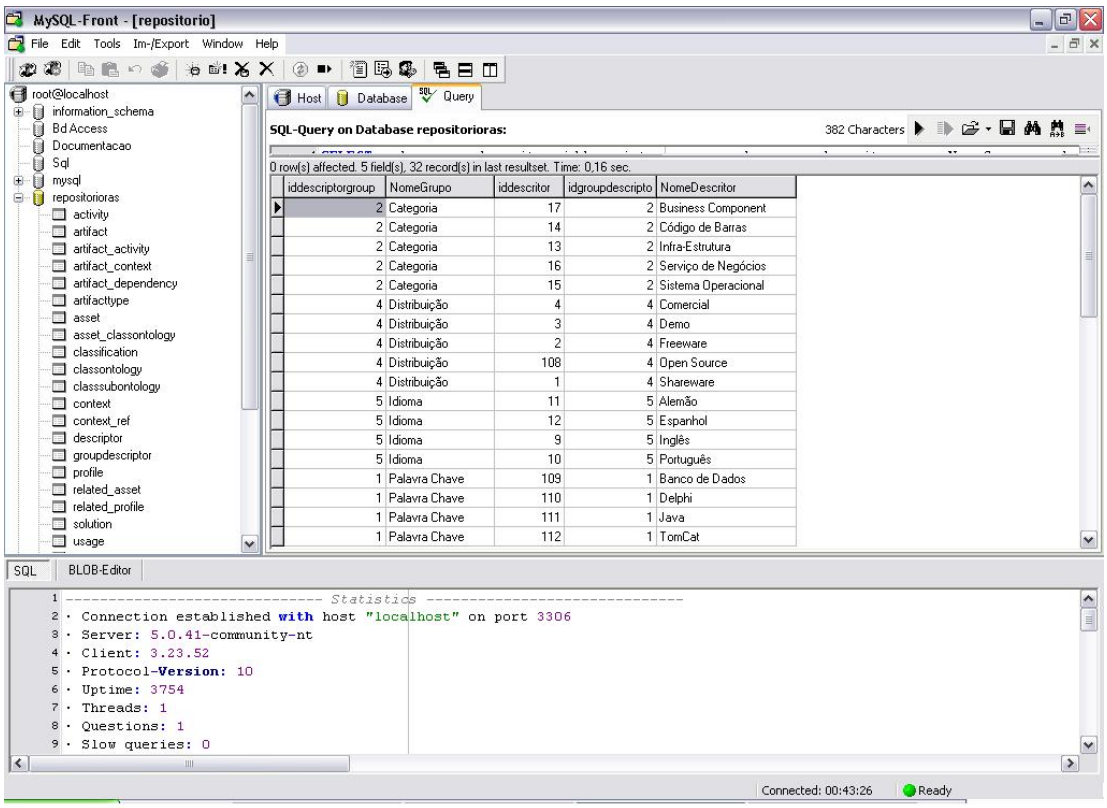


Figura 3.9: Ferramenta Front End para administrar o SGBD.

3.6 Desenvolvimento do Aplicativo

A arqutitera ideal para o desenvolvimento de uma RCCS é através de uma arquitetura independente de plataformas e da utilização dos padrões Web Services.

Um requisito típico é que o repositório possa ser acessado de qualquer ponto numa rede, uma vez que seus usuários estão distribuídos geograficamente. O que normalmente se tem observado na implementação dos repositórios é a utilização de modelos cliente-servidor em 3 (três) ou mais camadas, distribuindo as atividades em cada uma das camadas, tais como: uma camada é a responsável pelo armazenamento dos ativos, a outra pelo processamento das principais funcionalidades do sistema (pesquisa e recuperação) e a terceira corresponde à aplicação cliente, por onde o repositório poderá ser acessado.

Neste trabalho, foi desenvolvido um aplicativo cliente servidor na plataforma Borland Delphi 7. O aplicativo foi desenvolvido com a finalidade principal de facilitar a classificação, o cadastro dos ativos de software no repositório, permitindo assim uma experimentação mais real do modelo estendido. Foram desenvolvidas também interfaces de Busca e Recuperação de ativos de software.

A escolha da Plataforma Delphi foi devido a simplicidade da linguagem de programação Pascal e sua facilidade de aprendizado, obtendo assim maior rendimento nos trabalhos. Além disso, possibilita a reutilização de formulários e códigos já produzidos, como por exemplo os formulários padronizados de cadastro, que são herdados a partir de um modelo predefinido. As conexões e consultas ao banco de dados são feitas de forma simples e rápida, diminuindo a probabilidade de erros na busca de informações. Apesar da rapidez na construção de formulários, a ferramenta dispõe de meios pouco eficientes para geração de relatórios de impressão. A montagem do relatório é dificultada pela forma manual de compor os elementos da página a ser impressa.

3.6.1 Formulário de Cadastro de Ativos no Repositório

O formulário de cadastro permite incluir, alterar e excluir um ativo de software. Um ativo pode possuir uma ou mais classificações dentro de um determinado contexto. Para cada classificação podem ser associados um ou mais pares descritores (atributo, valor), por exemplo, o par (Idioma, Inglês) e (Tecnologia Principal, Java). Um ativo pode estar relacionado a um ou mais ativos. Além da classificação RAS, o ativo pode ser cadastrado conforme a classificação ontológica, pode ser classificado em uma ou mais categorias. A Figura 3.10 mostra o formulário desenvolvido para cadastro do ativo de software.

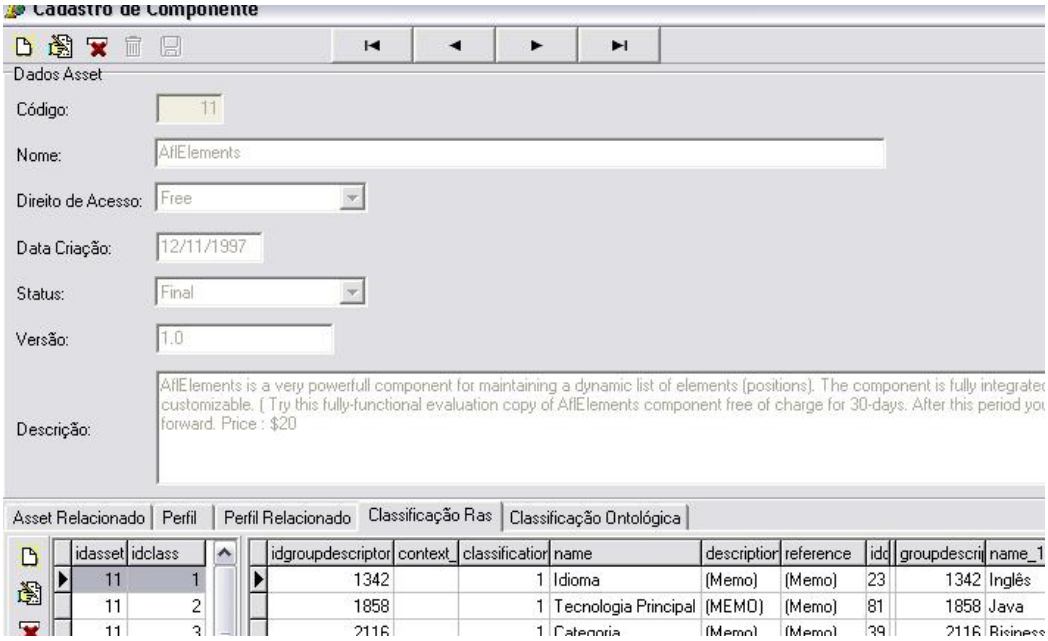


Figura 3.10: Formulário de Cadastro de Ativos no Repositório.

3.6.2 Formulário de Busca e Recuperação de Ativo

O formulário permite o uso de diversos filtros para a seleção de ativos de softare no repositório, tais como: (1) Classificação Ontológica, (2) Classificação RAS, por nome do ativo e também pela descrição do ativo. O resultado da seleção é retornado e exibido no grid (3). Em (4), o usuário poderá verficar mais detalhes sobre o ativo retornado, poderá procurar por ativos semelhantes conforme classificação ontológica, procurar

por ativos que pertençam a classes de nível superior as classe do ativo selecionado ou procurar por ativos que pertençam a classes de nível inferior as classe do ativo selecionado. Estes detalhes são mostrados na Figura 3.11.

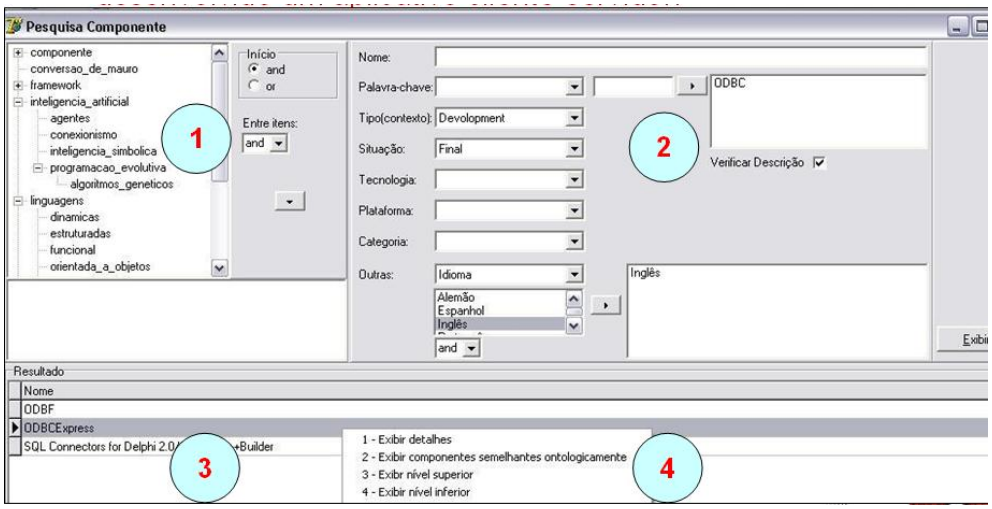


Figura 3.11: Formulário de Busca e Recuperação de Ativo no Repositório

A Figura 3.12 mostra os detalhes do ativo de software selecionado pelo usuário. Estas funcionalidas implementadas tiveram como foco principal a validação das extensões propostas por este trabalho.

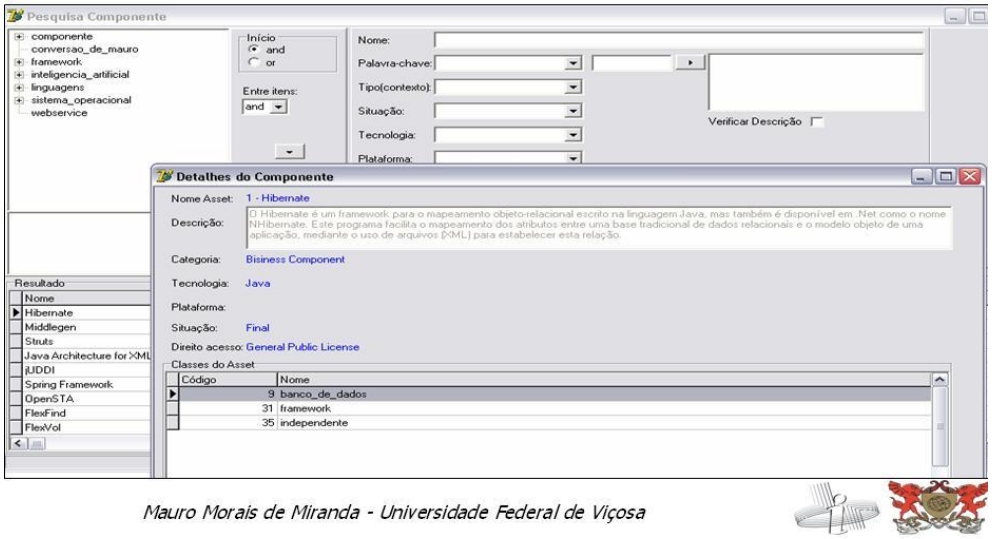


Figura 3.12: Formulário de Busca e Recuperação com detalhes do Ativo pesquisado.

3.7 Agente de Busca

O agente de busca proposto será utilizado para auxiliar na localização de componentes. Cada repositório terá um agente de busca associado e sua implementação na forma de agente deve-se à necessidade de interação com os outros agentes dos outros repositórios para a localização remota de componentes. A Figura 3.13 ilustra o caráter distribuído da sociedade de agentes associados aos repositórios.

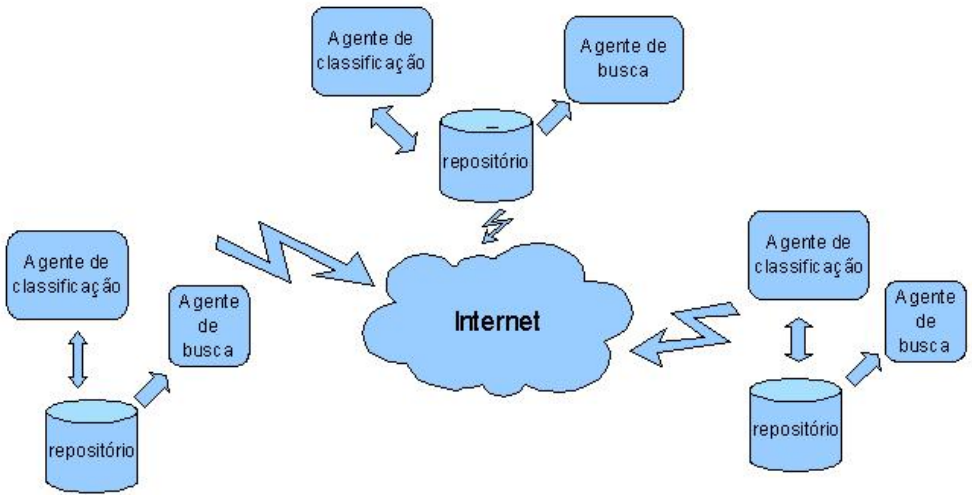


Figura 3.13: Associação dos Agentes com os Repositórios.

O agente de busca poderá ser implementado na linguagem de programação *Java* dentro do *framework* **JADE** (*Java Agent DEvelopment Framework*) (BELLIFEMINE et al., 2001). *Jade* é um framework para desenvolvimento de sistemas multi-agentes que segue as especificações da **FIPA1** (*Foundation for Intelligent Physical Agents*) que tem por objetivo simplificar o desenvolvimento de sistemas multi-agentes, disponibilizando serviços tais como uma plataforma distribuída de agentes, serviços de nomes, transporte de mensagens, serviços de codificação e decodificação de mensagens e uma biblioteca de protocolos de interação (padrão *FIPA*) pronta para ser usada.

Para fazer a comunicação entre os agentes, o *framework Jade* utiliza o padrão de mensagens *FIPA-ACL*. O agente receberá as consultas por meio de uma mensagem *FIPA-ACL* e traduzirá o pedido para um comando **SQL**(Structured Query Language) e o submeterá ao **SGBD** (Sistema Gerenciador de Banco de Dados) do repositório.

Caso a consulta não seja bem sucedida o agente usará a ontologia por meio da API *protégé-OWL* e com o auxílio do raciocinador *Pellet*, obterá as categorias imediatamente superiores da categoria do componente solicitado, refazendo assim, a consulta, envolvendo categorias mais amplas.

O Pellet servirá para obter as categorias superiores não declaradas e que podem ser inferidas a partir das definições das classes. As informações sobre os componentes selecionados serão enviadas na forma de mensagem *FIPA-ACL* para quem realizou a consulta. A Figura 3.14 ilustra a interação entre os módulos que participam do processo.

Figura 3.14: Interação do Agente de Busca.

Capítulo 4

Conclusões e Trabalhos Futuros

4.1 Conclusões

Em razão do crescimento do desenvolvimento de software baseado em componentes, várias ferramentas para o armazenamento e busca de componentes têm sido propostas. O traballho apresentado em GARCIA et al. (2005), por exemplo, apresenta uma arquitetura para um sistema de busca de componentes armazenados em repositórios CVS. Além de não se basear em um repositório organizado segundo a especificação RAS, a arquitetura difere do trabalho apresentado nesta dissertação por não utilizar agentes e ontologia, além de estar restrito ao uso do ambiente Eclipse.

Neste trabalho, foi realizado um estudo do meta-modelo de informações RAS, que é o padrão proposto pela OMG (CARLSON et al., 2005), visando uma melhor organização e caracterização dos ativos de software para fins de reuso. Foi proposta uma extensão do meta-modelo de informações RAS com o uso de uma ontologia sobre componentes. Ela permite assim a definição de um componente de software como o conjunto de meta-informações que ele deve possuir de forma estruturada, o que facilita a catalogação e a recuperação de componentes de software. Foi projetado um conjunto de tabelas que espelham a especificação RAS e as novas funcionalidades do modelo proposto, tais como: gestão de versões de componentes de software, gestão de usuários, gestão de recuperação, gestão de liberação de componentes e gestão de divulgação.

Foi implementado um repositório de componentes utilizando o Sistema Gerenciador de Banco de Dados MySQL versão 5.0, tendo como base tabelas que espelham o modelo RAS. Foram acrescidas ao meta-modelo tabelas para cadastro dos grupos descritores e descritores de forma a criar uma padronização dos termos que são utilizados para a classificação de componentes (RAS) com um par atributo e valor, que na realidade representa aplicação do método de classificação por facetas (DIAZ, 1991). A padronização dos termos leva a um grande ganho na qualidade da classificação e catalogação dos componentes. Isto leva a uma melhor qualidade na pesquisa e recuperação de componentes, já que as facetas são consideradas como perspectivas, visões ou dimensões de um domínio de aplicação em particular.

O modelo RAS considera que requisitos funcionais e não funcionais são artefatos do ciclo de desenvolvimento. Estes artefatos são referenciados a um documento externo, o que muitas vezes dificulta a pesquisa. Também muitos dos componentes não apresentam uma boa documentação de sua especificação de desenvolvimento. Por isto, foram incluídas tabelas com a finalidade de permitir o registro dessas informações, que muitas vezes são essenciais para definição se o componente selecionado é adequado ou não à necessidade que o desenvolvedor necessita.

Nesta dissertação são propostas novas tabelas para cadastro das informações da interface que é de fundamental importância para saber como o componente pode associar-se a outro componente ou a um subsistema. Trabalhos correlatos já demonstram esta preocupação. O conhecimento da interface é também um fator decisório na seleção e escolha de um componente de software e sua integração a outros componentes.

Neste trabalho, foram acrescidas tabelas para registro de versões do ativo de software. Um ativo de software pode sofrer manutenções adaptativas ou corretivas. Para uma boa funcionalidade da *Gestão de Componentes* é de fundamental importância o registro das atualizações das diversas versões dos ativos de software, pois usuários que estão trabalhando com uma versão do ativo devem ser comunicados da atualização da versão, pois as versões podem sofrer alterações relevantes, que podem às vezes alterar requisitos não funcionais e/ou funcionais, interface, detre outras.

Acredita-se que o uso da ontologia de componentes proposta neste trabalho possa proporcionar um grande ganho na pesquisa e recuperação de componentes, pois permite uma hierarquização das classes e subclasses e a vinculação de componentes a uma ou mais classes. Esta associação possibilita que ao ser localizado um determinado componente através de uma consulta simples ao seu nome, a descrição, ou de acordo com alguma faceta, possamos através de inferência expandir a busca por outros componentes da mesma categoria, ou das categorias imediatamente superiores à categoria do componente solicitado (expandindo a busca) ou ainda das categorias imediatamente inferiores da categoria do componente solicitado (restringindo a busca). A Figura 3.5 mostra um formulário de um aplicativo cliente-servidor que, através de comandos SQL, faz uma busca à base de dados relacional do repositório e que permite a busca semelhantemente à utilização de uma máquina de inferência. Este tipo de solução também pode ser implementado na forma de um sistema multi-agente, o que torna o sistema mais modular e com maior escalabilidade.

Todas as extensões propostas podem facilitar a catalogação e a busca, mas devido à falta de tempo, não foi possível seu uso em produção para a realização de uma validação na prática.

4.2 Trabalhos Futuros

A extensão da especificação RAS assim como a ontologia e o agente de busca descritos neste trabalho fazem parte de uma arquitetura mais ampla que será composta por um agente de catalogação de componentes e por um agente mantenedor da ontologia.

O desenvolvimento de plug-ins para os ambientes de desenvolvimento será facilitado pelo desenvolvimento de um web service para servir como ponto de acesso. O objetivo geral da arquitetura é facilitar o reuso de componentes e por meio do provimento de facilidades de catalogação e busca.

As extensões propostas podem permitir estender as funcionalidades do aplicativo usado para cadastro e pesquisa de ativos de software, tais como: gestão de usuários, gestão de estatística, gestão de recuperação conforme descritos nos casos

de uso levantados como requisitos do repositório, descritos na Tabela 3.3 com a descrição dos casos de uso.

Referências Bibliográficas

AGRESTI, W. W. and MCGARRY, F. E. (1998). **The Minnowbrook workshop on software reuse: a summary report**. *IEEE COMPUTER SOCIETY PRESS*, pages 33–40.

BELLIFEMINE, F., POGGI, A., and RIMASSA, G. (2001). **JADE: a FIPA2000 compliant agent development environment**. In *AGENTS '01: PROCEEDINGS OF THE FIFTH INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS*, pages 216–217, New York, NY, USA. ACM.

BOLDYREFF, C., NUTTER, D., and RANK, S. (2002). Open-Source Artefact Management. In ***Meeting Challenges and Surviving Success: The 2nd Workshop on Open Source Software Engineering***, pages 3 – 10, Boulder, CO, USA. ACM. Available online at http://opensource.ucc.ie/icse2002/; visited January 14th 2005.

BORST, W. N. (1997). ***Construction of Engineering Ontologies for Knowledge Sharing and Reuse***. PhD thesis, Centre for Telematica and Information Technology, University of Twente, Enschede.

CARLSON, B., STACK, C., STRONG, C., WEINAND, D., BACON, E., BOOCH, G., LARSEN, G., JACOBSEN, I., CONALLEN, J., GREEN, J., KEREKES, J., CHEESMAN, J., STEELE, J., GINBAYASHI, J., DELANO, L., AMAR, L., LECLERK, M., BOYETTE, N., VAGAPARTY, K., and RIVETT, P. (2005). **Reusable Asset Specification**. OMG Available Specification Version 2.2, Object Management Group.

CHRISTENSEN, H. B. (1999). **The Ragnarok Architectural Software Configuration Management Model**. In *HICSS '99: PROCEEDINGS OF THE THIRTY-SECOND ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES-Volume 8*, page 8067, Washington, DC, USA. IEEE Computer Society.

CORAZZON, R. (2002). **Ontology: A resource guide for philosophers**. In *Ontology: A resource guide for philosophers*. http://www.formalontology.it/. Acesso em: 2007.

DIAZ, R. P. (1991). **Implementing faceted classification for software reuse**. *Commun. ACM*, 34(5):88–97.

D'SOUZA, D. F. and WILLS, A. C. (1999). ***Objects, components, and frameworks with UML: the catalysis approach***. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

EMMERICH, W. and KAVEH, N. (2002). **Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model**. *PROCEEDINGS OF 24TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING*. citeseer.ist.psu.edu/emmerich02component.html.

EZRAN, M., MORISIO, M., and TULLY, C. (2002). ***Practical software reuse***. Springer-Verlag, London, UK.

GARCIA, V. C., DURÃO, F. A., LUCRÉDIO, D., de Lemos MEIR, S. R., and et al. (2005). **Especificação, Projeto e Implementação de uma Arquitetura para um Engenho de Busca de Componentes**. *WDBC2005 - 5o. WORKSHOP DE DESENVOLVIMENTO BASEADO EM COMPONENTES*.

GRUBER, T. R. (1993). **A translation approach to portable ontology specifications**. Technical Report 2, Knowledge Acquisition, London, UK.

GUARINO, N. (1998). **Formal Ontology and Information Systems**. *PROCEEDINGS OF THE 1ST INTERNATIONAL CONFERENCE ON FORMAL ONTOLOGIES IN INFORMATION SYSTEMS, FOIS'98*, pages 3–15.

GUERRA, P. A. C., MORONTE, T. C., TOMITA, R. T., and RUBIRA, C. M. F. (2005). **Um Modelo Conceitual e uma Terminologia para o Desenvolvimento Baseado em Componentes e Centrado na Arquitetura de Software**. *WDBC2005 - 5o. WORKSHOP DE DESENVOLVIMENTO BASEADO EM COMPONENTES. SBC - SOCIEDADE BRASILEIRA DE COMPUTAÇÃO.Juiz de Fora, MG, Brasil. 2005.*

HINZ, V. T. and PALAZZO, L. A. M. (2006). **Colaboração em Sistemas Multiagentes Modelados por Ontologias**.

HOOPER, J. W. and CHESTER, R. O. (1991). ***Software Reuse: Guidelines and Methods***. Perseus Publishing.

KOZACZYNSKI, W., WULFERT, W., PATTERSON, S., and et al (2003). **RAS - Reusable Asset Specification Rational Software Corporation and Catapulse**. Technical report, IBM, Draft Recommendation.

LER, C. and ROSENBLUM, D. (2001). **Wren—An Environment for Component-Based Development**. *IN JOINT 8TH EUROPEAN SOFTWARE ENGINEERING CONFERENCE (ESEC)AND 9TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING - FSE-9.*

LUCREDIO, D., do PRADO, A. F., and de ALMEIDA, E. S. (2004). **A Survey on Software Components Search and Retrieval**. *EUROMICRO '04: PROCEEDINGS OF THE 30TH EUROMICRO CONFERENCE (EUROMICRO'04)*, pages 152–159.

MOREIRA, A., ALVARENGA, L., and de Paiva OLIVEIRA, A. (2004). **Thesaurus and Ontologies: a study over the definitions found in**

the computer and information science literature, by means of analytical-synthetic method. *KNOWLEDGE ORGANIZATION*, pages 231–244. http://www.dgz.org.br/dez04/Art$_0$1.*htm*.

MURTA, L. G. P. (2004). **Odyssey-SCM: Uma Abordagem de Gerência de Configuração de Software para o Desenvolvimento Baseado em Componentes**.

PEREZ, A. and BENJAMINS, V. (2006). **Overview of Knowledge Sharing and Reuse Components: Ontologies and Problem-Solving Methods**. *IJCAI-99 WORKSHOP ON ONTOLOGIES AND PROBLEM-SOLVING METHODS - KRR5*.

PINHO, H. S. (2006). **RIGEL - Um Repositório com Suporte para Desenvolvimento Baseado em Componentes**. Dissertação de Mestrado Profissional em Computação. Universidade Estadual de Campinas.

SAMETINGER, J. (1997). ***Software Engineering with Reusable Components***. Springer-Verlag New York, Inc., New York,NY, USA.

SEACORD, R., HISSAM, S., and WALLNAU, K. (1998). ***Agora: A search engine for software components***. Software Engineering Institute, Pittsburgh, PA 15213-3890.

SILVA, C. C. (2004). **Um arcabouço para conexões de componentes de software**.

SIRIN, E., PARSIA, B., GRAU, B. C., and KALYANPUR, A. (2007). **Pellet: A practical OWL-DL reasoner**. *Journal of Web*, 5(2).

SMITH, M. K., WELTY, C., and MCGUINNESS, D. L. (2004). **OWL Web Ontology Language - Guide**. W3c:rec, W3C. http://www.w3.org/TR/2004/REC-owl-guide-20040210/.

SOUSA, L. G. and LEITE, J. C. (2004). **Utilizando ontologia para a descrição da interação em componentes de interface de usuário**. *VI SIMPÓSIO SOBRE FATORES HUMANOS EM SISTEMAS COMPUTACIONAIS*.

SOWA, J. F. (2000). ***Knowledge Representation: Logical, Philosophical, and Computational Foundation***. MIT Press, Pacific Grove, CA.

SUGUMARAN, V. and STOREY, V. C. (2003). **A semantic-based approach to component retrieval**. *SIGMIS DATABASE*, 34(3):8–24.

VITHARANA, P., ZAHEDI, F. M., and JAIN, H. (2003). **Knowledge-based repository scheme for storing and retrieving business components: a theoretical design and an empirical analysis**. *SOFTWARE ENGINEERING, IEEE TRANSACTIONS ON*, 29(7):649–664.

WOOLDRIDGE, M. (2002). ***Introduction to MultiAgent Systems***. John Wiley and Sons, New York, NY, USA.

WOOLDRIDGE, M. and JENNINGS, N. R. (1995). **Intelligent Agents: Theory and Practice**. *KNOWLEDGE ENGINEERING REVIEW*, 10(2):115–152.

YE, Y. (2001). ***Supporting component-based software development with active component repository systems***. PhD thesis, University of Colorado at Boulder, Boulder, CO, USA. Director-Gerhard Fischer.