“RIGEL - Um Repositório com Suporte para Desenvolvimento Baseado em Componentes”

Helder de Sousa Pinho

Trabalho Final de Mestrado Profissional em Computação

Instituto de Computação
Universidade Estadual de Campinas

"RIGEL - Um Repositório com Suporte para Desenvolvimento Baseado em Componentes"

Helder de Sousa Pinho

24 de fevereiro de 2006

Banca Examinadora:

- **Profª. Drª. Cecília Mary Fischer Rubira (Orientadora)**
  Instituto de Computação – (IC/UNICAMP)
- **Prof. Dr. Marcos Lordello Chaim**
  Escola de Artes, Ciências e Humanidades – (EACH/USP Leste)
- **Profª. Drª. Eliane Martins**
  Instituto de Computação – (IC/UNICAMP)
- **Profª. Drª. Ariadne Maria Brito Rizzoni Carvalho (Suplente)**
  Instituto de Computação – (IC/UNICAMP)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Pinho, Helder de Sousa
P655r RIGEL – Um repositório com suporte para Desenvolvimento Baseado em Componentes / -- Campinas, [S.P. :s.n.], 2006.

Orientador : Cecília Mary Fischer Rubira
Trabalho final (mestrado profissional) - Universidade Estadual de Campinas, Instituto de Computação.

1. Componentes de software. 2. Metadados. 3. Engenharia de software. 4. Software – Desenvolvimento. 5. Software - Reutilização . I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: RIGEL – A repository with support for Component Based Development

Palavras-chave em inglês (Keywords): 1. Component software. 2. Metadata. 3. Software engineering. 4. Computer software – Development. 5. Sotware reusability.

Área de concentração: Engenharia de Computação

Titulação: Mestre em Computação

Banca examinadora: Profa. Dra. Cecília Mary Fischer Rubira (IC-UNICAMP)
Prof. Dr. Marcos Lordello Chaim (EACH/USP Leste)
Profa. Dra. Eliane Martins (IC-UNICAMP)

Data da defesa: 24/02/2006

“RIGEL - Um Repositório com Suporte para Desenvolvimento Baseado em Componentes”

Este exemplar corresponde à redação final do Trabalho Final devidamente corrigido e defendido por Helder de Sousa Pinho e aprovado pela Banca Examinadora.

Campinas, 24 de fevereiro de 2006.

Profª. Drª. Cecília Mary Fischer Rubira
(Orientadora)

Trabalho Final apresentado ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Computação na área de Engenharia de Computação

Aos meus pais,
Francisco e Hêda,

Agradecimentos

Agradeço principalmente a Deus, se eu cheguei até aqui foi por que Ele me guiou e me protegeu.

Aos meus pais, pelo amor, incentivo nesses anos todos. Agradeço aos meus irmãos pela cumplicidade, amizade e apoio.

Agradeço a minha orientadora, Cecília Rubira pela paciência e disposição para me orientar. Obrigado pelas aulas, conversas, correções, sugestões e dicas.

Um obrigado todo especial a Alessandra, que com carinho e amor sempre me ajudou a superar os momentos mais difíceis nesta jornada.

Aos professores do IC pela dedicação, profissionalismo e entusiasmo durante todo o curso.

Aos funcionários do IC, especialmente à Claudia, Ione e Olívia pelos favores e pelos problemas resolvidos com toda atenção e simpatia durante o curso.

Agradeço a todos os meus colegas do Mestrado, por contribuírem de várias formas para este trabalho: Leonardo Tizzei, Patrick Brito, Rodrigo Tomita, Paulo Astério, Tiago Moronte, Ana Elisa Lobo, Janaina Ruas, Leonel, Júnia Neves, Douglas Marques, Ana Cabral, Eder Alves, Rodrigo Burger, Sandro Dangui, Paulo Tavares, Roberto Bresil, Paulo Martins, Alessandro Santos e Valeria Reis.

Resumo

O desenvolvimento baseado em componente (DBC) permite que uma aplicação seja construída pela composição de componentes de software que já foram previamente especificados, construídos e testados, resultando em ganhos de produtividade e qualidade no software produzido. Para haver reuso de componentes, é necessário que usuários consigam procurar e recuperar componentes previamente especificados ou implementados Um repositório de componentes é essencial para possibilitar tal reuso. Interoperabilidade é um requisito importante para repositórios, mas nem todas as ferramentas a tratam com a devida relevância. O modelo de metadados de um repositório para DBC deve contemplar características de componentes, tais como interface e separação entre especificação e implementação.

Este trabalho apresentou o Rigel, um repositório de bens de software reutilizáveis com suporte para desenvolvimento baseado em componentes. O Rigel apresenta características que facilitam atividades executadas durante o desenvolvimento de sistemas baseados em componentes, tais como pesquisa, armazenamento e recuperação de bens e integração com CVS.

O padrão RAS foi adotado como o formato de metadados e de empacotamento de bens, facilitando a integração do Rigel com outros sistemas. O modelo de metadados do RAS foi estendido para apoiar um modelo conceitual de componentes e arquitetura de software. Esta adaptação resultou na criação de quatro novos *profiles* RAS, para apoiar bens relacionados à DBC: componente abstrato, componente concreto, interface e configuração arquitetural. Um estudo de caso foi conduzido a fim de mostrar como o Rigel apóia um processo de desenvolvimento baseado em componentes. Conclui-se que as características do repositório Rigel facilitam um desenvolvimento baseado em componentes.

Abstract

The component based development (CBD) permits an application to be built by composition of previously specified, build and tested components, resulting in increases in productivity and quality of the produced software. In order to make the reuse of components happen, it is necessary that users are able to search and retrieve previously specified or implemented components. A component repository is important to support this reuse. Interoperability is an important requirement for repositories, but not all the tools consider it with the required relevance. The metadata model of a CBD repository must handle components features, such as interface and separation between specification and implementation.

This work presents Rigel, a repository of reusable software assets with a support for component based development. Rigel presents features that make activities performed during the development of component based systems easier, such as search, storage and retrieval of assets and CVS integration.

RAS standard was adopted as the asset metadata and packaging format, making Rigel integration with other systems easier. The RAS metadata model was extended to support a conceptual model of components and software architecture. This adaptation resulted in the creation of four new RAS profiles to support CBD related assets: abstract component, concrete component, interface and architectural configuration. A case study was conducted in order to show how Rigel supports a CBD process. We also conclude that Rigel repository features make the component based development easier.

"Mestre não é quem sempre ensina, mas quem de repente aprende."
(Guimarães Rosa).

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1 Introdução

Os sistemas de software têm se tornado cada vez maiores e mais complexos. Desenvolver tais sistemas é um desafio cada vez maior, pois além da complexidade intrínseca do processo, existem exigências para implementar sistemas de alta qualidade a um baixo custo, num espaço de tempo curto. A engenharia de software aborda tais problemas, através da criação de mecanismos que possibilitam uma melhor produção de sistemas de software. Um processo de desenvolvimento de software define um conjunto de passos, métodos, técnicas e práticas de engenharia de software que reduzem os riscos, aumentam a produtividade dos recursos empregados e melhoram a confiabilidade do software produzido [Jacobson+99].

A idéia de reusar partes de sistemas já existentes surgiu para atacar problemas como diminuir custos e manejar a complexidade no desenvolvimento de software, e melhorar a qualidade dos sistemas produzidos [Mcllroy69]. A divisão de um sistema em módulos foi a base para o surgimento de uma metodologia de desenvolvimento de software fundamentada na composição de componentes de software chamada **desenvolvimento baseado em componentes** (DBC). **Componentes de software** são unidades de software desenvolvidas e testadas separadamente e que podem ser integradas com outras para construir algo com maior funcionalidade [Szyperski97].

O desenvolvimento baseado em componente permite que uma aplicação seja construída pela composição de componentes de software que já foram previamente especificados, construídos e testados, o que resulta em ganho de produtividade e qualidade no software produzido. O aumento da produtividade advém da reutilização de componentes existentes em novos sistemas, enquanto que o aumento da qualidade advém do fato do componente já ter sido utilizado, modificado e testado em outros sistemas, eliminando boa parte dos erros que poderiam ocorrer. Com isso os produtos finais não são mais produtos de software monolíticos e fechados, mas baseados em componentes que podem ser integrados com outros produtos [Larsson99].

Um componente de software possui **interfaces** especificadas contratualmente e dependências de contexto explícitas [Szyperski97]. Uma interface identifica um ponto de interação entre um componente e o seu ambiente [Garlan00]. Componentes implementam uma ou mais interfaces providas, que identificam os serviços oferecidos a outros componentes e declaram os serviços dos quais ele depende para funcionar, isto é, suas interfaces requeridas [Bronsard+97]. As interfaces providas e requeridas descrevem explicitamente a especificação de um componente, atendendo ao seu contrato. Componentes encapsulam seu projeto e implementação e expõem a sua especificação para os seus clientes [D'Souza99], desta forma a implementação de um componente é separada da sua especificação.

A separação entre especificação e implementação de componentes permite a classificação da atividade de desenvolvimento de software em dois grupos independentes: a produção de componentes de software e a integração de sistemas usando componentes prontos. Dessa forma, podemos definir dois pontos de vista para o desenvolvimento de software: o desenvolvimento para reuso e o desenvolvimento com reuso [Moore91]. O desenvolvimento para reuso, ou Engenharia de Domínio, se preocupa com a geração de componentes reutilizáveis em um determinado domínio. O desenvolvimento com reuso, ou Engenharia da Aplicação, se preocupa em construir aplicações reutilizando os componentes criados na Engenharia de Domínio.

Um **repositório de bens de software reutilizáveis** é o ponto centralizado de acesso e armazenamento de bens reutilizáveis. No contexto deste trabalho, **bem** é uma solução para um problema de desenvolvimento de software, e mais especificamente [RAS04], um **bem reutilizável** é uma solução para um problema recorrente [RAS04]. Interfaces, implementações de um sistema ou de parte de um sistema, especificações de um sistema são exemplos de bens. **Artefato** é um elemento lógico ou físico de um bem. Um bem lógico contém ao menos um artefato físico. Artefatos físicos correspondem a arquivos no sistema de arquivos [RAS04]. Um diagrama de classes é um exemplo de um artefato que faz parte de um bem que contém uma especificação de um sistema; outro exemplo de artefato é um arquivo contendo código fonte, sendo este parte de um bem que contém uma implementação de um sistema. Um componente geralmente é mapeado para um bem, ou para um conjunto de bens. Devido ao fato de componentes sempre terem características particulares como a separação entre implementação e especificação, um componente não é um bem qualquer. Portanto, um bem nem sempre é um componente ou parte de um componente.

Mais especificamente temos repositórios de componentes, cujos bens são produtos de um desenvolvimento baseado em componentes. Um repositório de componentes apóia a pesquisa, o fornecimento e o gerenciamento de componentes para a construção de aplicações de negócios [Sametinger97]. Os repositórios de componentes atuam como elos entre o desenvolvimento com reuso e o desenvolvimento para reuso. Esses repositórios funcionam como um ponto de conexão central onde os produtores de componentes depositam os artefatos produzidos por exemplo, especificações e implementações de componentes, e consumidores de componentes os recuperam para integrá-los em seus sistemas. Repositórios de componentes armazenam, registram e gerenciam todos os artefatos produzidos durante o ciclo de vida dos componentes. Pesquisas avançadas e navegação em informações sobre componentes são funcionalidades utilizadas por repositórios para apoiar o reuso de componentes em um processo de desenvolvimento baseado em componentes [Luqi99] [Seacord99] [Mili98].

Produtores de componentes e consumidores de componentes podem estar localizados em diferentes locais e organizações. O espalhamento de produtores e consumidores de componentes é um dos fatores que os induzem a utilizarem diferentes repositórios de componentes e diferentes ambientes integrados de desenvolvimento de software (IDE – *Integrated Development Environment*), causando um isolamento entre produtores e consumidores. Um **ambiente integrado de desenvolvimento** de software é formado por um conjunto integrado de ferramentas que apóiam o desenvolvimento de software [Ambriola97]. A interoperabilidade entre repositórios de componentes e ferramentas de desenvolvimento auxilia a conexão entre produtores e consumidores de componentes, possibilitando mais oportunidades de reutilizações de software. **Interoperabilidade** é a capacidade de interagir com outros sistemas de acordo com o que foi especificado [ISO2005].

Desta forma, a efetiva reutilização de componentes em larga escala requer o apoio de repositórios de componentes que: (i) permitam consumidores de componentes pesquisar e recuperar componentes, (ii) possam interoperar para formação de uma ampla base de componentes de software disponíveis para os integradores, mesmo sendo de diferentes produtores e estando geograficamente distribuídos, (iii) permitam produtores de componentes adicionar seus artefatos ao repositório.

1.1 Motivação e Problema

O desenvolvimento baseado em componentes visa fornecer um conjunto de procedimentos, ferramentas e notações que possibilitem, ao longo do processo de software, a ocorrência tanto da produção de novos componentes quanto a reutilização de componentes existentes. Os processos de desenvolvimento de software convencionais não incluem explicitamente nos seus objetivos a reutilização de componentes de software. Através desses processos, essa reutilização pode ocorrer apenas ocasionalmente e de maneira não sistemática. Por outro lado, um processo de DBC deve, como um dos seus principais objetivos, promover essa reutilização de componentes em larga escala em diversas fases do desenvolvimento de software.

Atividades como especificar, implementar e procurar componentes são importantes para processos de desenvolvimento baseado em componentes. Para haver um reuso efetivo de componentes, é necessário que usuários consigam procurar e recuperar componentes previamente especificados ou implementados. Um repositório de componentes é essencial para apoiar tais atividades. Componentes prontos devem ser armazenados em repositórios para sua posterior utilização. A pesquisa de componentes em um repositório possibilita um integrador de sistemas a encontrar componentes já existentes, a fim de reutilizá-lo na construção de novos sistemas.

Repositórios são igualmente importantes durante o desenvolvimento de um componente, antes da finalização de sua implementação. Muitas vezes, diferentes pessoas de uma equipe de desenvolvimento trabalham nas diferentes fases do desenvolvimento do mesmo componente. Neste caso, o repositório é o ponto comum, onde desenvolvedores armazenam e recuperam especificações parciais de um componente.

Em um processo DBC, é necessário se preocupar sistematicamente com o controle da interação entre os componentes. Essa visão interativa do sistema é materializada na arquitetura de software. A **arquitetura de software** engloba a definição das estruturas gerais, descreve os componentes do sistema, as interações entre eles, e busca uma solução técnica para um problema de negócio [Shaw+96].

Durante o ciclo de desenvolvimento, os componentes vão evoluindo desde sua especificação até o final de seu ciclo de desenvolvimento. Devido à evolução contínua dos bens

produzidos, se faz necessário ter um controle entre as versões produzidas. Cada evolução do componente deve ser armazenada em um repositório, de modo que seja possível continuar o próximo estágio do desenvolvimento a partir do estado anterior do componente. Gerência de Configuração de Software (GCS) é o controle da evolução de sistemas complexos. GCS é a área que nos permite manter sob controle os produtos de software em evolução, e assim contribui para satisfazer as restrições de qualidade e tempo [Estublier94]. Para que exista um controle adequado de todas as versões de arquiteturas de componentes e de componentes é necessária uma integração do repositório com um sistema GCS. Estes sistemas (GCS) permitem o rastreamento das diversas versões de artefatos.

No contexto deste trabalho, interoperabilidade pode ser vista como a integração entre repositórios diferentes ou entre repositórios e IDEs. Apesar de interoperabilidade ser uma característica importante, nem todos os repositórios a tratam com a devida relevância. Algumas abordagens chegam a ignorar esse requisito, enquanto outras o oferecem de maneira incompleta.

Um maior número de componentes disponíveis significa mais opções de escolha para integradores de sistemas, aumentando a escala em que o reuso acontece. Porem, para maximizar a chance de reuso dos componentes de um repositório, é necessário utilizar padrões que possibilitem a compatibilidade entre diversos sistemas, sejam eles repositórios ou IDEs [Voth04]. Esta compatibilidade permite que um bem possa estar disponível em mais de um repositório. A especificação de um formato de dados que seja compatível com várias ferramentas ainda é um desafio nesta área.

Além da interoperabilidade, existem outros fatores importantes na definição de formato de dados para armazenamento de componentes de software em repositórios. A definição dos metadados que descrevem o componente armazenado deve contemplar características, como precisão, eficiência, generalidade e interface com o usuário [Mili98].

A descrição de componentes deve também incluir características das suas partes constituintes [Vitharana03]. Essa questão da granularidade também é importante em relação ao reuso de software. Partes de componentes como componentes requeridos ou interfaces podem ser mais facilmente reutilizadas, se elas estiverem devidamente armazenadas como bens. A possibilidade de pesquisa, identificação e recuperação isolada destas partes de componentes é importante para promover o reuso com menor granularidade.

1.2 Solução Proposta

Neste trabalho desenvolvemos o Rigel[1] - um repositório de bens de software reutilizáveis em geral, com suporte a componentes. O projeto e a implementação do repositório Rigel consideraram as seguintes características:

- O repositório Rigel é também um repositório de componentes, através do suporte a um modelo de metadados que considere características das partes constituintes de componentes tais como: interfaces, especificação de componentes, implementação de componentes e arquitetura de componentes.
- Apoio às operações básicas de um processo de desenvolvimento de componentes: pesquisar componentes, armazenar especificações e implementações de componentes, recuperar componentes.
- Uma arquitetura interoperável, possibilitando a integração com outros repositórios ou IDEs.
- Integração a um sistema de GCS, possibilitando o controle de versão de componentes.

O modelo de metadados deste repositório foi estendido para acomodar conceitos específicos de desenvolvimento baseado em componentes. O trabalho de Guerra [Guerra05] propõe um modelo conceitual para a representação de conceitos de DBC e arquitetura de componentes. Conceitos como componente de software, interface requerida, interface provida e arquitetura de software são tratados em seu trabalho. Adaptamos estes modelos conceituais aos metadados de Rigel, de modo a acomodar no repositório as informações relativas a conceitos de desenvolvimento baseado em componentes.

Questões importantes em um repositório, tais como interoperabilidade entre repositórios e IDEs, foram resolvidas através da adoção de um padrão para o formato dos metadados dos

[1] Rigel é a estrela mais brilhante da constelação de Orion, e a sétima mais brilhante do céu. É chamada também de Orion Beta.

bens de um repositório, assim como possuir uma arquitetura interna capaz de apoiar integrações com outros repositórios ou ferramentas de desenvolvimento.

O modelo RAS [RAS04] é uma especificação proposta pela Rational e aceita pela OMG como padrão para especificação de bens. Este padrão especifica um modelo de metadados extensível que deverá representar bens de software reutilizáveis. A partir do modelo inicial do RAS, adicionamos quatro novas extensões ao modelo de metadados RAS, para apoiar o modelo conceitual proposto por Guerra [Guerra05]:

- Componente Abstrato – especifica um bem que representa um componente abstrato. Um **componente abstrato** é composto por uma especificação abstrata que pode ser materializada em diferentes implementações [Gibson00]. A especificação de componente é uma abstração que define o comportamento observável externamente de um componente de software, de forma independente de qualquer implementação.
- Componente Concreto – especifica um bem que representa um **componente concreto**, que é uma implementação de comportamento [Gibson00] que pode ser instanciado para compor um ou mais sistemas executáveis. Por exemplo: um arquivo .jar contendo um conjunto de classes Java compiladas (bytecode). Um tipo concreto de componente está sempre associado a um tipo abstrato de componente, que é a sua especificação.
- Interface – especifica um bem que representa uma interface de um componente.
- Configuração – especifica um bem que representa uma configuração arquitetural concreta. Elemento arquitetural é uma abstração que representa uma parte de um sistema de software responsável por determinado comportamento ou propriedade desse sistema. Uma **configuração arquitetural concreta** descreve um determinado aspecto de uma arquitetura de software através da sua decomposição num conjunto de elementos arquiteturais interconectados [Bass03].

Na Figura 1 temos um exemplo de um conjunto de componentes. ComponenteA e ComponenteB são componentes abstratos, e conseqüentemente só possuem uma especificação. ComponenteConcretoA1 é um componente concreto que implementa ComponenteA, analogamente temos ComponenteConcretoB1 implementando ComponenteB. Interface_A é

uma interface provida de ComponenteA e um interface requerida de ComponenteB. No formato de metadados proposto por Rigel, Interface_A é armazenada separadamente como um bem do tipo Interface. ComponeteA e ComponenteB são armazenados separadamente no repositório como bens do tipo componente abstrato. ComponenteConcretoA1 e ComponenteConcretoB1 são armazenados separadamente como bens do tipo componente concreto. Finalmente temos o bem Configuração que tem referências para os bens ComponenteConcretoA1 e ComponenteConcretoB1.

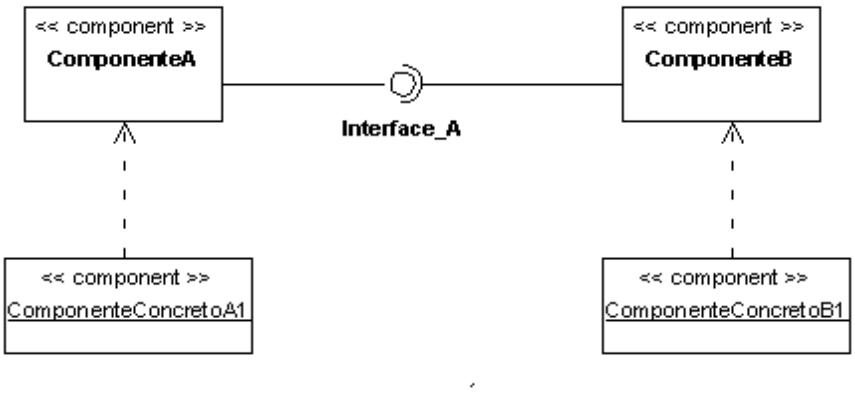

Figura 1 – Exemplo de um configuração arquitetural de componentes

O padrão RAS especifica que os metadados sejam armazenados em XML [W3XML] (Seção 2.5.2). Na implementação do Rigel projetamos uma mecanismo de busca capaz de indexar e pesquisar metadados de bens no formato RAS.

O Rigel possui uma arquitetura interna dividida em camadas. Interfaces Java RMI [RMI] são disponibilizadas pelos componentes da camada de sistema – a camada responsável pela acesso aos serviços do repositório [Cheesman+00]. Estas interfaces permitem que o Rigel seja integrado com outros repositórios ou IDEs.

O suporte a versionamento de artefatos é alcançado através da integração do repositório Rigel com o CVS (*Concurrent Version System*) [CVS] – um sistema de GCS de código aberto.

O repositório Rigel é parte de um ambiente para desenvolvimento de componentes produzido pelo grupo de alunos da Profa. Cecilia Rubira, no Instituto de Computação. Uma das ferramentas deste grupo é o Bellatrix[2], sendo Bellatrix também uma estrela da constelação Orion.

[2] Bellatrix é a terceira estrela mais brilhante da constelação Orion, também conhecida como Gamma Ori.

O nome Rigel surgiu para transmitir uma idéia de conjunto, pois Rigel é também o nome de uma outra estrela da constelação Orion.

1.3 Organização deste Documento

Este documento foi organizado em sete capítulos da seguinte forma:

- **Capítulo 2 – Fundamentos** : O segundo capítulo apresenta fundamentos teóricos de desenvolvimento baseado em componentes, gerência de configuração, repositórios e algumas tecnologias relativas a repositórios, assim como uma comparação entre vários repositórios.
- **Capítulo 3 – Requisitos do Repositório Rigel**: Neste capítulo, são apresentados os requisitos para o repositório Rigel. Inicialmente são descritos os requisitos funcionais, e logo depois, os requisitos de qualidade.
- **Capítulo 4 – Projeto do Repositório Rigel**: Este capítulo descreve o projeto do repositório Rigel. São apresentados o projeto arquitetural, o modelo de metadados e o projeto detalhado.
- **Capítulo 5 – Implementação do Repositório Rigel**: Neste capítulo, são apresentados detalhes de implementação do repositório Rigel.
- **Capítulo 6 – Estudos de Caso usando o Rigel**: Neste capítulo, são apresentados estudos de caso envolvendo o repositório Rigel. O primeiro estudo de caso exercita o uso do repositório com um processo DBC. Já o segundo, mostra um exemplo de utilização de componentes prontos no Rigel.
- **Capítulo 7 – Conclusões e Trabalhos Futuros**: O último capítulo apresenta as conclusões deste trabalho, suas contribuições e sugestões para trabalhos futuros.

Capítulo 2 Fundamentos de Engenharia de Software

Este capítulo estabelece parte da terminologia e define vários conceitos utilizados neste documento. A seção 2.1 apresenta conceitos relativos a desenvolvimento baseado em componentes e processos de desenvolvimento. A seção 2.2 apresenta uma comparação entre repositórios de componentes. Já a seção 2.3 apresenta conceitos relativos à arquitetura de software, inclusive apresentando um modelo conceitual para DBC e arquitetura de software. A seção 2.4 apresenta conceitos sobre Gerência de Configuração de Software. Finalmente, a seção 2.5 apresenta tecnologias relacionadas à implementação de componentes.

2.1 Desenvolvimento Baseado em Componentes

Em 1976 [Mcllroy69], surgiu a idéia de utilizar componentes já existentes para construir sistemas de software, diminuindo custos, melhorando a qualidade e manejando a complexidade destes sistemas. Mas, o interesse por desenvolvimento baseado em componentes (DBC) cresceu somente após a realização do WCOP'96 [WCOP96], vinte anos depois. Atualmente, prazos mais curtos e melhor qualidade são os fatores que mais incentivam a utilização de DBC.

No DBC, uma aplicação é construída a partir da composição de componentes de software previamente especificados, construídos e testados, proporcionando um aumento na produtividade e na qualidade do software produzido. Esse ganho na produtividade acontece através da reutilização de componentes existentes na construção de novos sistemas. O aumento da qualidade é uma conseqüência do fato dos componentes utilizados já terem sido empregados e testados em outros contextos [Larsson99].

Componentes de software são unidades de software desenvolvidas e testadas separadamente e que podem ser integradas com outras para construir algo com maior

funcionalidade [Szyperski97], conforme definido no Capítulo 1, página 1. Componentes implementam uma ou mais interfaces providas, que identificam os serviços oferecidos a outros componentes e declaram os serviços dos quais ele depende para funcionar, suas interfaces requeridas [Bronsard+97]. Definições sobre componentes podem variar um pouco de autor para autor, porém, um aspecto muito importante é sempre ressaltado na literatura: um componente deve encapsular dentro de si seu projeto e implementação, além de oferecer interfaces bem definidas para o meio externo. O baixo acoplamento, resultado dessa estratégia, proporciona muitas flexibilidades, tais como: (i) facilidade de montagem de um sistema a partir de componentes *COTS* (*Common Off-The-Shelf*) ; e (ii) facilidade de substituição de componentes que implementam interfaces equivalentes.

Um conector é um tipo de componente dedicado a mediar interações entre outros componentes. Um conector pode ser tão complexo quanto um componente [Bishop96].

2.1.1 O Processo de Desenvolvimento Baseado em Componentes

O processo de desenvolvimento baseado em componentes visa fornecer um conjunto de procedimentos, ferramentas e notações que possibilitem, ao longo do processo de software, a ocorrência tanto da produção de novos componentes quanto a reutilização de componentes existentes. Alguns processos de desenvolvimento de software baseado em componentes têm surgido na literatura, como é o caso do Catalysis [D'Souza99] e o UML Componentes [Cheesman+00]. Um processo de desenvolvimento de software é um conjunto de etapas, métodos, técnicas e práticas que empregam pessoas para o desenvolvimento e manutenção de um software e seus artefatos associados (planos, documentos, modelos, código, casos de testes, manuais, etc.). É composto de boas práticas de engenharia de software que conduzem o desenvolvimento do software, reduzindo os riscos e aumentando a confiabilidade dos sistemas [Jacobson+99].

Embora existam muitos processos de desenvolvimento de software, algumas atividades fundamentais estão presentes em todos eles. São elas: levantamento de requisitos, projeto, implementação e testes. A fase de levantamento de requisitos foca na identificação dos serviços que devem ser automatizados, as informações que devem ser processadas, além de questões como desempenho, confiabilidade, disponibilidade e segurança. Durante a fase de projeto, os

desenvolvedores definem como os dados serão estruturados, como a funcionalidade será descrita através de uma arquitetura de software e em como as interfaces serão caracterizadas. Na implementação, o projeto é convertido para uma linguagem de implementação em forma de código fonte e, durante a etapa de testes, o sistema é validado para certificar-se de que os requisitos especificados estão todos implementados corretamente.

Um processo de desenvolvimento de software baseado em componentes possui algumas características adicionais aos processos convencionais. Essas características podem ser notadas no acréscimo de estágios técnicos ao processo convencional ou em uma ênfase maior em algumas práticas já realizadas nesses processos. Segundo Jacobson et. al. [Jacobson+99], um processo de desenvolvimento de software baseado em componentes geralmente inclui a definição de estratégias para:

- Separação de contextos a partir do modelo de domínios;
- Particionamento do sistema em unidades independentes (componentes);
- Identificação do comportamento interno dos componentes;
- Identificação das interfaces dos componentes;
- Definição de um kit de arquitetura, que inclua princípios e elementos que facilitem a conexão de componentes; e
- Manutenção de um repositório de componentes.

2.1.2 O Processo UML Components

O UML Components [Cheesman+00] é um processo de desenvolvimento de software baseado em componentes, que trata do problema de especificar e arquitetar sistemas baseados em componentes.

Uma característica importante de UML Components é o fato dele ser baseado na linguagem UML [11]. Esta escolha pragmática possibilita que usuários do método utilizem ferramentas já existentes para UML (tais como EclipseUML [Omondo] e o Rational Rose [Rational]) para a construção de seus sistemas baseados em componentes. O processo propõe extensões à linguagem UML através de seu mecanismo de estereótipos para a especificação das interfaces de componentes, de componentes propriamente ditos e de suas implementações.

O método UML Components é iterativo e define um conjunto de passos para que, a partir da especificação de um conjunto de casos de uso, seja construído um sistema de software baseado em componentes que simplifique o trabalho de lidar com mudanças (evolução / substituição de componentes).

As fases definidas pelo método são: (1) especificação, na qual são gerados um conjunto de especificações de componentes e uma arquitetura de componentes, (2) provisionamento, que garante que os componentes necessários estão disponíveis, seja através de seu desenvolvimento, aquisição ou reuso e (3) montagem, que compõe os diversos componentes entre si e com artefatos de software pré-existentes, incluindo a interface com o usuário. A Figura 2 mostra uma visão geral do processo. O UML Components não é um processo de desenvolvimento completo, já que não inclui atividades relacionadas ao processo gerencial e nem atividades de testes. UML Components enfatiza as etapas de análise, projeto e, em menor grau, implementação.

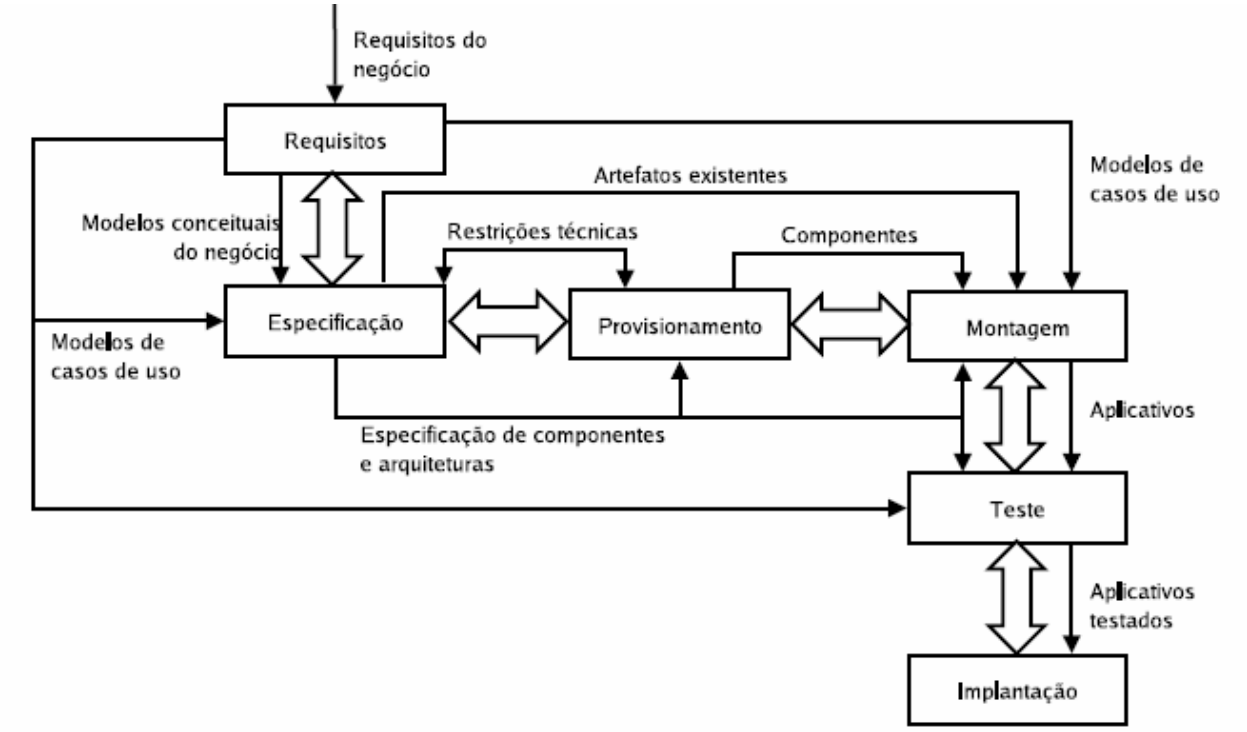

Figura 2 – Fases do Processo UML Components

A fase de especificação ainda é dividida nas seguintes sub-fases (Figura 3):

- Identificação de componentes, cujo objetivo é identificar um conjunto inicial de interfaces (e o nome de seus métodos) e componentes e combiná-los em uma

arquitetura inicial. As seguintes atividades fazem parte desta sub-fase: identificar interfaces de sistema e operações, desenvolver modelo de tipos do negócio, identificar interfaces de negócio e criar especificações de componentes iniciais e arquitetura

- Interação de componentes, na qual através de modelos de interação, são descobertas mais operações com suas assinaturas completas e as interfaces podem ser refinadas, agrupadas ou divididas. Esta sub-fase contém as seguintes atividades: descobrir operações de negócio, refinar interfaces e operações e refinar especificações de componentes e arquitetura

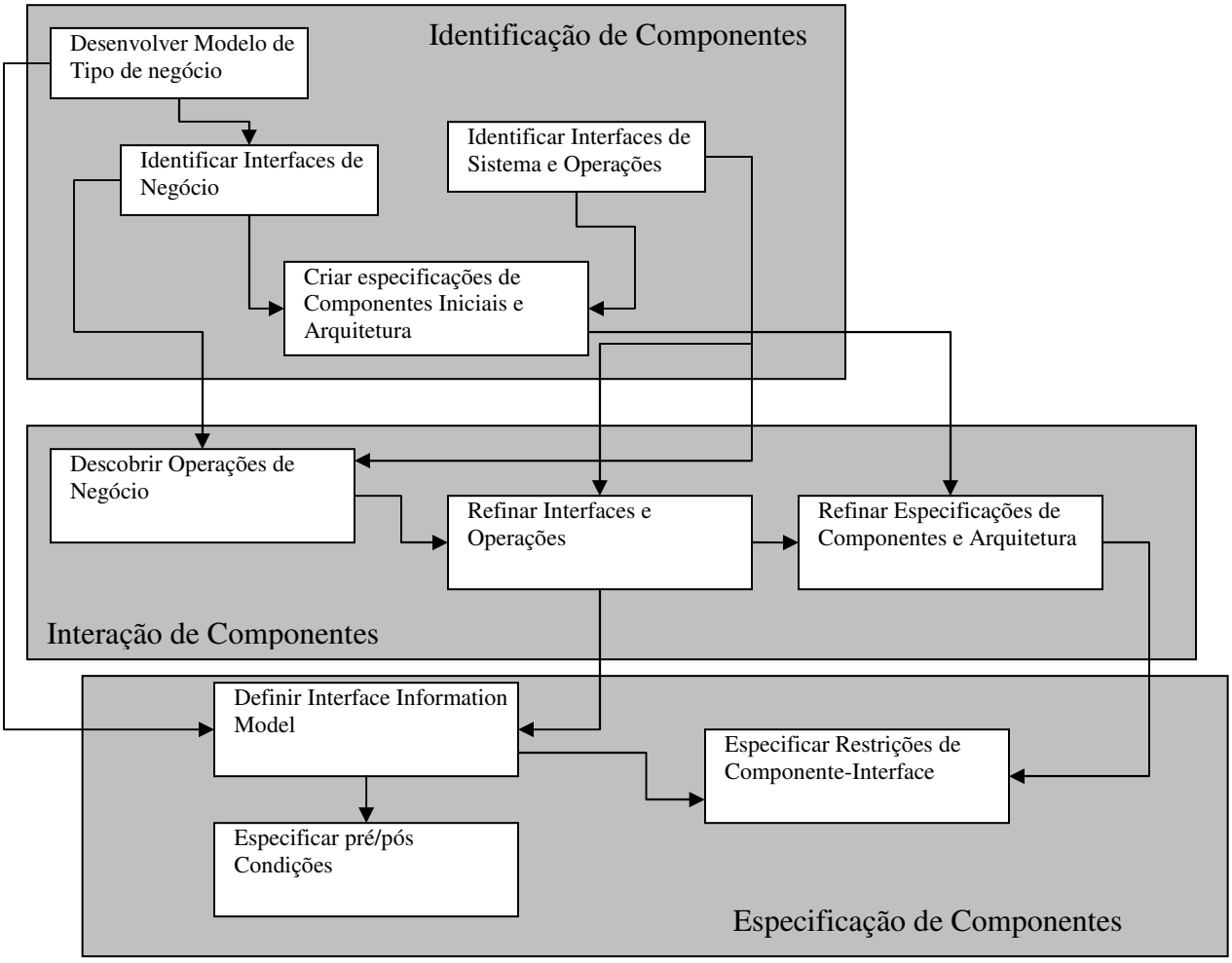

Figura 3 – Sub-fases da fase Especificação

- Especificação de componentes, em que é feito uma especificação detalhada das interfaces,incluindo suas restrições, em termos de pré e pós-condições. As seguintes atividades fazem parte desta sub-fase: definir Interface Information Model, especificar pré/pós condições e especificar restrições de componente-interface

2.2 Trabalhos Relacionados

Nesta seção, discutimos outros trabalhos também relacionados a repositório de componentes. Selecionamos nove repositórios que contém características importantes para um repositório de componentes e os comparamos com o repositório Rigel. A seguir faremos uma breve descrição dos repositórios selecionados.

2.2.1 AGORA

Agora [Seacord98] 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.

O Agora utiliza introspecção combinada com o mecanismo Web AltaVista Internet Service, para procurar, analisar e indexar componentes disponíveis na Internet. Os produtos de software encontrados são classificados por modelos de componentes.

A localização e indexação de componentes no Agora são feitas através de agentes de software, que vasculham a Internet para descobrir e coletar recursos.

A base de componentes do Agora é distribuída, ou seja os componentes são armazenados na Web, e não dentro do repositório. Ao realizar pesquisas, referências (URL) das localizações dos componentes na Internet são retornadas. O armazenamento de componentes é responsabilidade do produtor do componente, o Agora apenas mantem referências para os componentes.

2.2.2 RAGNAROK

SCM Ragnarok [Christensen99] 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 (*check-in*) e recuperação de artefatos (*check-out*) 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 termo **ramo** (*branch*) representa versões temporárias que não seguem a linha principal do desenvolvimento [Leon00]. Os ramos são utilizados para realizar o desenvolvimento paralelo de um mesmo sistema. No Ragnarok, atividades de GCS relacionadas com o desenvolvimento paralelo de componentes também têm suporte a arquitetura. Tanto as operações de criações de ramos, quanto a junção de ramos atualizam todos os componentes envolvidos de acordo com a arquitetura.

Outra característica importante do SCM Ragnarok é a possibilidade de rastrear a evolução da arquitetura do sistema.

Entretanto, 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.

2.2.3 WREN

WREN [Luer01] 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.

2.2.4 CODE BROKER

CodeBroker [Ye01] 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, por exemplo. 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.

2.2.5 OSCAR

OSCAR [Boldyreff02] 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 [Boldyreff03].

O OSCAR oferece suporte a controle de versões através da integração com o sistema GCS CVS.

Apesar de ser um repositório distribuído, os metadados do OSCAR não seguem um formato padronizado, que facilitaria uma integração com outros repositórios e ferramentas de desenvolvimento.

2.2.6 CKBR-P

CKBR-P [Vitharana03] é um protótipo de um repositório que aborda principalmente problemas relacionados à classificação e busca de componentes.

O repositório implementa um esquema proposto no artigo de Vitharana [Vitharana03], 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 **semi-estruturados** são dados com uma organização que pode mudar ou que apresenta uma irregularidade [Brian01].

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.

2.2.7 Odyssey-SCM

O Odyssey [Murta04] é uma ferramenta que fornece uma infra-estrutura para reutilização de software através de 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 [Clements01].

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 modificações 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.

O Odyssey Share é outro sub-projeto que trata a aplicação de trabalho colaborativo no processo de desenvolvimento de software.

O sub-projeto Odyssey-SCM tem um foco mais relacionado com gerência de configuração e mineração de rastros de modificações. Entretanto, o projeto do repositório Rigel se preocupa mais com a definição do metamodelo dos bens, modelando o componente, suas propriedades e seus relacionamentos.

2.2.8 FLASHLINE

Flashline 4 [Flashline] é um repositório de bens de software com uma boa capacidade de integração com outras ferramentas.

O repositório tem adaptadores para as principais ferramentas IDE do mercado. Vários sistemas GCS são apoiados. O sistema ainda oferece uma interface para fazer integrações adicionais como processos de desenvolvimento, outras ferramentas de desenvolvimento, etc.

É possível pesquisar bens de software por palavras-chave, categorias e classes de bens. Várias métricas de utilização do repositório estão disponíveis.

O modelo de metadados do repositório é flexível e pode ser estendido de acordo com as necessidades. O Flashline apóia o padrão RAS. Mas o modelo de dados do Flashline não tem conceitos importantes de componentes como a separação entre implementação e especificação.

2.2.9 LOGIC LIBRARY – LOGIDEX

LogicLibrary Logidex [Logidex] é um repositório com mecanismos de busca e mapeamentos de artefatos de software. O Logidex armazena artefatos relacionados a componentes, como por exemplo, executáveis, código fonte e documentação de desenvolvimento.

A pesquisa de artefatos pode ser realizada por palavras-chave ou por modelos. Na pesquisa por modelos, o usuário pode procurar artefatos relacionados com um determinado modelo de aplicação. Neste caso, o usuário seleciona um modelo (por exemplo: aplicação para processamento de pedidos), e logo em seguida seleciona um elemento do modelo (por exemplo: carrinho de compras). A ferramenta então procura todos os artefatos associados com o elemento selecionado.

A ferramenta também apóia associações e relacionamentos entre os artefatos cadastrados.

Integração com ferramentas de desenvolvimento (plataforma .NET) é provida através de um componente adicional.

Através do componente Anysource Asset Adapter, o Logidex permite uma integração com alguns sistemas GCSs, incluindo o Clearcase. O Logidex é compatível com o formato RAS [RAS04] de especificação de artefatos. O Anysource Asset Adapter ainda permite alguma adaptação para processos de desenvolvimento.

Apesar do suporte ao formato RAS, o modelo de metadados do Logidex não apresenta alguns conceitos de DBC, como por exemplo, separação entre especificação e implementação, interfaces e configurações de arquitetura.

2.2.10 Quadro Comparativo

Nesta seção apresentamos um quadro comparativo (Tabela 1 e Tabela 2) dos repositórios mencionados anteriormente. Os parâmetros utilizados na comparação estão explicados a seguir:

Suporte a controle de versões - implementações de bens podem ter diferentes versões. Portanto, é importante que o repositório tenha um sistema de controle de versões, a fim de distinguir diferentes distribuições dos mesmos bens [Schuenck05].

Suporte a processo de desenvolvimento de componentes - durante o desenvolvimento de um componente, diversos artefatos são produzidos. Assim, é possível que um repositório dê suporte não apenas ao componente acabado, mas também aos demais artefatos originados ao longo de sua criação [Schuenck05].

Representação de consultas - há diferentes formas de se representar consultas para busca de componentes. Algumas formas são por palavras-chave, por especificação de assinaturas e por especificação formal [Schuenck05].

Pesquisa - suporta pesquisa de bens.

Tipos de bens - repositórios podem manipular artefatos de software de vários tipos, dentre os quais pode-se destacar: componente compilado, código fonte, casos de uso, diagramas de classes, documentação de uso e planos de testes.

Formato padrão para metadados - Utilização de um formato padrão para o armazenamento de metadados, de modo a facilitar a interoperabilidade.

Metadados apóiam conceitos de DBC - metadados apóiam conceitos relativos a DBC, como por exemplo, separação entre especificação e implementação, interface requerida, interface provida.

Metadados apóiam conceitos de arquitetura - metadados apóiam conceitos de arquitetura de software, como por exemplo, configurações arquiteturais.

	RIGEL	WREN	Odyssey-SCM	RAGNAROK	CKBR-P	FLASHLINE

Suporte a controle de versões	Sim (CVS)	Sim	Sim.	Sim.	Não	Sim
Suporte a processo de desenvolvimento de componentes	Sim	-	-	-	-	Sim
Representação de consultas	Palavras-chave	Palavras-chave	-	-	Palavras-chave	Palavras-chave
Pesquisa	Sim	Sim	-	-	Sim	Sim
Tipos de bens	Artefatos de software genéricos	Código fonte	Artefatos de software genéricos	Artefatos de software genéricos	Artefatos de software genéricos	Artefatos de software genéricos
Formato padrão para metadados	RAS	-	-	-	-	RAS
Metadados apóiam conceitos de DBC	Sim.	-	Sim.	Sim.	Sim.	-
Metadados apóiam conceitos de arquitetura	Sim.	-	Sim.	Sim.	-	-

Tabela 1 – Quadro comparativo de repositorios I

	RIGEL	LOGIC LIBRARY – LOGIDEX	AGORA	CODE BROKER	OSCAR
Suporte a controle de versões	Sim (CVS)	Sim	-	-	Sim
Suporte a processo de desenvolvimento de componentes	Sim	Sim	Parcial	-	Sim
Representação de consultas	Palavras-chave	Palavras-chave	Palavras-chave	Linguagem natural e especificação de assinaturas	Palavras-chave e amostra de comportamento
Pesquisa	Sim	Sim	Sim	Sim	Sim
Tipos de bens	Artefatos de software genéricos	Artefatos de software genéricos	Componente compilado	Componente compilado	Artefatos de software genéricos
Formato padrão para metadados	RAS	RAS	-	-	-
Metadados apóiam conceitos de DBC	Sim.	-	Sim.	Parcial	Parcial
Metadados apóiam conceitos de arquitetura	Sim.	-	-		-

Tabela 2 - Quadro comparativo de repositórios II

Como visto na seção 1.2, o Rigel é um repositório que reúne características importantes para DBC:

- Suporte a processo de desenvolvimento de componentes.
- Pesquisa em bens por palavras-chave.

- Formato padrão para metadados.
- Metadados apóiam conceitos de DBC.
- Metadados apóiam conceitos de arquitetura.
- Suporte a controle de versões.

Baseado nos quadros acima (Tabela 1 e Tabela 2) podemos perceber a falta de um outro repositório que reúna todas estas características.

2.3 Arquitetura de Software

A arquitetura de software, através de um alto nível de abstração, define o sistema em termos de seus componentes arquiteturais; a interação entre essas entidades e os atributos e funcionalidades de cada um. Componentes Arquiteturais representam unidades abstratas do sistema [Sommerville01]. Por conhecerem o fluxo interativo entre os componentes do sistema, é possível estabelecer nos conectores protocolos de comunicação e coordenar a execução dos serviços que envolvam mais de um componente do sistema.

Essa visão estrutural do sistema em um alto nível de abstração proporciona benefícios importantes, que são imprescindíveis para o desenvolvimento de sistemas de software complexos. Os principais benefícios são: (i) a organização do sistema como uma composição de componentes lógicos; (ii) a antecipação da definição das estruturas de controle globais; (iii) a definição da forma de comunicação e composição dos elementos do projeto; e (iv) o auxílio na definição das funcionalidades de cada componente projetado. Além disso, uma propriedade arquitetural representa uma decisão de projeto relacionada a algum requisito não-funcional do sistema, que quantifica determinados aspectos do seu comportamento, como confiabilidade, reusabilidade e modificabilidade [Clements03, Sommerville01].

Uma determinada propriedade arquitetural pode ser obtida através da utilização de estilos arquiteturais que possam garantir a preservação dessa propriedade durante o desenvolvimento do sistema [Shaw+96, Monroe97]. Um estilo arquitetural caracteriza uma família de sistemas que são relacionados pelo compartilhamento de suas propriedades estruturais e semânticas. Esses

estilos definem inclusive restrições de comunicação entre os componentes do sistema. A maneira como os componentes de um sistema ficam dispostos é conhecida como configuração.

As propriedades arquiteturais são derivadas dos requisitos do sistema e influenciam, direcionam e restringem todas as fases do seu ciclo de vida. Sendo assim, a arquitetura de software é um artefato essencial no processo de desenvolvimento de softwares modernos, sendo útil em todas as suas fases [Chen02]. A importância da arquitetura fica ainda mais clara no contexto do desenvolvimento baseados em componentes. Isso acontece uma vez que na composição de sistemas, os componentes precisam interagir entre si para oferecer as funcionalidades desejadas. Além disso, devido à diferença de abstração entre a arquitetura e a implementação de um sistema, um processo de desenvolvimento baseado em componentes deve apresentar uma distinção clara entre os conceitos de componente arquitetural, que é abstrato e não é necessariamente instanciável; e componente de implementação, que representa a materialização de uma especificação em alguma tecnologia específica e deve necessariamente ser instanciável.

2.3.1 Um modelo Conceitual para DBC e Arquitetura de Software

Nesta seção apresentamos trechos do modelo conceitual para DBC e Arquitetura de Software proposto no trabalho de Guerra [Guerra05]. Este modelo foi a base para o projeto do modelo de metadados do Rigel.

Especificação de Componente

O modelo da especificação de componente pode ser visto na Figura 4. A seguir descrevemos os elementos identificados no modelo:

Propriedade Externa (*External Property*) é uma característica de uma especificação de componente de software que representa um de seus pontos de conexão.

Interface Provida (*Provided Interface*) é uma propriedade externa associada a um conjunto de serviços providos pelo componente de software. Esses serviços são especificados pelo tipo da interface provida. Uma interface provida pode ser utilizada como um ponto de

conexão de um elemento arquitetural, fornecendo um meio de interação entre um componente e o ambiente onde está inserido.

Dependência de Contexto (*Context Dependency*) é uma propriedade externa associada a um conjunto de serviços que o componente de software requer do seu ambiente. As dependências de contexto incluem os serviços a serem providos por outros componentes ou pela infra-estrutura do sistema.

Interface Requerida (*Required Interface*) é um tipo de dependência de contexto associada a serviços a serem providos por outros componentes de software. Esses serviços são especificados pelo tipo da interface requerida. Assim como a interface provida, a interface requerida também pode ser utilizada como um ponto de conexão de um elemento arquitetural.

Tipo de Interface (*Interface Type*) define um conjunto de serviços e especifica o comportamento esperado desses serviços [Luckham00].

Porta (*Port*) é uma propriedade externa associada a um comportamento do componente de software que é encapsulado por um conjunto de interfaces providas ou requeridas. Uma porta também pode ser utilizada como um ponto de conexão de um elemento arquitetural.

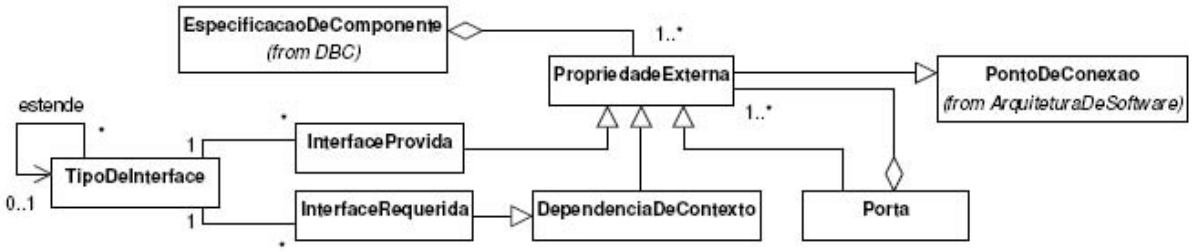

Figura 4 – modelo de especificação de componente.

Implementação de Componente

O modelo da implementação de componente pode ser visto na Figura 5. A seguir descrevemos os elementos identificados no modelo:

Arquitetura de Componentes (*Component Architecture*) descreve um conjunto de componentes de software, seus relacionamentos e suas dependências de comportamento [Cheesman+00].

Sub-Componente (*SubComponent*) é um componente interno a outro componente de software que é descrito por uma especificação.

Conexão de Interface (*Interface Connection*) é uma conexão entre uma interface requerida de um componente e uma interface provida de outro componente, representando uma possível interação uni-direcional, onde o segundo componente provê serviços requeridos pelo primeiro. Uma conexão de interface pode representar uma conexão entre dois elementos arquiteturais.

Conexão de Serviço (*Service Connection*) é uma conexão entre duas portas simétricas de dois componentes, representando uma possível interação bi-direcional entre esses componentes. Duas portas são ditas simétricas quando para toda interface provida através dessas portas haja uma correspondência com uma interface requerida de mesmo tipo, associada à outra porta. Assim como uma conexão de interface, uma conexão de serviço também pode representar uma conexão entre dois elementos arquiteturais.

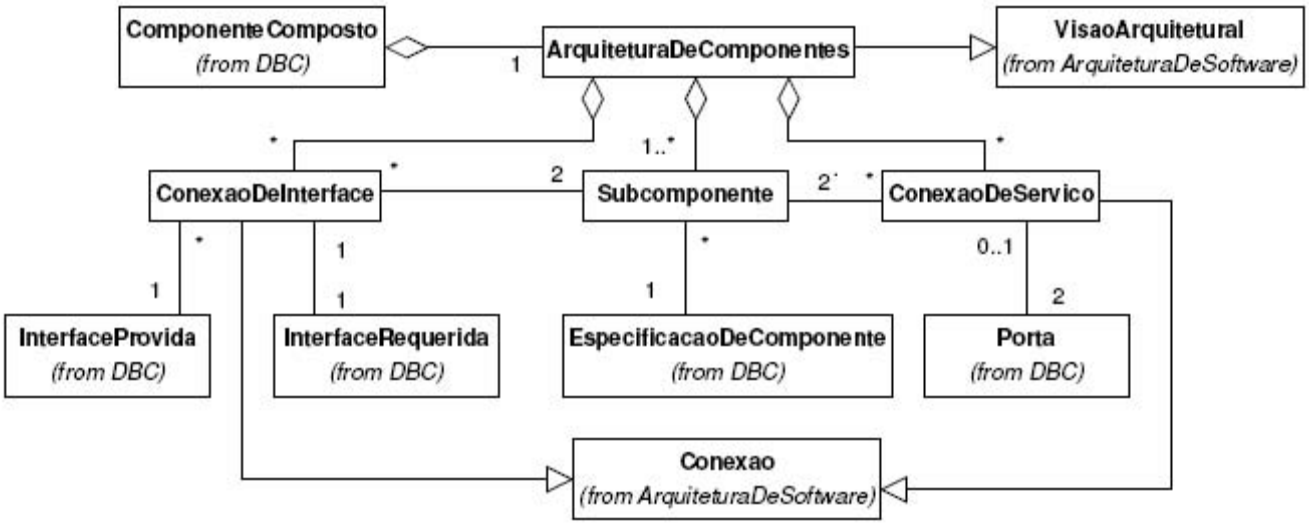

Figura 5 – modelo de implementação de componente

Arquitetura de Software

O modelo da arquitetura de software pode ser visto na Figura 6. A seguir descrevemos os elementos identificados no modelo:

Configuração de Componente (*Componente Configuration*) é um conjunto de instâncias concretas ou abstratas em conformidade com a arquitetura de um componente composto instanciado pela configuração.

Instância Concreta (*Concrete Instance*) é um sistema concreto ou um componente elementar.

Sistema Abstrato (*Abstract System*) é uma configuração de componente que concretiza apenas parte dos sub-componentes do componente composto instanciado pela configuração. Um sub-componente não concretizado por uma configuração é uma instância abstrata da configuração.

Sistema Concreto (*Concrete System*) é uma configuração de componente que não possui instâncias abstratas, ou seja: concretiza todos os sub-componentes do componente composto instanciado pela configuração.

Família de Sistemas (*System Family*) é um conjunto de configurações de componente relacionadas através de associações de herança simples. Sistemas de uma mesma família instanciam um mesmo componente composto. As instâncias concretas definidas pela configuração filha substituem instâncias abstratas ou concretas da configuração mãe. Uma configuração filha herda as instâncias concretas da configuração mãe que não são substituídas pela configuração filha.

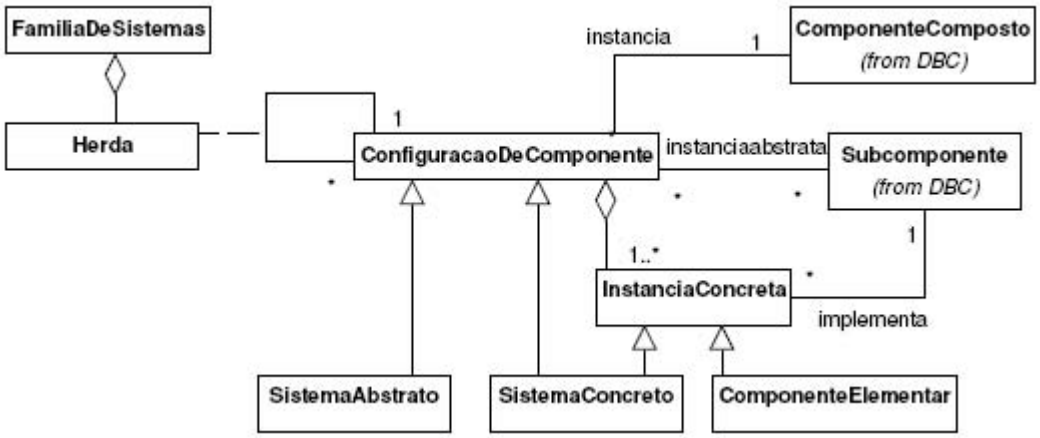

Figura 6 – modelo de configuração de software.

2.4 Gerência de Configuração de Software

Gerência de Configuração de Software (GCS, e em inglês SCM – Software Configuration Management) é o controle da evolução de sistemas complexos. GCS é a área que nos permite manter sob controle os produtos de software em evolução [Estublier00].

Durante o final dos anos 70 e início dos anos 80, GCS surgiu como uma tentativa de tratar questões como arquitetura, evolução e construção (*building*). Construção é o processo de criação do código binário para uma determinada versão do software.

Atualmente, um sistema típico de GCS deve prover serviços nas seguintes áreas [Estublier00] :

- Gerenciar repositório de componentes – existe uma necessidade em armazenar de forma segura os diferentes componentes de um produto de software e as suas versões. Este tópico inclui gerência de versões e gerência de objetos complexos.
- Ajudar engenheiros em suas atividades usuais – Engenharia de software envolve aplicar ferramentas em artefatos. Produtos de GCS tentam prover as ferramentas corretas aos engenheiros para construção de sistemas e controle de versão de artefatos. Isto é geralmente referenciado como controle da área de trabalho (*workspace control*). São questões importantes a compilação e controle dos artefatos gerados pela compilação.
- Controle e suporte a processos – no final dos anos 80, ficou claro que um grande problema em GCS era relacionado com os usuários. Tradicionalmente, controle de mudanças (*change control*) já é parte de sistemas GCS. O controle de mudanças é capacidade de agrupar alterações de artefatos, ou versões de artefatos, relacionando este grupo a uma determinada atividade no processo de desenvolvimento de software. Atualmente a tendência é estender ainda mais o suporte a processos.

2.4.1 CVS

O CVS (Concurrent Versions Systems) [CVS] é uma ferramenta GCS derivada do RCS (Revision Control System). O CVS que é um software livre mantido pela Free Software Foundation.

Algumas características diferenciais do CVS são:

- O CVS trabalha com o modelo cliente/servidor; isto permite que o documento seja acessado de qualquer local, bastando ter uma conexão com a Internet, ou no caso de um CVS interno, bastando a máquina estar conectada na rede interna
- Disponível para vários sistemas operacionais, incluindo Linux, Solaris, OS/2 e Windows;
- Permite que qualquer formato de documento seja incluído, desde arquivos com código, documentos legais e até imagens.
- A ferramenta possui uma interface web, permitindo que arquivos e módulos possam ser vistos através de um navegador.

O CVS armazena as versões dos arquivos controlados por ele no sistema de arquivos. Uma variável chamada CVSROOT define o local onde estas versões de arquivo são armazenadas.

2.5 Implementação de Repositórios

Nesta seção discutiremos os repositórios e tecnologias que podem ser utilizadas em sua construção. Na seção 2.5.1 apresentamos os principais conceitos sobre repositórios. Já nas seções 2.5.2 e 2.5.3 apresentamos o XML e o XML Schema respectivamente, duas tecnologias relacionadas ao armazenamento de dados. Na seção 2.5.4, apresentamos detalhes sobre o padrão RAS. Finalmente na seção 2.5.5, mencionaremos alguns conceitos sobre máquinas de busca.

2.5.1 Repositórios

Repositório de bens de software reutilizáveis é um ponto centralizado de acesso e armazenamento de bens reutilizáveis. Bem é uma solução para um problema de desenvolvimento de software. Bem reutilizável é uma solução para um problema recorrente. Artefato é um elemento lógico ou físico de um bem. Um elemento lógico contém ao menos um artefato físico. Artefatos físicos correspondem a arquivos no sistema de arquivos [RAS04].

Mais especificamente temos o repositório de componentes - 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 o ciclo de vida dos componentes. Pesquisas avançadas e navegação em informações sobre componentes são funcionalidades utilizadas por repositórios para apoiar o reuso com componentes em um processo de DBC [Luqi99] [Seacord99] [Mili98].

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 [Schuenck05] [PrietoDiaz87] [Ye01].

Repositórios podem apoiar um Processo de Desenvolvimento dos Componentes. Durante o desenvolvimento de um componente, diversos artefatos são produzidos. Um repositório apóia um processo DBC se este oferece suporte não apenas ao componente acabado, mas também aos demais artefatos originados ao longo de sua criação.

Metadados são dados capazes de descrever outros dados, ou seja, dizer do que se tratam, dar um significado real e plausível a um arquivo de dados, eles são a representação de um objeto digital. Mais sinteticamente, podemos dizer que um metadado é um dado utilizado para descrever um dado primário [RAS04] [Wikipedia]. Para repositórios, metadados são informações sobre o bem. O nome, a descrição textual, a lista de artefatos e a classificação de um bem são exemplos de metadados de um bem.

2.5.2 XML

Extended Markup Language (linguagem extendida de marcação), abreviado XML, descreve objetos de dados chamados documentos XML e parcialmente descreve o comportamento 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. Pela construção, os documentos XML estão em conformidade com os documentos SGML [W3XML].

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 destes formam os dados em caracteres (character data), e alguns destes formam a marcação (*markup*). A marcação codifica uma descrição do leiaute do armazenamento e estrutura lógica. O XML provê mecanismos para impor restrições no leiaute de armazenamento e na estrutura lógica [W3XML]. *Tags* são estruturas de marcação que consistem em breves instruções, tendo uma marca de início e outra de fim.

O exemplo da Figura 7, mostra um trecho em XML contendo metadados de um bem.

```
<?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 7 – Exemplo de XML para metadados de bens.

Um processador XML é um módulo de software utilizado para ler documentos XML e provê acesso a seu conteúdo e estrutura [W3XML].

2.5.3 XML Schema

XML Schema é uma linguagem para a descrição da estrutura e das restrições relativas ao conteúdo de documentos XML.

XML Schema consiste de componentes tais como definições de tipos e declarações de elementos. Estes podem ser usados para avaliar a validade de items de informações de elementos

e atributos, e ainda mais podem especificar adições para estes itens e seus descendentes[W3Schema].

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.

XML Schema é uma alternativa ao DTD (Document Type Definition) baseada em XML. O DTD foi o primeiro padrão para descrição de documentos XML. Entretanto, o DTD não possui algumas características importantes como "espaços de nomes". Na realidade XML Schema é visto como um sucessor do DTD.

Inicialmente o XML Schema foi proposto pela Microsoft, mas tornou-se uma recomendação oficial do W3 em maio de 2001 [W3Schema].

2.5.4 RAS

O RAS é uma especificação proposta pela Rational e aceita pela OMG [RAS04] como padrão para especificação de bens de software reutilizáveis. É uma representação totalmente baseada em meta modelos, onde existem definições sobre os metadados que um bem deve possuir. O principal objetivo do RAS é especificar a quantidade mínima de meta informações que um bem reutilizável (*Reusable Asset*) deve ter.

Podemos citar também alguns outros objetivos do RAS, como estimular o desenvolvimento baseado na utilização de componentes e aumentar a comunicação entre os produtores e os consumidores de componentes. O modelo de um bem é estruturado em seções, como mostrado na Figura 8. Estas seções têm os seguintes nomes: *Classification* (Classificação), *Solution* (Solução), *Usage* (Utilização) e *Related Asset* (Bem Relacionado).

A seção *Classification* contém uma lista de descritores do bem, assim como descrições dos contextos nos quais o bem é relevante. A seção *Solution* descreve os artefatos e detalhes de projeto do bem. *Usage* contém regras para instalação, adaptação e uso do bem. *RelatedAsset* contém os relacionamentos com outros bens.

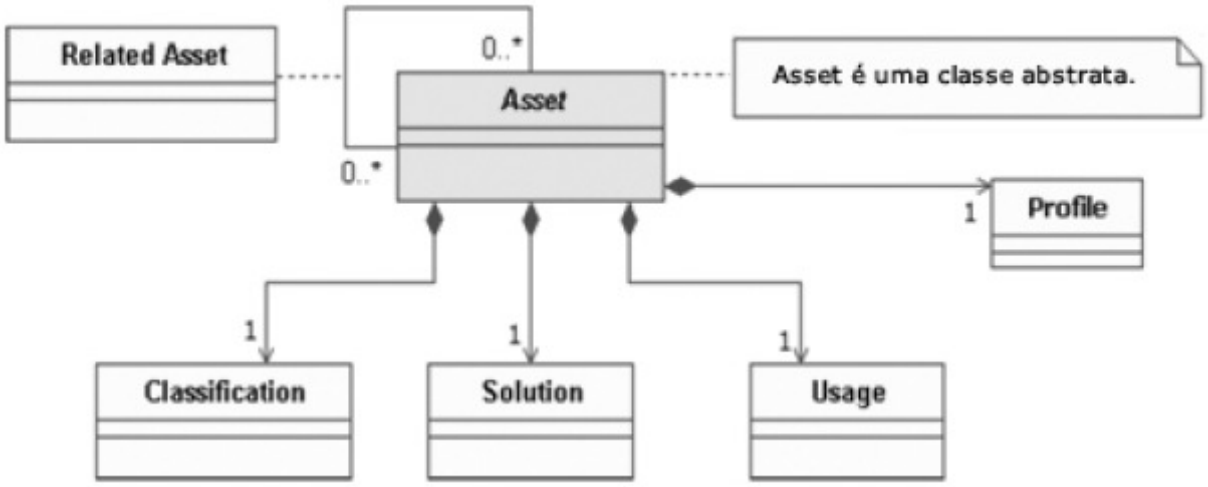

Figura 8 – principais domínios do RAS.

O bem pode estar relacionado a um determinado *profile*, que pode conter meta informações específicas para determinados tipos de bem, como Web Services, patterns e frameworks. O *profile* básico é o Core RAS, que contém propriedades que se aplicam a todos os bens de software em geral. O Core RAS pode ser estendido através da criação de novos *profiles*.

O RAS ainda especifica o formato de armazenamento de um bem é armazenado. Os metadatados são armazenados em XML, seguindo uma especificação em XML Schema. Cada *profile* possui uma especificação XML Schema associada contendo detalhes do modelo de metadados relativo ao profile. O XML Schema define o nome dos elementos XML, o tipo dos atributos e elementos XML, relacionamentos entre elementos XML e a cardinalidade destes relacionamentos. Os artefatos podem estar armazenados nos sistema de arquivos ou empacotados em um arquivo comprimido (compressão Zip), em ambos os casos os metadados devem ser incluídos no empacotamento.

.

2.5.5 Máquinas de Busca

Máquina de Busca (*search engine*) é um programa projetado para auxiliar a encontrar informações armazenadas em um sistema de computação, por exemplo a World Wide Web, ou em um computador pessoal. Uma máquina de busca permite que se pesquise um conteúdo que satisfaz um critério específico. Estes critérios geralmente contêm uma palavra ou frase, e o resultado desta pesquisa é uma lista de referências que satisfazem este critério [Hatcher04].

Para operar com rapidez e eficiência, máquinas de busca utilizam índices atualizados regularmente. Sem um qualificação adicional, o termo "*search engine*" usualmente se refere a máquinas de busca específicas para a Web, as quais procuram por informações na Web pública. Outros tipos máquinas de busca são as máquinas de busca corporativas, para pesquisar em Intranets, e máquinas de busca pessoais, para pesquisa em computadores pessoais.

Máquinas de busca podem ser utilizadas para pesquisar informações em repositórios. O fato de muitas vezes termos metadados com uma estrutura variável, ou seja metadados semi-estruturados, dificulta o armazenamento e pesquisa destes dados utilizando banco de dados relacionais e favorece a utilização de máquinas de busca.

2.6 Resumo

Neste capítulo apresentamos vários conceitos utilizados no restante deste documento. As seções 2.1 e 2.3, apresentam vários conceitos de DBC e arquitetura de software, estes conceitos guiam o projeto e a implementação do repositório Rigel. O processo UML Components, apresentado na seção 2.1.2, é especialmente importante no Capítulo 6, no qual descrevemos os estudos de caso deste trabalho. A seção 2.2 faz uma comparação entre vários repositórios, ressaltando o diferencial do Rigel.

As seções 2.4 e 2.5 mencionam maiores detalhes sobre tecnologias relacionadas a repositórios. Estas tecnologias são utilizadas no projeto e na implementação do repositório Rigel.

Capítulo 3 Requisitos do Repositório Rigel

Neste capítulo descrevemos os requisitos considerados na construção do repositório Rigel. Dividimos estes requisitos em dois grupos: requisitos funcionais e requisitos de qualidade.

3.1 Requisitos Funcionais

Alguns requisitos funcionais são considerados básicos para a construção de repositórios, tais como pesquisar, armazenar e recuperar bens. Estes requisitos foram considerados no projeto, mas tentamos manter um foco em componentes e desenvolvimento baseado em componentes durante a definição dos requisitos. Podemos agrupar os requisitos funcionais em 5 grupos:

1. Editar bens e armazenar artefatos
2. Recuperar bens e artefatos
3. Pesquisar bens
4. Apoio ao controle de versão dos artefatos
5. Apoio a processos de Desenvolvimento Baseado em Componentes. No caso deste trabalho utilizamos o UML Components.

3.1.1 Editar bens e armazenar artefatos

Editar bens e armazenar artefatos é um requisito essencial a ser implementado por um repositório. Os dados a serem armazenados estão divididos em metadados do bem e artefatos.

Os metadados são informações sobre o bem cadastrado no repositório. O uso desta informação é diverso, mas, principalmente, estes dados apóiam as seguintes funcionalidades: pesquisa de bens, descrição de bens, navegação entre bens relacionados.

A pesquisa de bens é um dos processos que mais utilizam as informações contidas nos metadados. Descrições sobre componentes, propriedades que classificam um bem e até mesmo relacionamentos entre bens devem estar disponíveis para processos de pesquisa.

O outro tipo de informação armazenada no repositório são os artefatos. Artefatos são arquivos que contêm partes dos bens armazenados.

Ainda é importante que exista um suporte a um modelo de metadados que considere características de componentes tais como: interfaces e separação entre especificação e implementação de componentes. A granularidade é outro fator importante em relação ao reuso de software. Partes de componentes como componentes requeridos ou interfaces podem ser mais facilmente reutilizadas, se estas estiverem devidamente armazenadas. Deve ser possível pesquisar, identificar e recuperar isoladamente estas partes de componentes para apoiar o reuso com menor granularidade. Destacamos que os metadados deverão apoiar os seguintes tipos de bens: componente abstrato, componente concreto, interface e configuração arquitetural de um componente.

A seguir temos um exemplo de um caso de uso relacionado ao requisito de editar bens e armazenar artefatos.

1. **Adicionar Bem**

   Objetivo: adicionar um bem ao repositório.

   Pré-condição: o bem não existe no repositório.

   Pós-condição: o bem está armazenado no repositório

   1) O usuário seleciona a opção de Adicionar bem
   2) O repositório abre um tela de cadastro do bem
   3) O usuário cadastra os dados do bem
   4) O usuário confirma os dados
   5) O repositório armazena o bem.

O Apêndice A contém a lista completa dos casos de uso do repositório Rigel.

3.1.2 Recuperar bens e artefatos

O usuário deve poder selecionar um bem no repositório e em seguida, executar a recuperação do bem. O repositório deverá empacotar o bem juntamente com os seus artefatos e

disponibilizá-lo para transferência. O bem deverá ser empacotado de acordo com o formato padronizado pelo repositório.

3.1.3 Pesquisar bens

A pesquisa de bens tem como requisito os seguintes itens: precisão, recuperação, desempenho, interoperabilidade e atributos disponíveis para pesquisa.

Precisão ilustra qual a proporção entre os bens considerados relevantes e aqueles efetivamente recuperados [KENT55]. Alta precisão indica que a maior parte dos bens retornados é relevante. O repositório Rigel deve conter mecanismos que possibilitem alta precisão em suas pesquisas.

Recuperação (*recall*) de resultados em pesquisas é o valor que indica quantos bens foram recuperados dentre o total considerado relevante na coleção pesquisada [KENT55]. Alta recuperação indica que poucos bens relevantes foram ignorados no resultado da pesquisa. O Rigel deverá apoiar meios para melhorar a recuperação de resultados em pesquisas de bens.

Os atributos disponíveis para pesquisa são todas as informações sobre um bem que estão disponíveis para os mecanismos de buscas. Quanto mais propriedades de um componente puderem ser indexadas, maior a flexibilidade e eficiência do sistema de buscas. O Rigel deverá possibilitar pesquisas por quaisquer propriedades dos meta-dados de um bem. A pesquisa de atributos que materializam os relacionamentos entre bens deverá ser implementada a fim de dar suporte a navegação entre bens.

Interoperabilidade é a capacidade de integração com outros sistemas. O Rigel deverá ter uma interface baseada em padrões, a fim de facilitar a integração com outros sistemas. Esta padronização deverá ser observada nos protocolos de comunicação e também no formato dos dados retornados pelo repositório.

Desempenho é geralmente medido através do tempo de resposta de uma pesquisa. [Lucredio04] O Rigel deverá utilizar mecanismos que permitam um tempo de resposta compatível com os tempos apresentados pelos repositórios similares.

Neste trabalho, limitamos a pesquisa aos metadados de bens e artefatos. A pesquisa em conteúdo de artefatos não foi abordada neste projeto.

3.1.4 Apoio ao controle de versão dos artefatos

Cada evolução de um bem deve ser armazenada em um repositório, de modo a permitir o rastreamento das diversas versões dos componentes. É necessária uma integração com um sistema GCS para ter um controle adequado de todas as versões da arquitetura e dos componentes.

3.1.5 Apoio a processos de Desenvolvimento Baseado em Componentes

O Rigel deverá oferecer suporte as principais operações requeridas por processos DBC: adição, modificação e remoção de componentes, recuperação de componentes, pesquisas em componentes. O modelo de metadados deverá refletir os principais conceitos de componentes, conforme discutido nas seções 2.3.1 e 3.1.1.

3.2 Requisitos de Qualidade

A seguir apresentamos os requisitos de qualidades abordados no projeto do repositório Rigel.

3.2.1 Interoperabilidade

Interoperabilidade é a capacidade de interagir com outros sistemas de acordo com o que foi especificado [ISO2005]. Repositórios geralmente se integram com outros repositórios ou com ferramentas de desenvolvimento de software.

A integração entre repositórios na maior parte das vezes tem como objetivo o compartilhamento do conjunto de bens e o suporte a pesquisas mais abrangentes uma vez que existem mais bens a serem pesquisados. A integração com outros repositórios visa o compartilhamento de bens e facilidades de pesquisa.

A integração com ferramentas de desenvolvimento tem principalmente um foco na execução de operações do repositório a partir de ambientes de desenvolvimento integrados (IDE).

As funcionalidades tais como pesquisa, recuperação, adição e alterações de bens devem poder ser acionadas a partir de IDEs.

Em ambos os casos, é importante ter um formato de dados aberto, que facilite o manuseio dos bens recuperados em diferentes repositórios. Um formato de dados aberto tem vários pontos positivos:

- Máquinas de Buscas (*search engines*) podem indexar metadados de bens, disponibilizando-os para buscas.
- Ferramentas IDEs podem automatizar tarefas de recuperação e adição de bens de um repositório.
- Compartilhamento de bens por outros repositórios. Os bens podem ser transferidos entre repositórios, sem conversões de dados, se ambos os repositório envolvidos na transferência oferecerem suporte ao formato de dados do bem.

3.2.2 Manutenibilidade

Manutenibilidade é a facilidade para fazer alterações [ISO2005]. Evolução de sistemas e correção de defeitos estão entre as principais razões para alterar um sistema existente. No caso do repositório Rigel, algumas alterações podem ser razoavelmente freqüentes, tais como: suporte a novos formatos de dados e integração com outros sistemas.

O Rigel deverá ter um projeto que facilite a manutenibilidade do sistema, especialmente se estas alterações estiverem relacionadas com integrações com outros sistemas.

3.2.3 Conformidade

Conformidade é a observância a padrões, convenções ou regras estabelecidas [ISO2005]. Em relação ao Rigel, conformidade está relacionada à interoperabilidade. Este repositório deverá seguir padrões que facilite a integração com outros sistemas.

Um formato de dados padronizado para empacotamento de bens é essencial para promover interoperabilidade entre repositórios.

A interface do Rigel deverá facilitar a integração com outros sistemas através da utilização de protocolos de comunicação padronizados e abertos.

3.2.4 Segurança

Segurança contra intrusão é a capacidade de evitar acesso não-autorizado, seja ele acidental ou deliberado, a programas e dados [ISO2005]. O acesso à informações confidenciais pode afetar a confiabilidade do sistema. Não iremos abordar este requisito neste projeto.

3.2.5 Desempenho

Em relação a pesquisas, desempenho é geralmente medido através do tempo de resposta de uma pesquisa [Lucredio04]. O Rigel deverá utilizar mecanismos que permitam um tempo de resposta compatível com outros repositórios existentes no mercado, principalmente para as pesquisas mais freqüentes do DBC.

As operações de cadastro de bens também devem apresentar métricas de desempenho aceitáveis, tendo em vista a freqüência com que estas são executadas.

3.2.6 Portabilidade

Portabilidade é o conjunto de atributos que permitem que um software seja transferido de um ambiente para outro [ISO2005]. Em relação à repositório portabilidade é mais relevante para a interface com o usuário. Especificamente temos as seguintes sub-características de portabilidade como importantes para interfaces com usuários de repositório:

- Adaptabilidade – deve ser possível acessar o repositório através da execução de sua interface com usuário em várias platarformas.
- Capacidade de Instalação – deve ser fácil instalar a interface com usuário do repositório.

3.3 Resumo

Este capítulo apresentou os requisitos do repositório Rigel. Inicialmente foram mostrados os requisitos funcionais, divididos nos seguintes itens: editar bens e armazenar artefatos, recuperar bens e artefatos, pesquisar bens, apoio ao controle de versão dos artefatos, apoio a processos de DBC. Os requisitos de qualidade mais relevantes para o repositório Rigel foram levantados e descritos neste capítulo.

Capítulo 4 Projeto do Repositório Rigel

Neste capítulo, apresentamos detalhes sobre o projeto do repositório Rigel. Inicialmente descrevemos a arquitetura do sistema, com ênfase em metadados, interoperabilidade, integração com sistemas GCS.

Posteriormente, comentamos o projeto de três outras funcionalidades importantes: pesquisa, armazenamento e recuperação de bens.

4.1 Arquitetura de Componentes do Repositório Rigel

O processo utilizado no desenvolvimento do repositório Rigel foi o UML Components [Cheesman+00], descrito na seção 2.1.2. A arquitetura geral do repositório de componentes usa o padrão n-tier, de acordo com o UML Components (Figura 9). Em uma visão de alto nível, podemos entender a arquitetura do Rigel como sendo composta principalmente de quatro camadas: uma camada de interface e diálogo com o usuário, uma camada de sistema, uma camada de negócios e uma camada de infra-estrutura.

A camada de Interface e Diálogo é responsável por apresentar informações ao usuário, capturar as entradas de dados e gerenciar os diálogos com o usuário em uma sessão.

A camada de sistema é a representação externa do sistema e a suas interfaces provêem acesso às funcionalidades do sistema. A camada de negócio implementa regras e informações de negócio. A camada de infra-estrutura é responsável por apoiar a implementação da camada de negócio.

Um componente CVS representa a integração do Rigel com o sistema GCS CVS. Os metadados deste repositório são armazenados no sistema de arquivos do sistema operacional, por isso representamos o sistema de arquivos como um componente.

Na camada de sistema, temos os serviços básicos do repositório descritos no Capítulo 3, que são oferecidos pelas operações das seguintes interfaces: IAssetEdit é responsável pelas operações de adição (caso de uso 1, seção 3.1.1), remoção (caso de uso 2, seção A.13.1.1) e

alteração de bens (caso de uso 3, seção A.13.1.1) ; IArtifactEdit realiza as mesmas operações em artefatos (casos de uso 4, 5 e 6, seção A.1); ISearchAsset oferece serviço de pesquisa por bens (casos de uso 1 e 2, seção A.3); A interface ISearchAsset permite o acesso remoto ao repositório para pesquisa e recuperação de componentes. As funcionalidades de recuperação de componentes são oferecidas pela interface IRetrieve (caso de uso 1, seção A.2).

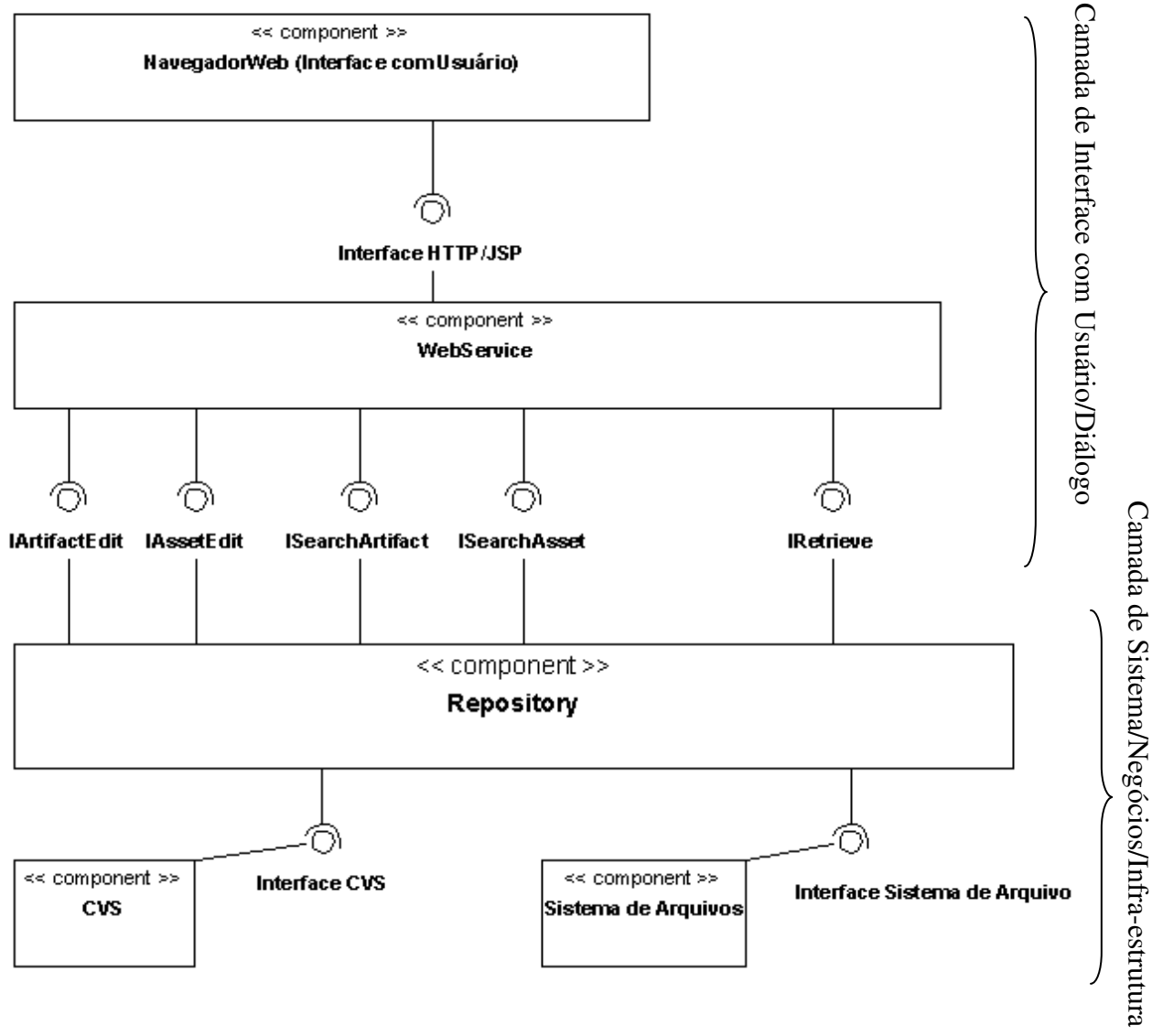

Figura 9 - Arquitetura Geral do Repositório de Componentes

4.1.1 Camada de Interface e Diálogo com o Usuário

Escolhemos uma interface Web como implementação das camadas de interface e de diálogo com o usuário. A disponibilidade em múltiplas plataformas, facilidade de acesso e instalação e curva de aprendizado foram os fatores que mais influenciaram na escolha de uma interface Web. Esta escolha esta baseada nos requisitos apresentados na seção 3.2.6.

Entre as tecnologias disponíveis para implementar serviços Web, selecionamos JSP (Java Server Pages) para a implementação no Rigel. JSP é uma tecnologia para criar conteúdo dinâmico para a Web [JSP]. Escolhemos JSP devido à portabilidade e ao suporte a separação entre a programação lógica (parte dinâmica) da programação visual (parte estática). Sendo JSP baseado em Java, ele tem praticamente as mesmas possibilidades de portabilidade de aplicações Java, a plataforma utilizada na camada de sistema do Rigel. Assim, aumentamos a sinergia entre a camada de interface e a camada de sistema, uma vez que ambas têm implementações baseadas em Java e possuem portabilidade semelhante.

Futuramente, outras implementações de interfaces de usuário poderão ser adicionadas ao Rigel. O fato de termos uma camada de sistema bem separada da camada de interface e de diálogo com o usuário facilita o desenvolvimento de interface de usuário adicionais. Uma possível evolução seria a implementação de um *plug-in* para um IDE (Integrated Development Environment – Ambiente de Desenvolvimento Integrado), por exemplo, o Eclipse [Eclipse], que facilite o acesso ao repositório direto do IDE. Neste caso, toda a camada de sistema seria reutilizada, sendo necessário apenas implementar a camada de interface.

4.1.2 Camada de Sistema

Dividimos a camada de sistema em duas outras camadas: serviços de sistema e serviços de negócios (Figura 10). A seguir descrevemos detalhes sobre a camada de sistema. RetrieveAndEdit é um componente de sistema responsável pelas funcionalidades de recuperação de bens, adição, modificação e remoção de bens e artefatos. Todas as modificações no Rigel passam pelo RetrieveAndEdit. A implementação de grande parte das funcionalidades de

RetrieveAndEdit fica a cargo da camada de negócio, assim existe: (i) uma dependência entre RetrieveAndEdit e o AssetEdit, componente de camada de negócio relacionado a alterações em bens e artefatos; (ii) uma dependência entre RetrieveAndEdit e o AssetRetrieve, componente da camada de negócio que apóia operações de recuperação de bens e artefatos.

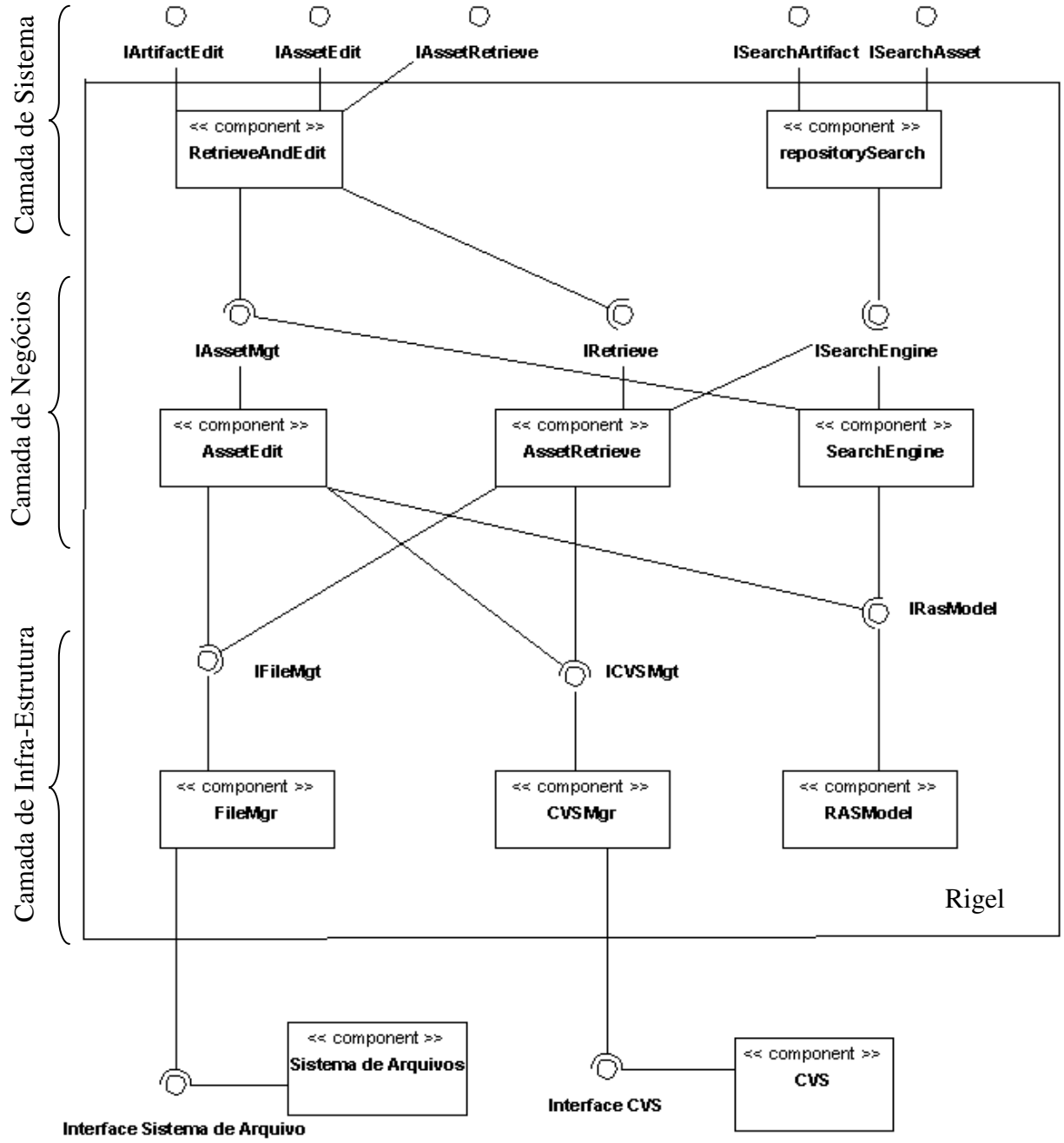

Figura 10 – Arquitetura Interna do Repositório Rigel

RepositorySearch é um componente de sistema responsável pelas pesquisas de bens e artefatos. RepositorySearch possui uma interface que permite especificar parâmetros para pesquisas de bens. A pesquisa é executada pela camada de negócio, e os resultados são empacotados em objetos da interface ISearchAsset antes de serem retornados ao cliente da interface. RepositorySearch possui dependência de dois outros componentes da camada de negócio: AssetEdit para obtenção de detalhes sobre bens presentes em resultados de pesquisas; e SearchEngine para implementação da pesquisa.

RepositorySearch pode ser estendido para realizar pesquisas de forma distribuídas. Para isto, é necessário incluir na arquitetura componentes conectores que ligariam RepositorySearch a máquinas de buscas (search engines) de outros repositórios.

Os componentes RetrieveAndEdit e RepositorySearch possuem uma interface Java RMI para comunicação com as camadas de interface e diálogo com o usuário. Uma interface Java RMI permite um acesso remoto as funcionalidades providas pela interface, possibilitando a construção de aplicações distribuídas.

A seguir discutiremos a camada de negócio.

4.1.3 Camada de Negócio

A camada de negócios é composta pelos componentes AssetEdit, AssetRetrieve e SearchEngine, como pode ser visto na Figura 10.

AssetEdit é responsável por persistir os metadados do bem e dos artefatos, utilizando o FileMgr e o CVSMgr para realizar a persistência. O FileMgr é utilizado para realizar a persistência dos metadados no sistema de arquivos, enquanto o CVSMgr faz a persistência dos artefatos utilizando o CVS. O AssetEdit também utiliza o componente RASModel para fazer a conversão de arquivos no formato RAS(XML-Schema) para o modelo de objetos que representa os metadados.

O AssetRetrieve provê funcionalidades de recuperação de bens. Este componente tem dependências de CVSMgr, FileMgr e SearchEngine. As razões para estas dependências estão

relacionadas a seguir: CVSMgr é utilizado para extrair os artefatos que compõem um bem do repositório. FileMgr auxilia nas manipulação de arquivo necessárias ao empacotamento do bem. SearchEngine é utlizado para encontrar a localização de bens e artefatos a serem recuperados.

SearchEngine é o componente que implementa a máquina de busca (search engine) do Rigel. Este tem dependência de RASModel, que auxilia na a leitura dos metadados em XML durante as atividades de indexação de bens. O SearchEngine utiliza o Lucene, uma biblioteca de código livre do projeto Apache, que auxilia na implementação de uma máquina de busca.

O componente RetrieveAndEdit é responsável por adicionar, alterar e remover bens e artefatos. AssetRetrieve recupera um bem do repositório. Ele liga-se com FileMgr, que gerencia os arquivos e com CVSMgr para recuperar os artefatos que compõem um bem. AssetEdit é responsável por persistir metadados do bem e artefatos, utilizando o FileMgr e o CVSMgr para realizar a persistência. O AssetEdit também utiliza o componente RASModel para fazer a conversão de arquivos no formato RAS(XML-Schema) para o modelo de objetos que representa os metadados. Esse modelo de objetos é definido pelos *profiles*. Consequentemente, cada componente que use estes objetos deverá dar suporte aos *profiles* existentes no repositório. O RepositorySearch formata os dados e envia requisições de busca para o componente SearchEngine. Ao indexar os metadados, o SearchEngine utiliza o RASModel a fim de extrair as informações a serem indexadas dos arquivos em formato RAS.

4.1.4 Camada de Infra-Estrutura

Na camada de infra-estrutura temos três componentes: RASModel, FileMgr e CVSMgr. A seguir discutiremos mais detalhes sobre estes componentes.

RASModel é o componente responsável por auxiliar as conversões de metadados do formato RAS [RAS04] para objetos Java. O metadados manipulados pelo RASModel seguem especificações XML Schema segundo o padrão RAS. O RASModel utiliza o JAXB para auxiliar nas conversões de objetos Java para XML e vice-versa. O Sun Java XML Binding (JAXB) [JAXB] é um framework da Sun para realizar mapeamentos e manipulações de dados em XML Schema.

FileMgr é o componente que realiza todas as operações que envolvem arquivos, funcionando como uma camada entre o Rigel e o sistema de arquivos do sistema operacional. FileMgr provê serviços de cópia, leitura, gravação, criação e remoção de arquivos.

CVSMgr é responsável pela interface com o sistema GCS CVS [CVS]. CVSMgr realiza a persistência de artefatos através do CVS. CVSMgr oferece operações de adição, recuperação e remoção de artefatos no CVS. Atualizações em artefatos devem ser rastreáveis, e por isso o CVSMgr disponibiliza as operações de *check-out* e *check-in*. O *check-out* ainda é utilizado na recuperação de versões de artefatos.

4.1.5 Interoperabilidade Repositório-Repositório e Repositório-IDE

O projeto Rigel considerou o requisito de qualidade interoperabilidade da seguinte forma: protocolo de comunicação entre sistemas e formato de dados compartilhados (3.2.1). As soluções para interoperabilidade entre repositórios e interoperabilidade entre repositório e IDE estão baseadas na escolha um protocolo de comunição para a interface de sistema, e a definição de um formato padronizado para o empacotamento de bens.

Quanto ao protocolo de comunicação, os fatores decisivos na escolha foram flexibilidade e difusão. As interfaces de Rigel apóiam Java RMI - um protocolo para aplicações distribuídas, disponível em múltiplas plataformas. A utilização de Java RMI permite a integração do Rigel com outros repositórios que estejam funcionando em outros servidores, em diferentes plataformas.

Em relação ao formato de dados compartilhados, a solução adotada foi a utilização do RAS, um padrão para metadados e empacotamento de bens de software reutilizáveis. Todos os bens, recuperados através do Rigel, estão empacotados segundo o padrão RAS.

4.1.6 Integração com Sistema de Gerencia de Configuração de Software

O sistema GCS escolhido para ser integrado ao Rigel é o CVS. Os critérios utilizados nesta seleção foram: acessibilidade – o CVS é uma ferramenta de código livre, e difusão – o CVS é uma das ferramentas mais populares do mercado.

A integração com o CVS é realizada através do componente CVSMgr. Este componente é responsável pela implementação das chamadas ao CVS de *check-out*, *check-in* de arquivos e diretórios. Através destas operações conseguimos implementar todas as funcionalidades para o versionamento de artefatos: criação, alteração, remoção e recuperação de arquivos e diretórios no CVS.

Para auxiliar esta implementação, foi utilizado uma biblioteca do projeto NetBeans [NetBeans], chamada NetBeans-CvsClient. Esta biblioteca reúne soluções de comunicação com CVS, tratamento de erros e adaptação da interface CVS para Java.

4.1.7 Formato dos Metadados e Reuso

O repositório Rigel adota um padrão para definição de metadados: o RAS. Este padrão define um formato para metadados de bens baseado em arquivos XML, definido segundo especificações XML Schema. As propriedades fundamentais de especificações de bens, são agrupadas em uma entidade chamada Core RAS. O padrão RAS prevê o suporte a diferentes tipos de bens, através do conceito de "*Profiles*", especificações XML Schema que adicionam propriedades ao Core RAS.

Sendo o Rigel um repositório de bens de software reutilizáveis em geral, mas voltado para componentes, adicionamos profiles RAS para bens relacionados a componentes.

Baseados na definição do Core RAS, criamos quatro Profiles: AbstractComponent, ConcreteComponent, InterfaceDefinition e Configuration Profile que descrevem respectivamente uma definição de interface, um componente abstrato, um componente concreto e uma configuração. A Figura 11 mostra o diagrama de classe dos "Profiles" definidos.

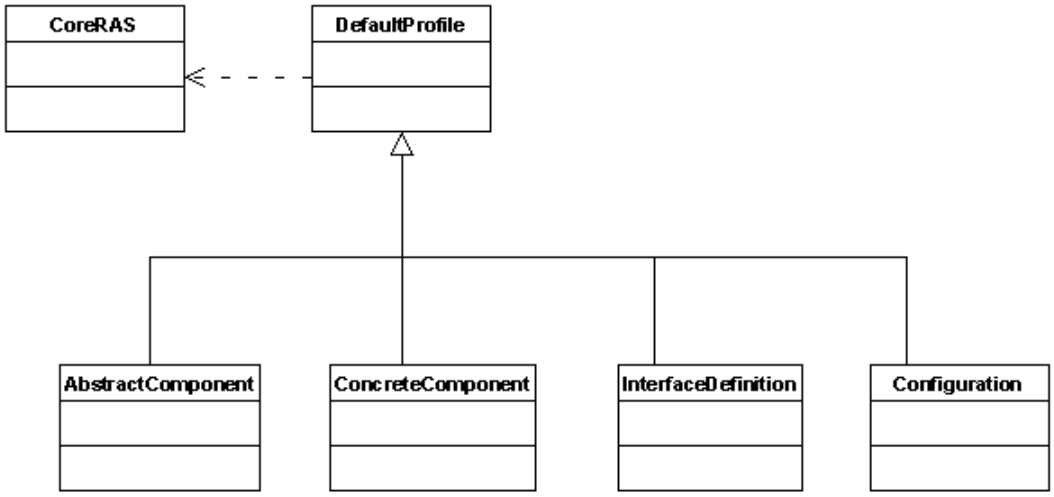

Figura 11 – Hierarquia de profiles utilizada pelo Rigel

O componente abstrato é a descrição do comportamento e a especificação de um elemento arquitetural. O componente concreto é a implementação do comportamento especificado pelo componente abstrato. As definições de interface contém o nome e as operações das interfaces que serão implementadas pelos componentes. A configuração é a materialização da arquitetura. Estas extensões do padrão RAS estão baseadas no modelo conceitual para DBC e arquitetura do software proposto por Guerra [Guerra05], a seção 2.3.1, contém mais detalhes sobre este modelo conceitual.

A decisão de ter quatro Profiles separados tem como objetivo aumentar o grau de reuso dos bens de um repositório. O usuário pode escolher reutilizar somente um dentre esses bens, respeitando as dependências entre eles. Isso também auxilia na separação entre especificação e implementação, uma propriedade básica de componentes.

O RASModel é o componente responsável pela manipulação de profiles, e conversão dos metadados do formato genérico Rigel utilizando objetos Java para o formato XML especificado pelo padrão RAS. Este componente tem a responsabilidade de isolar a manipulação de dados RAS do restante do sistema. Assim, todos os outros componentes do Rigel podem ler e escrever propriedades em bens, através da utilização dos serviços de RASModel. Este isolamento facilita a manutenibilidade e extensibilidade do Rigel: ao adicionar novos "Profiles", o único

componente que necessita alterações é o RASModel. Esta característica é um reflexo do projeto *data-driven* do Rigel.

As propriedades definidas pelo modelo RAS estão organizadas de forma hierárquica, existindo elementos no modelo RAS que contém outros elementos. Por isso, adotamos neste trabalho a seguinte nomenclatura para propriedades RAS: todos os elementos são concatenados de acordo com a ordem crescente de sua hierarquia, a propriedade é adicionada ao final da cadeia, um ponto separa os elementos, um traço separa a propriedade, um índice entre colchetes especifica elementos de listas. Por exemplo, se temos a seguinte seqüência de elementos: *Asset* contém *Solution*, que contém uma lista de artefato; para representar a propriedade *reference* do artefato 1 usamos: *Asset.Solution.Artifact[1]_reference*.

A seguir descrevemos brevemente os modelos de metadados criados para o Rigel: *AbstractComponent, ConcreteComponent, InterfaceDefinition e Configuration*.

AbstractComponent

O *AbstractComponent* é modelo para componentes abstratos. Neste modelo são armazenadas as seguintes informações:

- Artefatos com especificações do componente.
- Algumas propriedades relativas à especificação do componente: nome do componente, informações sobre a qualidade do serviço provido pelo componente, informações sobre a sincronização das operações de um componente e a plataforma alvo do componente.
- As referências para as interfaces relativas à especificação e também as seguintes propriedades destas interfaces: nome da interface, direção da interface e estado da interface.

ConcreteComponent

O modelo para componente concreto pode representar uma implementação de um componente sem dependências, denominado componente elementar, ou uma implementação de um componente dependente de outros componentes, denominado componente composto.

Neste modelo as seguintes informações são armazenadas:

- No caso de componentes compostos, referências para os componentes dependentes, as interfaces e portas requeridas.
- Os artefatos que contêm a implementação do componente.

InterfaceDefinition

O modelo *InterfaceDefinition* armazena bens contendo interfaces. Neste modelo são armazenadas as seguintes informações:

- Propriedades da interface, tais como: nome da interface, descrição textual da interface e estado de desenvolvimento da interface
- Operações da interface e suas propriedades: nome da operação e descrição textual da operação.
- Parâmetros das operações, contendo: nome do parâmetro, tipo do parâmetro, direção do parâmetro e posição do parâmetro.
- Condições das operações, e suas propriedades: tipo da condição (pré-condição, pós-condição e invariante), descrição textual da condição e a expressão que modela a condição.

Configuration

O modelo *Configuration* armazena uma configuração de uma arquitetura concreta. Neste modelo, são armazenadas referencias para todos os bens de componentes concretos que compõem uma configuração arquitetura concreta.

O Apêndice B contém mais detalhes sobre os modelos de metadados do Rigel.

4.2 Edição e Armazenamento de Bens

Devido ao fato do Rigel apoiar diversos "Profiles", podemos considerar os metadados de Rigel como sendo semi-estruturados.

Devido à dificuldade de armazenamento de dados semi-estruturados em bancos de dados relacionais, escolhemos armazenar os metadados de Rigel em arquivos XML. Estes dados são armazenados seguindo as especificações RAS:

- Os arquivos XML estão em conformidade ao um esquema definido por um profile RAS.
- Os *profiles* RAS são especificados em XML Schema. Isto significa que os arquivos de dados XML devem ser validados de acordo com a especificação XML Schema referente ao *profile* utilizado pelo bem.
- Cada bem tem os seus metadados armazenados em um arquivo XML.
- O arquivo de metadados contém referências para a localização dos artefatos no repositório.

4.3 Pesquisa de Bens

Os metadados de Rigel estão armazenados em arquivos XML, seguindo especificações XML Schema, conforme explicado nas seções 2.5.2 e 2.5.3. O fato do formato dos metadados variar de acordo com o tipo de bem, ou seja, os metadados são semi-estruturados, dificulta o armazenamento e pesquisa destes dados utilizando banco de dados relacionais.

A solução adotada para implementar a pesquisa de bens foi a utilização de uma máquina de busca (search engine). Um problema relacionado às máquinas de buscas é a necessidade de interpretação de arquivos XML ao fazer a indexação destes, pois geralmente o *parsing* do conteúdo dos arquivos não é feito pelas *search engines*. Para resolver o problema do *parsing* dos arquivos XML, utilizamos o componente RASModel.

RASModel é o componente responsável pela conversão de XML para objetos Java. No caso da implementação da pesquisa Rigel, utilizamos o RASModel para realizar a indexação de arquivos de metadados. O RASModel converte o conteúdo XML em objetos Java das classes AssetNode e AssetProperty. O componente SearchEngine então consegue analisar estes objetos e adicionar as propriedades à maquina da busca para realizar a indexação.

4.4 Recuperação de Bens

A recuperação de bens é uma operação que envolve o empacotamento dos artefatos que compõem o bem, juntamente com os seus metadados.

O formato RAS apóia bens empacotados em vários arquivos e bens empacotados em um único arquivo. A fim de facilitar as transferências de bens do servidor para máquinas clientes, o formato de empacotamento em um único arquivo foi o escolhido para ser disponibilizado nas operações de recuperação de bens.

Os artefatos que compõem um bem são comprimidos através de uma compressão Zip. Os caminhos (*paths*) onde se localizam os artefatos são mantidos no pacote comprimido. Um arquivo XML contendo os metadados para o bem, juntamente com a sua especificação em XML Schema também são incluídos no pacote. Todas as referências presentes nos metadados à artefatos, são válidas para os caminhos (*paths*) de artefatos armazenados no pacote.

4.5 Resumo

Neste capítulo apresentamos o projeto do repositório Rigel. Inicialmente foi apresentado o projeto arquitetural. Em seguida foi discutido o modelo de metadados do Rigel. E por último, foi apresentado o projeto detalhado das principais funcionalidades do Rigel: edição e armazenamento de bens, pesquisa de bens e recuperação de bens.

A arquitetura do repositório Rigel está dividida em quatro camadas: camada de interface e diálogo com o usuário, camada de sistema, camada de negócios e camada de infra-estrutura. A

camada de Interface e Diálogo é responsável por apresentar informações ao usuário, capturar as entradas de dados e gerenciar os diálogos com o usuário em uma sessão. A camada de sistema é a representação externa do sistema e a suas interfaces provêem acesso às funcionalidades do sistema. A camada de negócio implementa regras e informações de negócio. A camada de infra-estrutura é responsável por apoiar a implementação da camada de negócio.

O modelo de metadados apresentado está baseado no trabalho de Guerra [Guerra05]. Os *profiles* RAS foram estendidos, e esta adaptação deu origem a quatro novos *profiles*: AbstractComponent, ConcreteComponent, InterfaceDefinition e Configuration.

No projeto da funcionalidade "Edição e Armazenamento de Bens", discute-se como os metadados são convertidos para XML. Na seção "Pesquisa de Bens", mostrou-se a principais decisões de projeto em relação a construção de uma máquina de busca para metadados em formato RAS. Finalmente, a seção "Recuperação de Bens" apresenta detalhes de como os artefatos são recuperados do repositório e empacotados.

Capítulo 5 Implementação do Repositório Rigel

Neste capítulo, apresentamos detalhes sobre a implementação do repositório Rigel. Inicialmente descrevemos algumas características gerais de implementação. Posteriormente, comentamos a implementação de três funcionalidades importantes: armazenamento, pesquisa e recuperação de bens.

Java foi a linguagem utilizada para implementar o repositório Rigel. A camada Web foi implementada em JSP, e o TomCat foi o servidor JSP/HTTP escolhido.

5.1 Edição e Armazenamento de Bens

Nesta seção descrevemos o processo de edição de metadados de um bem.

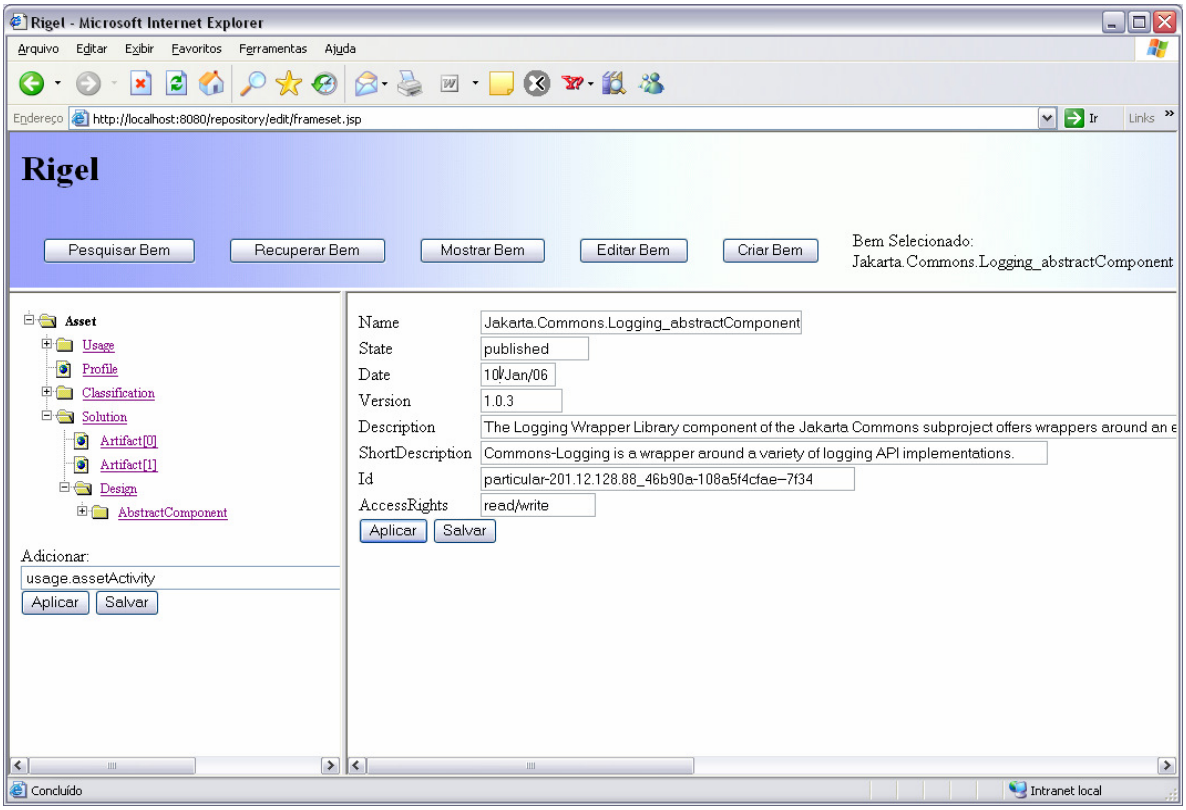

Figura 12 – Tela de alteração de metadados do Rigel

O processo de alteração de bens no repositório é composto pelas seguintes etapas:

1. Montagem da Tela da Alteração de Metadados (Figura 12)

    a. A partir de um bem selecionado, o usuário escolhe a opção alterar bem. A camada de dialogo requisita a camada de sistema os metadados do bem selecionado. O processamento na camada de sistema ocorre de acordo com os seguinte passos (Figura 13) :

        i. O componente AssetEdit requisita ao SearchEngine a localização do bem no sistema de arquivos.

        ii. AssetEdit solicita ao FileMgt que leia o arquivo XML contendo os metadados do bem.

        iii. O AssetEdit requisita ao RASModel para converter o conteúdo RAS XML em objetos Java.

    b. A partir dos metadados retornados pela camada de sistema, a camada diálogo e interface retorna o navegador uma página HTML para edição dos metadados.

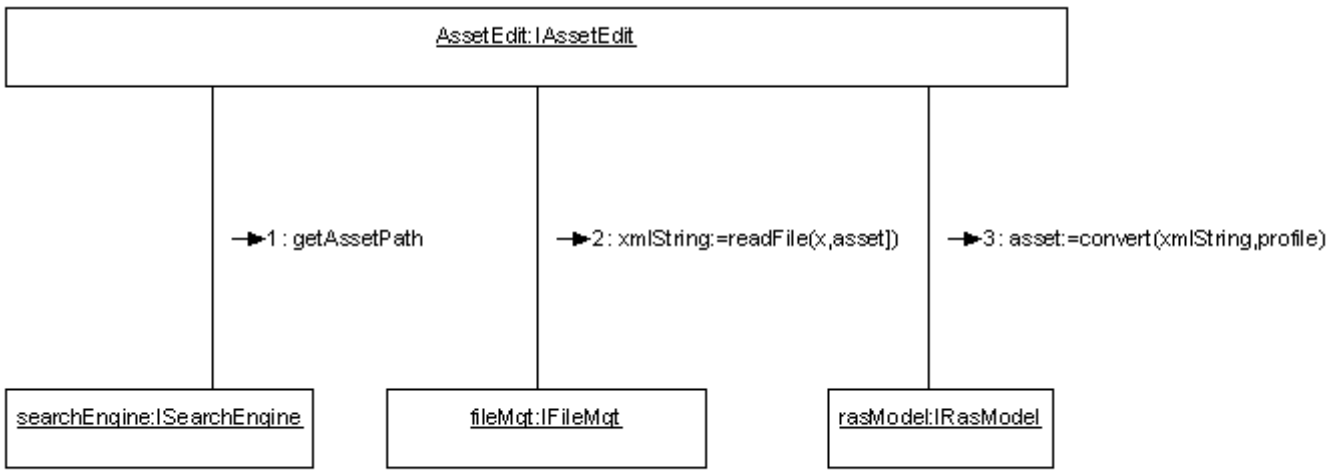

Figura 13 – Diagrama de Colaboração para a leitura de metadados

2. Alteração dos metadados

    a. O usuário altera os metadados através do navegador web.

b. Após as alterações, o usuário submete os dados. A camada de diálogo requisita a camada de sistema que altere o bem, os valores de metadados são enviados. O processamento na camada ocorre de acordo com os seguintes passos (Figura 14):
    i. O componente AssetRetrieve requisita ao RASModel que converta os objetos Java para RAS XML.
    ii. O AssetRetrieve requisita informações sobre a localização do arquivo de metadados ao searchEngine.
    iii. O AssetRetrieve solicita ao FileMgt que grave um novo arquivo de metadados com as última alterações.
    iv. AssetRetrieve requisita ao SearchEngine que atualize seus índices.

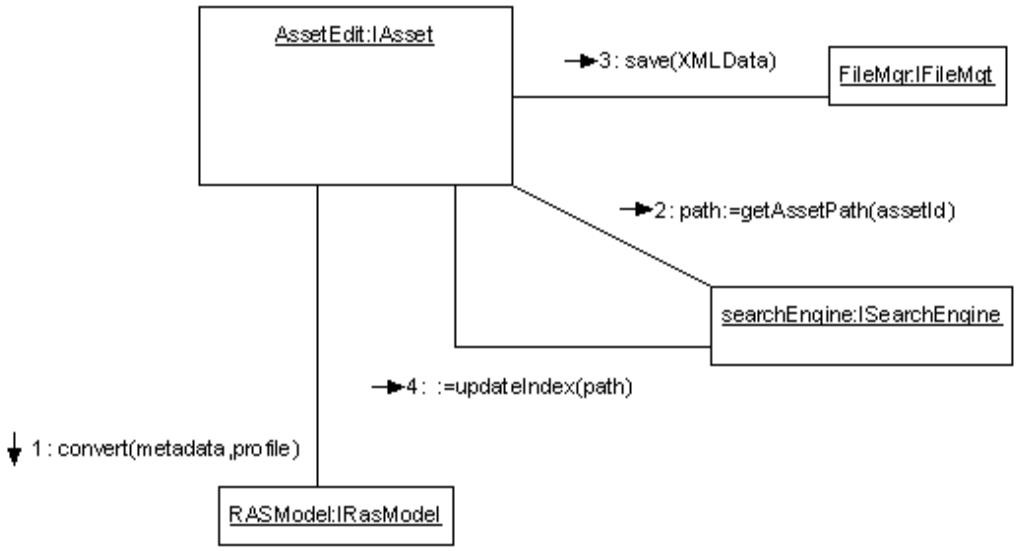

Figura 14 – Diagrama de colaboração para a alteração de metadados.

Os artefatos que compõem um bem são armazenados no sistema GCS CVS. O processo de alterações de artefatos no CVS segue as seguintes etapas:

- O usuário envia o artefato a ser adicionado através do navegador da internet.

- A camada de diálogo armazena o artefato em um diretório temporário
- O Componente AssetEdit faz um checkout dos artefatos que compõem o componente em um "workspace" temporário
  - O CVSMgr requisita um check-out ao CVS
- O Componente AssetEdit copia o artefato para o "workspace temporário"
- O Componente faz o check-in do artefato
  - O CVSMgr requisita o check-in ao CVS

5.2 Pesquisa de Bens

Lucene [Hatcher04] é a biblioteca escolhida para a implementação da máquina de busca do componente SearchEngine. O Lucene é uma biblioteca de uma máquina de busca de alto desempenho desenvolvida em Java. O Lucene é um projeto de código livre, que faz parte do projeto Apache [Lucene]. Implementação em Java, popularidade e código aberto foram os fatores que mais influenciaram na escolha do Lucene.

No Rigel, todas as propriedades dos metadados estão disponíveis para pesquisa. A pesquisa de bens é baseada em palavras-chave. O usuário escolhe de uma lista, as propriedades a serem pesquisadas, e atribui valores a estas propriedades. Ao submeter a pesquisa, o sistema verifica todos os bens que satisfazem todas as condições. O resultado é uma breve descrição dos bens selecionados (Figura 15 – Tela de pesquisa do Rigel).

Os valores dos parâmetros da pesquisa não precisam ter um valor exatamente igual ao valor da propriedade. Para uma condição selecionar um bem, basta que o parâmetro da pesquisa tenha parte de valor da propriedade. Por exemplo, uma pesquisa com parâmetros: Asset_Description = "Java", selecionaria um bem com a propriedade Asset_Description = "FileUpload is a Java Component...". Esta funcionalidade se deve a utilização de um "Analyser" adequado para o Lucene. O *Whitespace analyser* separa, de acordo com espaços em branco, o conteúdo de valores de propriedades antes da indexação. Assim, as palavras que compõem um valor de propriedade são indexadas separadamente, permitindo a procura por palavras-chave.

Algumas propriedades do modelo RAS são utilizadas para classificar componentes. A classificação de componentes é uma das estratégias mais eficientes para pesquisas de

componentes [PrietoDiaz87]. As propriedades Asset.Classification. DescriptorGroup. Descriptor_Name e Asset. Classification. DescriptorGroup. Descriptor_Value devem ser utilizadas para classificar bens. O Descriptor_Name contém o nome da faceta (*facet*) e Descriptor_Value contém o valor. A pesquisa por classficação de componentes, pode ser realizada incluindo estas duas propriedades.

A pesquisa de bens ainda pode ser utilizada para navegar entre bens. Algumas propriedades de bens referenciam outros bens. São exemplos destas propriedades: RelatedAsset e Implementation. CompositeComponent. ComponentBasedView. InterfaceConnection[0]. SubComponent_Id AssetId. Pesquisas envolvendo estas propriedades podem revelar em seus resultados referências para bens relacionados, a seleção de um determinado resultado executa a navegação.

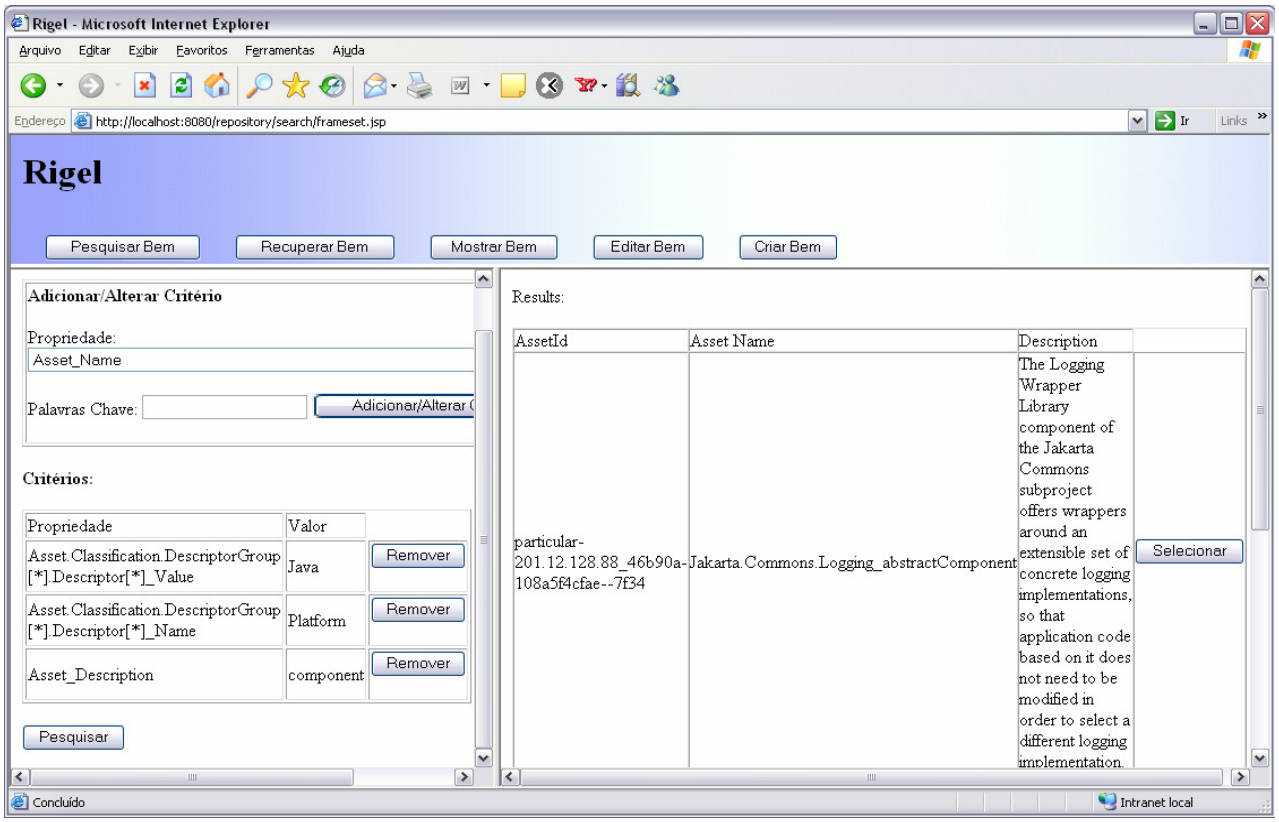

Figura 15 – Tela de pesquisa do Rigel

5.3 Recuperação de Bens

AssetRetrieve é o componente que implementa a maior parte da funcionalidade de recuperação de bens.

Para a recuperação de artefatos, o AssetRetrieve utiliza o componente CVSMgr para recuperar todos os artefatos que compõem um bem.

A transferência do bem do servidor para a estação cliente é implementada via protocolo HTTP. Uma das vantagens na utilização do HTTP é a integração com a interface web, o Rigel já oferece uma interface HTTP/JSP, o que facilita a recuperação de bens via http. Após o empacotamento de um bem, este é armazenado em uma área compartilhada. A URL deste bem é repassada ao usuário, que por sua vez inicia a transferência do bem a partir do mesmo Navegador Web que opera o sistema.

O processo de recuperação de bens segue, em linha gerais, os seguintes passos:

- O usuário requisita a recuperação de um bem
- O componente AssetRetrieve procura o bem a ser recuperado
- AssetRetrieve lê os metadados do bem
- AssetRetrieve requisita ao CVSMgr que recupere todos os artefatos que compõem o bem
    - Estes artefatos são armazenados em um diretório ("*CVS workspace*") temporário
- AssetRetrieve requisita ao FileMgt que copie os metadados e suas definições para o diretório temporário
- AsestRetrieve compacta todos os arquivos, em um arquivo de nome <nome do bem>.ras
- AssetRetrieve disponibiliza o arquivo compactado em um diretório compartilahdo
- O repositório retorna a URL do bem empacotado ao usuário
- O usuário requisita a transferência (*download*) do bem empacotado.

5.4 Resumo

Este capítulo apresenta os detalhes de implementação mais relevantes do repositório Rigel. Inicialmente discutem-se alguns aspectos gerais como linguagens utilizadas. As seções seguintes comentam a implementação das principais funcionalidades: edição e armazenamento de bens, pesquisa de bens e recuperação de bens.

A seção “Edição e Armazenamento de Bens” descreve o processo de edição de metadados de um bem. A seção “Pesquisa de Bens” mostra como o Lucene foi utilizado para implementar a máquina de buscas de Rigel. E finalmente a seção “Recuperação de Bens“ mostra o processo de empacotamento e disponibilização de bens para recuperação.

Capítulo 6 Estudos de Caso usando o Rigel

Neste capítulo discutiremos um pouco sobre a utilização do repositório Rigel. Inicialmente mostramos como o Rigel pode apoiar um processo de desenvolvimento baseado em componente, no nosso caso o UML Components.

Na seção seguinte, escolhemos alguns componentes de código livre, bem difundidos na Internet, para mostrar como o Rigel apóia componentes de uso geral.

6.1 Estudo de Caso 1: Apoiar um Processo DBC

O Processo UML Components [Cheesman+00] é um processo de desenvolvimento de software baseado em componentes, que trata do problema de especificar e arquitetar sistemas baseados em componentes. Para um estudo do suporte do Rigel a DBC, selecionamos o UML Components por ser um processo simples e bastante difundido na área de componentes.

A seguir, percorreremos todas as fases do UML Components mencionando as operações do repositório que auxiliam o desenvolvimento de componentes. A fim de ilustrar este estudo, utilizaremos o mesmo sistema reservas de hotel mostrado em [Cheesman+00]. A seção 2.1.2 contém um resumo das fases do processo UML Componentes.

6.1.1 Definição de Requisitos

Definição de requisitos é a primeira fase do processo UML Components (Figura 16). Nesta fase são criados alguns documentos básicos do projeto, tais como: diagramas de atividades, modelo conceitual do negócio, documento de visão do projeto, casos de uso. Como estes documentos são relativos ao projeto como um todo, iniciamos a nossa interação com o Rigel,

criando um bem que representa o projeto inteiro. Para o sistema de reserva de hotel, objeto de nosso estudo, criamos um bem chamado “Sistema de Reserva de Hotel”.

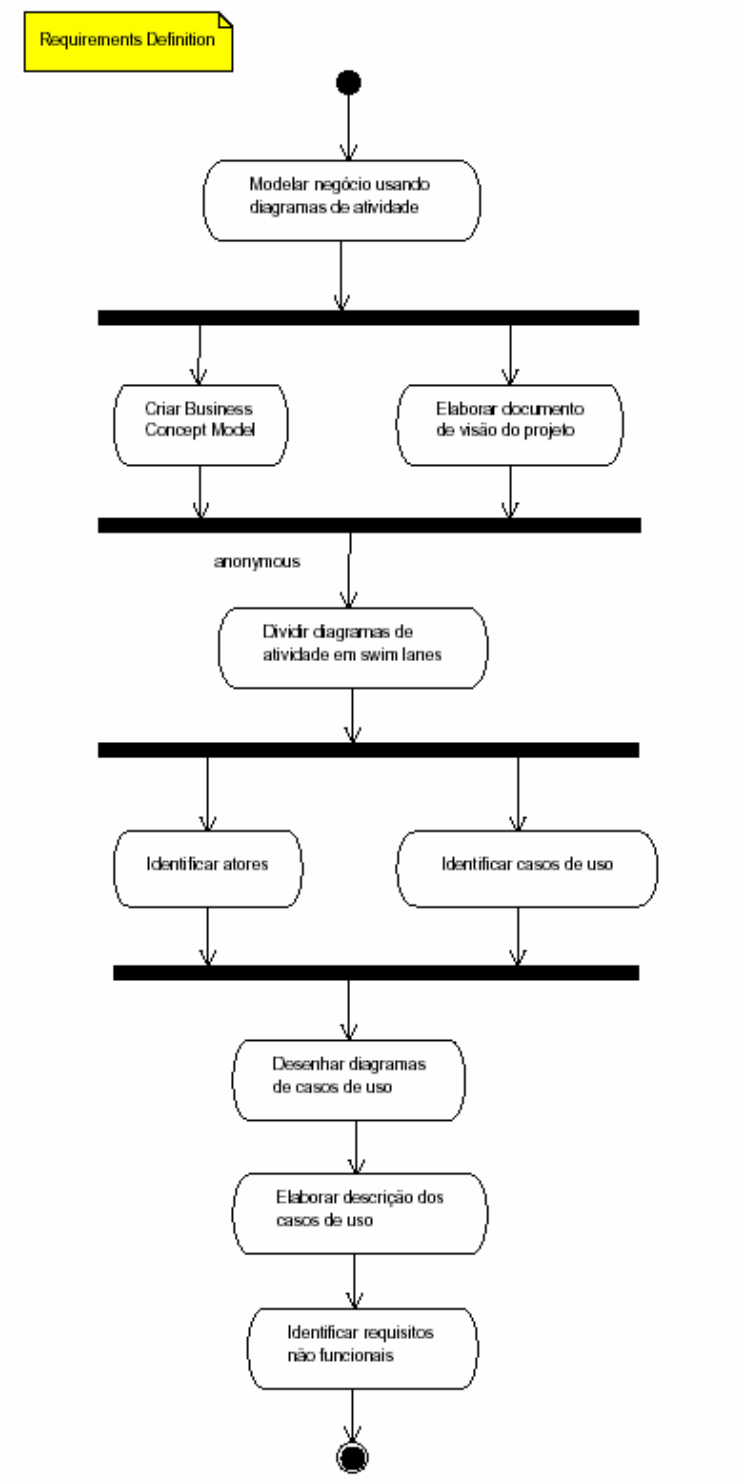

Figura 16 – Fase de Definição de Requisitos

Ao cadastrar o bem “Sistema de Reserva de Hotel”, preenchemos algumas propriedades importantes tais como descrição do bem, classificação e versão.

Em seguida, os novos documentos são adicionados ao Rigel, na medida em que vão sendo criados. Estes documentos são adicionados como artefatos, dentro do bem “Sistema de Reserva de Hotel”.

Ainda nesta fase, alguns documentos são atualizados, por exemplo, o diagrama de visão do projeto e o diagrama de atividades. Para essas atividades de atualização, os artefatos foram recuperados do repositório, alterados pelo usuário e depois uma nova versão foi adicionada ao repositório.

6.1.2 Identificação de Componentes – Identificação de Interfaces de Sistema e Operações.

A fase seguinte do UML Components tem como objetivo a criação da interface de sistema (Figura 17).

No sistema de hotel temos duas interfaces de sistema IMakeReservation e ITakeUpReservation. Para cada nova interface de sistema criamos um bem com um profile do tipo “Interface”. Nestes bens cadastramos a interface e suas operações.

A fim de permitir uma melhor rastreabilidade dos bens criados durante o desenvolvimento deste sistema, cadastramos para cada bem criado, um “relatedAsset ” que referencia o “Sistema de Reserva de Hotel”. Assim, sempre podemos listar todos os bens relacionados ao sistema de reserva de hotel, fazendo uma pesquisa por todos os bens cuja propriedade “RelatedAsset_AssetId” tem como valor o AssetId do “Sistema de Reserva de Hotel”.

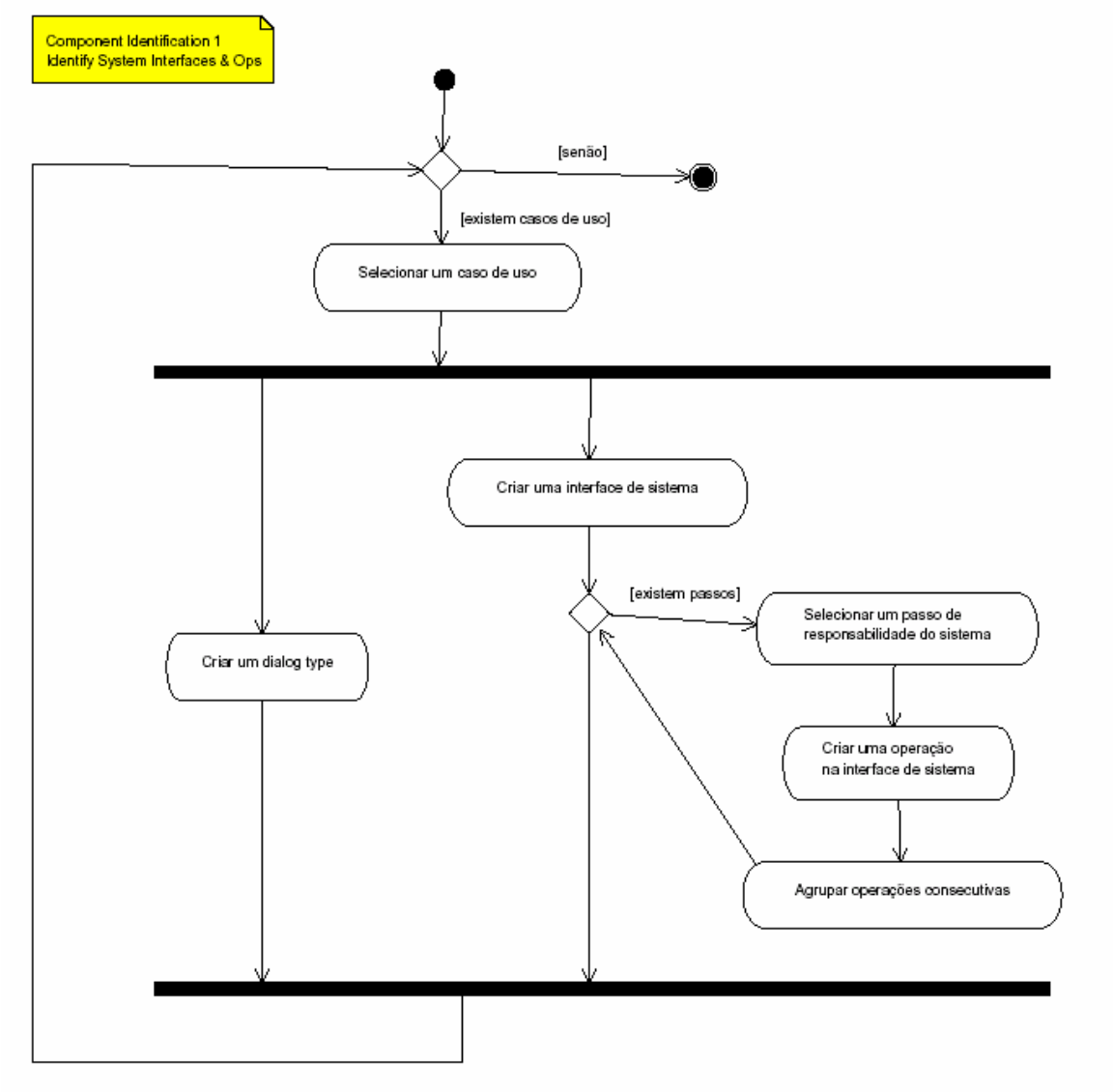

Figura 17 - Identificação de Componentes – Identificação de Interfaces de Sistema e Operações.

6.1.3 Identificação de Componentes – Desenvolver Modelo de Tipos do Negócio – Identificar Interfaces de Negócio

Nesta etapa, o primeiro passo foi a criação do modelo de tipos do negócio a partir do modelo conceitual do negócio (Figura 18). Para isto recuperamos o modelo conceitual do negócio, e após a criação do modelo de tipos do negócio, adicionamos este novo artefato no "Sistema de Reserva de Hotel".

O passo seguinte é a criação da primeira versão das interfaces de negócio. As interfaces IHotelMgtInterface e ICustomerMgtInterface foram adicionadas no Rigel como novos bens, associados a um profile do tipo "InterfaceDefinition". Finalmente os artefatos que representam as interfaces de negócio são adicionados aos respectivos bens de interface.

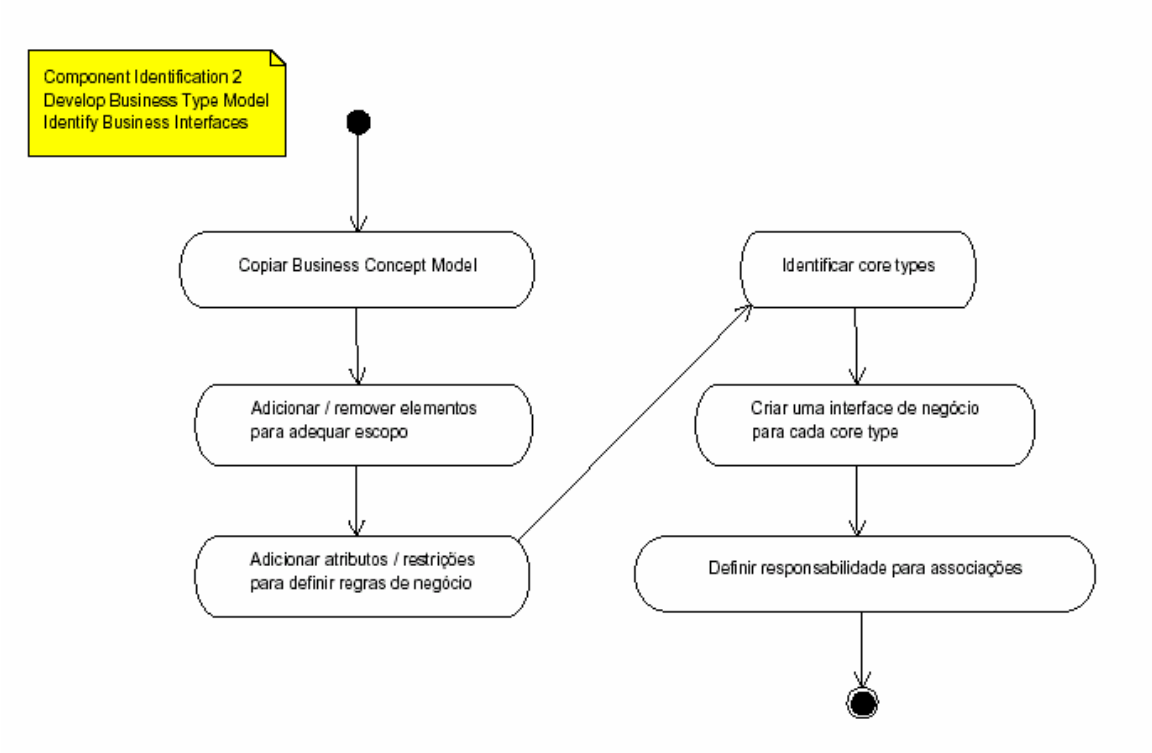

Figura 18 - Identificação de Componentes – Desenvolver Business Type Model – identificar interfaces de negócio

6.1.4 Identificação de Componentes – Criar especificações de Componentes Iniciais e Arquitetura

Nesta etapa (Figura 19), as interfaces de negócio e de sistemas são analisadas, e a partir destas criamos os componentes iniciais do sistema. Estes novos componentes são adicionados ao repositório como novos bens. Deste modo temos para os componentes de negócio os seguintes bens: BillingSystemAbstractComponent, CustomerMgtAbstractComponent, HotelMgrAbstractComponent; e um bem representando componente de sistema:

ReservationSystemAbstractComponent. Estes bens seguem o abstractComponent *profile*, que é apropriado para especificações de componentes.

Estes novos bens de componentes abstratos são ligados aos bens de interfaces através da propriedade: AbstractComponent.ExternalProperty[0].Interface[0].RelatedAsset[0]. Assim temos os seguintes relacionamentos entre bens:

- ReservationSystemAbstractComponent associado ao bem: IMakeReservation e ITakeUpeservation
- BillingSystemAbstractComponent associado ao bem: IBillingInterface
- CustomerMgtAbstractComponent associado ao bem: ICustomerMgtInterface
- HotelMgrAbstractComponent associado ao bem: IHotelMgtInterface

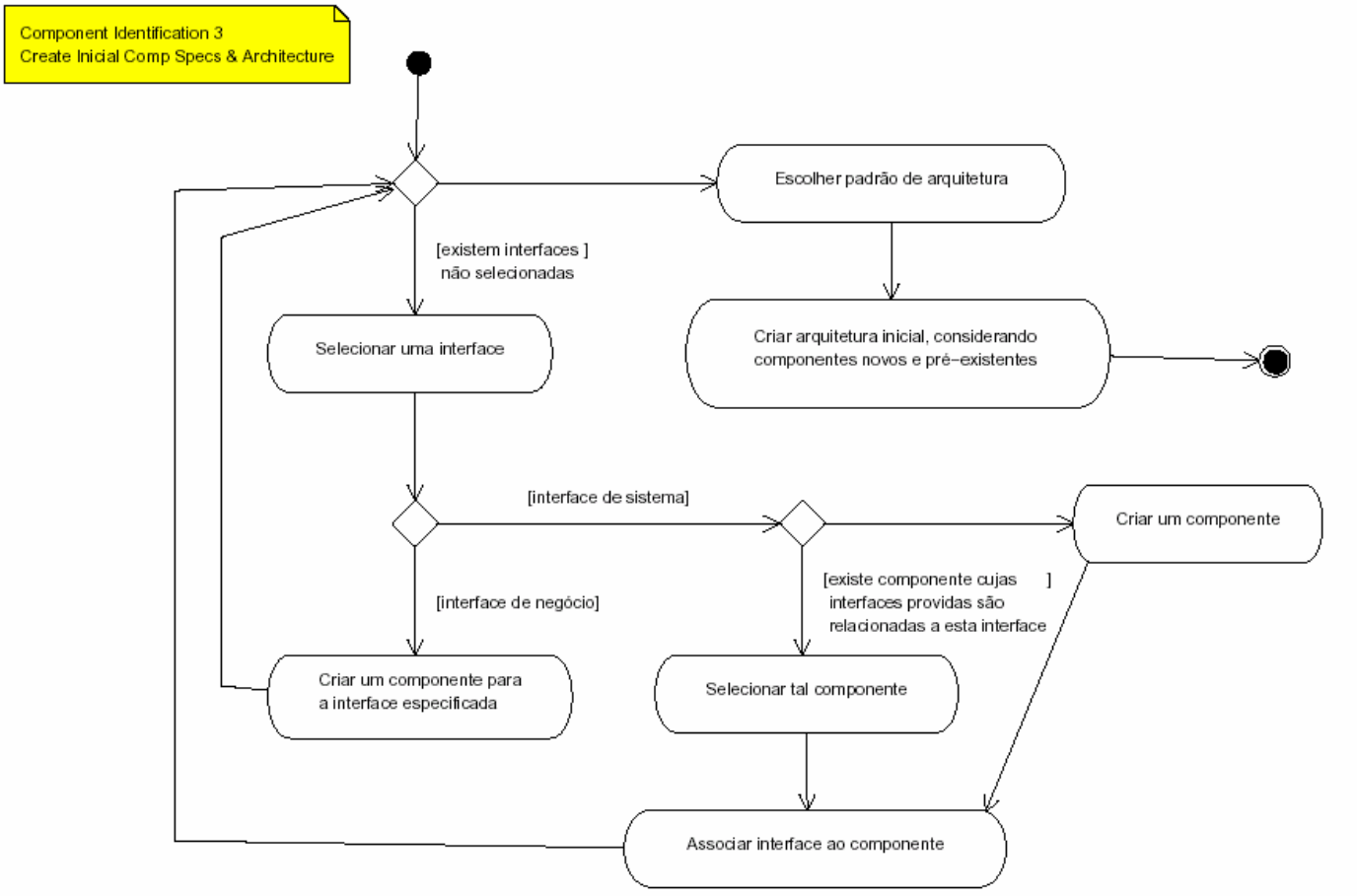

Figura 19 - Identificação de Componentes – Criar especificações de Componentes Iniciais e Arquitetura

Seguindo o processo, a arquitetura inicial do sistema é criada. Esta arquitetura é armazenada no bem "SistemadeReservadeHotel_configuration", como um novo artefato. Apesar

de existir um bem apropriado para armazenar a arquitetura, nesta fase é mais apropriado armazená-lo no bem principal do sistema, por se tratar de uma versão preliminar da arquitetura.

6.1.5 Iteração de Componentes: descobrir operações de negócio

Nesta etapa (Figura 20) criamos o primeiro conjunto de operações para as interfaces de sistema e de negócio. Estas operações foram adicionadas nos bens como artefatos e também como propriedades dos metadados do bem.

Assim temos as seguintes propriedades representando as operações no profile Interface:

- Solution .Design.InterfaceDefinition[i].Operation[j]_Name
- Solution .Design.InterfaceDefinition[i].Operation[j]_ DevelopmentState
- Solution .Design.InterfaceDefinition[i].Operation[j]_ Decription

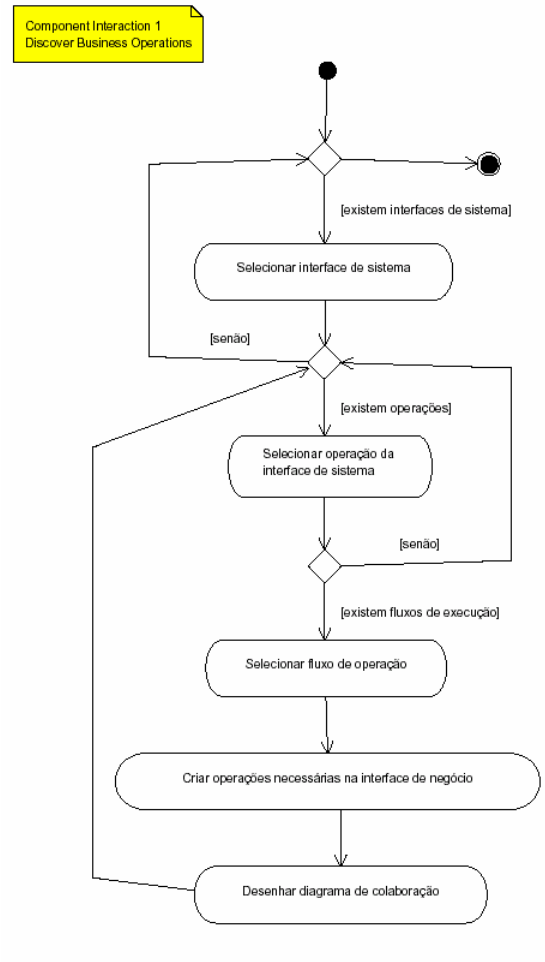

Figura 20 - Iteração de Componentes: descobrir operações de negócio

6.1.6 Iteração de Componentes: refinar interfaces e operações

O próximo passo no processo UML Components, é o detalhamento de interfaces e operações (Figura 21). Para detalhar as operações, foi necessário recuperar os artefatos referentes às interfaces alteradas. Após as alterações, estes artefatos foram adicionados novamente no Rigel.

Os metadados dos bens de Interface também apóiam informações sobre operações de interfaces. Assim, também adicionamos os detalhes das operações, tais como nome e tipos de parâmetros, em cada bem de Interface.

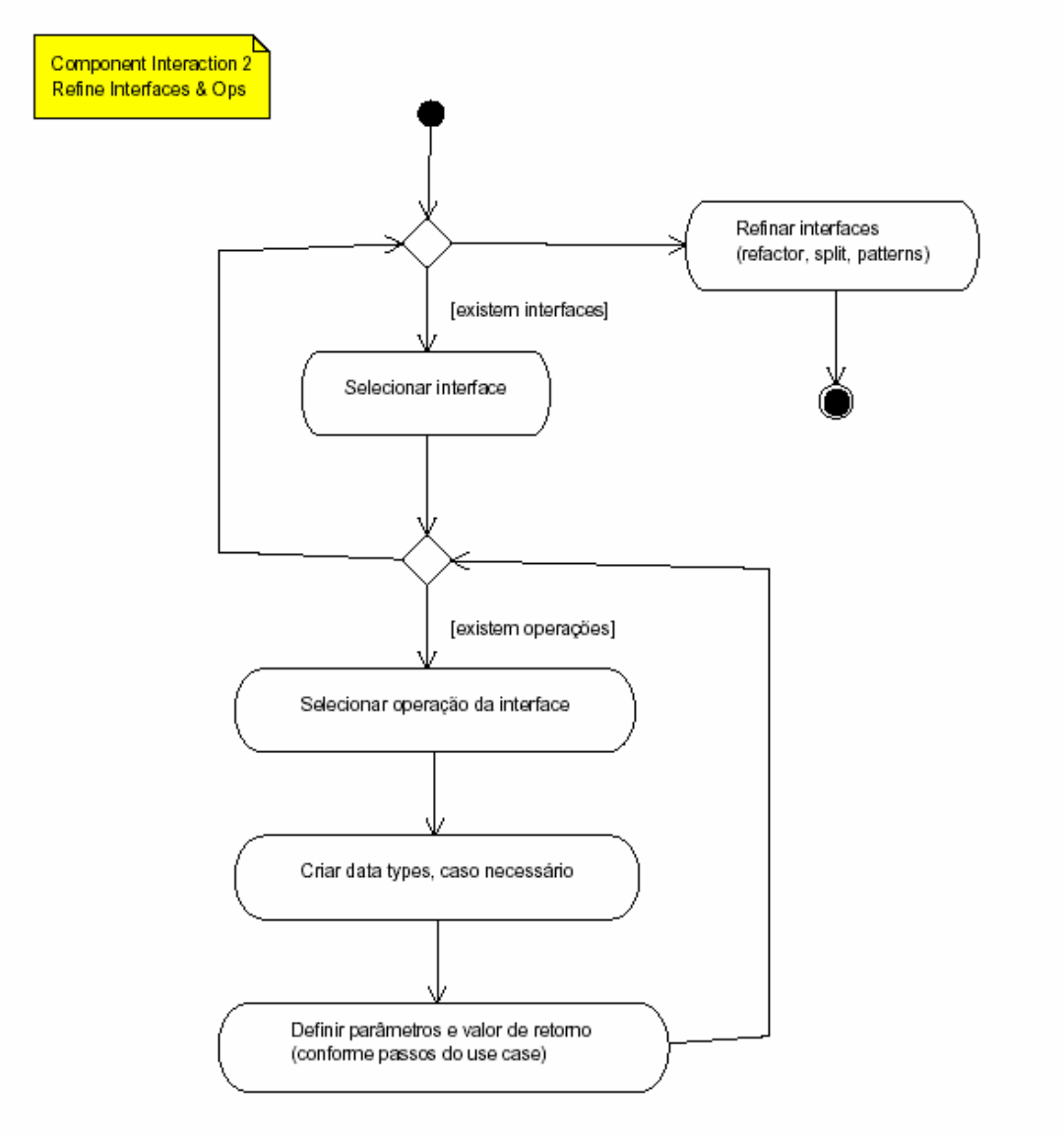

Figura 21 - Iteração de Componentes: refinar interfaces e operações

6.1.7 Iteração de Componentes: refinar especificações de componentes e arquitetura

Nesta fase fizemos o detalhamento da arquitetura dos componentes (Figura 22). Para cada bem do tipo componente abstrato criamos um bem análogo do tipo componente concreto. Estes bens relativos a componentes concretos utilizam um profile do tipo concreteComponent. Este profile oferece suporte a relacionamentos de dependência entre componentes, através da definição de interfaces requeridas.

Ao detalharmos a arquitetura do sistema, adicionamos informações sobre relacionamento de dependência entre componentes ao repositório. Para cada bem de tipo componente concreto, fizemos as ligações entre este bem e o bem relativo ao componente abstrato que implementa a interface requerida pelo componente concreto. Esta ligação é implementada pela propriedade Solution.Implementation.CompositeComponent.ComponentBasedView.InterfaceConnection[0]. SubComponent_Id, que deve conter o identificador do bem que contém o componente abstrato requerido. Como cada interface já estava associada a um componente abstrato, ao ligar o componente concreto à interface, estamos completando a associação componente concreto -> interface -> componente abstrato. Esta ligação ainda pode ser realizada através de conectores, para isto devemos considerar conectores como componentes, e adicioná-los ao repositório como bens individuais.

Nesta fase, a configuração arquitetural já possui um refinamento suficiente para ser adicionada aos metadados do repositório. Para isto, um bem do tipo Configuration é criado com o nome “SistemaHotel_configuration” . Todos os bens de componentes concretos que compõem o sistema são referenciados neste bem.

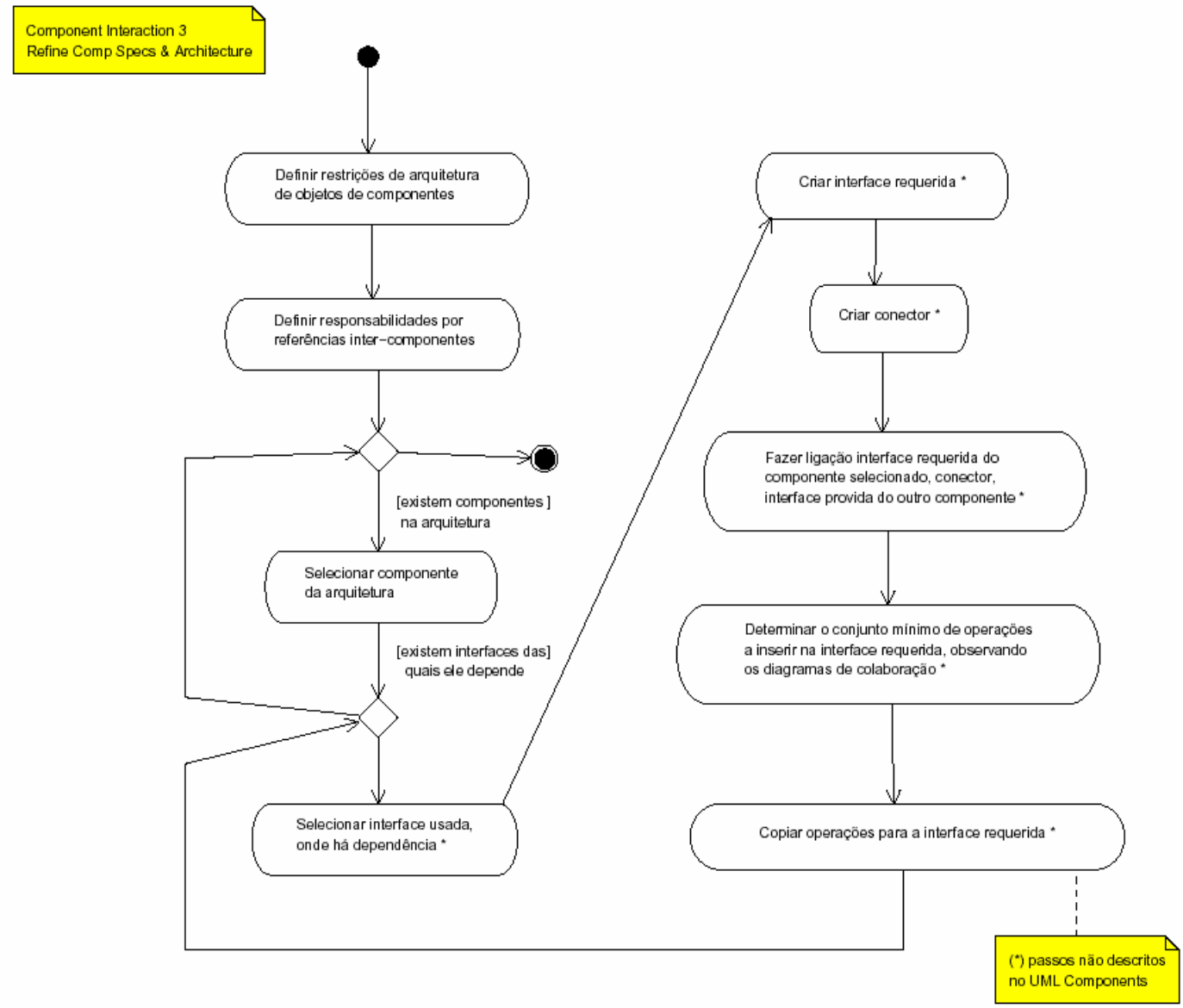

Figura 22 - Iteração de Componentes: refinar especificações de componentes e arquitetura

6.1.8 Especificação de Componentes: definir Interface Information Model, especificar pré/pós condições

A principal atividade nesta fase (Figura 23) é a criação de Interface Information Models para todas as interfaces em desenvolvimento. Interface Information Model são geralmente diagramas que são armazenados em arquivos. Uma vez prontos, estes diagramas foram adicionados ao respectivo bem de interface como artefatos.

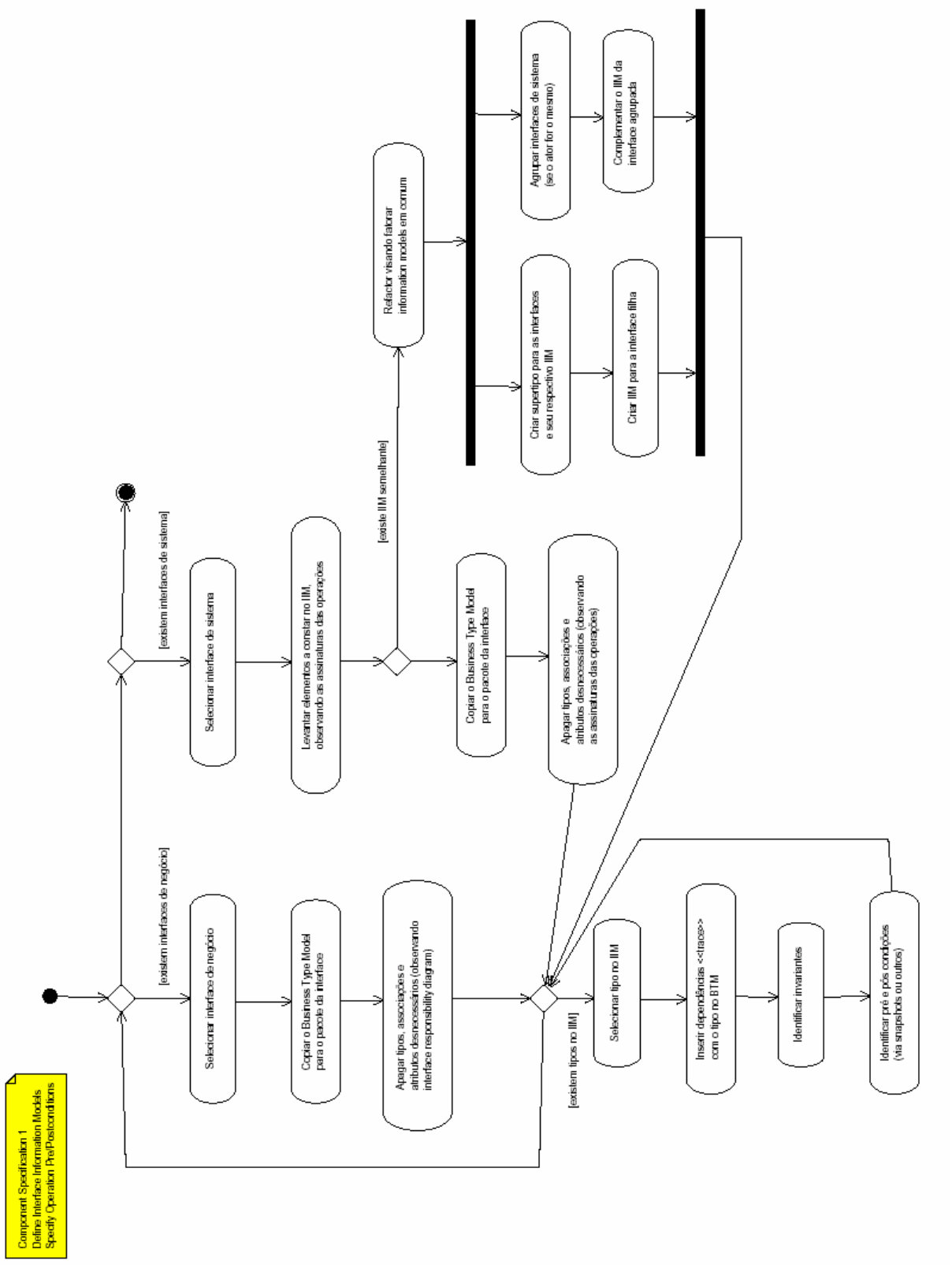

Figura 23 - Especificação de Componentes: definir Interface Information Model, especificar pré/pós condições

Ainda ocorrem nesta etapa, a definição de pré-condições, pós-condições e invariantes. A descrição destas condições foi realizada em documentos, os quais foram também adicionados ao Rigel como artefatos de bens de interfaces. Além destes documentos, também adicionamos esta informação nos meta-dados dos bens de interface, através das seguintes propriedades:

- Solution .Design.InterfaceDefinition[i].Operation[j].Condition[k].Type
- Solution .Design.InterfaceDefinition[i].Operation[j].Condition[k].Description
- Solution .Design.InterfaceDefinition[i].Operation[j].Condition[k].Expression

O fatoramento de interfaces é uma atividade que também ocorre nesta fase. Nesta operação agrupamos algumas interfaces, e como cada interface estava definida em um bem diferente, houve impacto no cadastro corrente do Rigel. Para sincronizar o dados do repositório, ao realizar um agrupamento em duas interfaces, removemos um dos bens de interface, e transferimos os meta-dados e artefatos para o segundo bem de interface.

6.1.9 Especificação de Componentes: Especificar Restrições de Componente-Interface

Nesta etapa (Figura 24) alteramos alguns documentos e diagramas de bens, a fim de adicionar restrições de componente-interface. Em relação ao repositório, apenas recuperamos os artefatos a serem alterados, realizamos as alterações e depois adicionamos os artefatos ao bem.

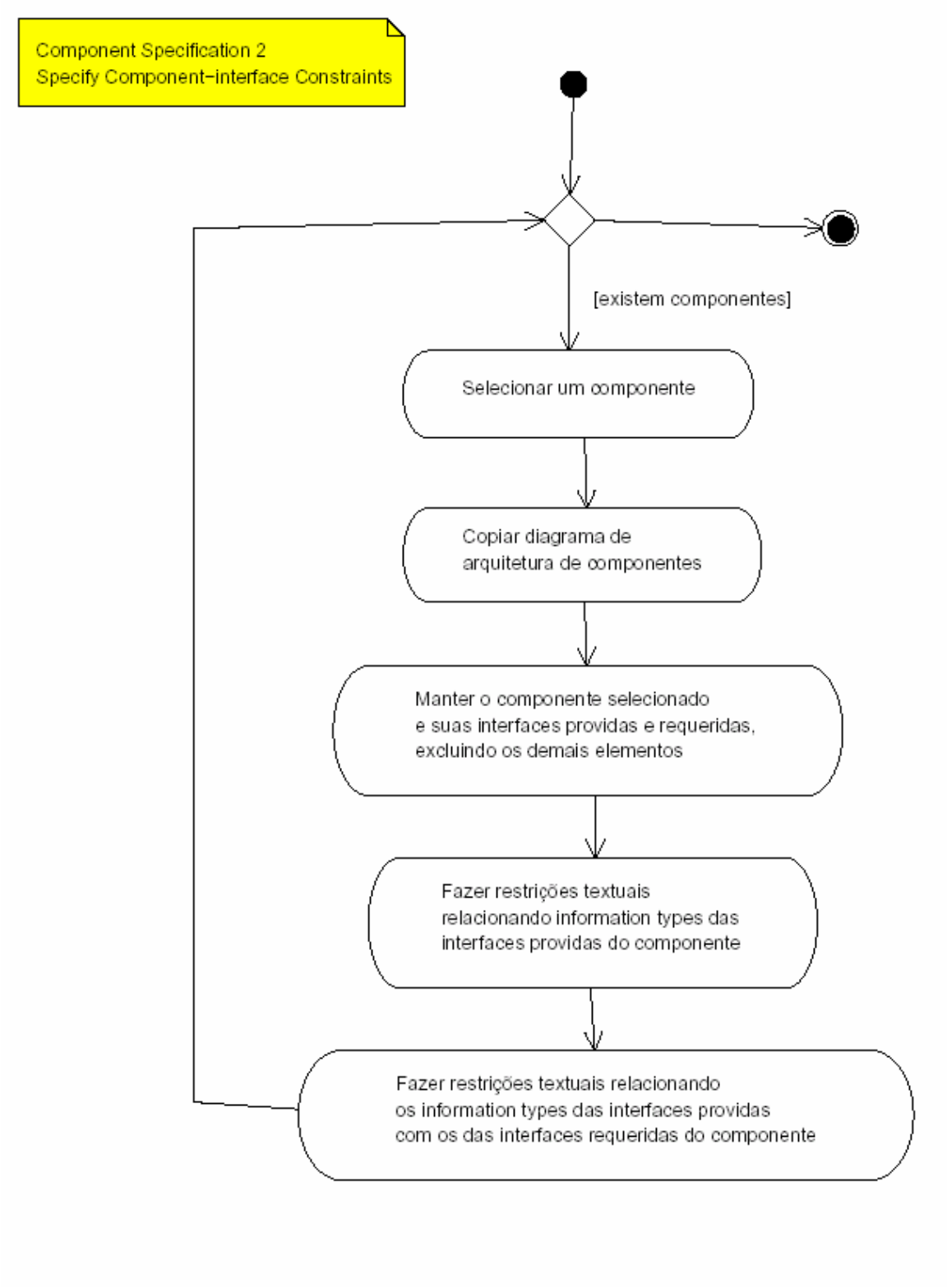

Figura 24 - Especificação de Componentes: Especificar Restrições de Componente-Interface

6.1.10 Provisionamento

A fase de Provisionamento (Figura 25) consiste em providenciar implementações para os componentes especificados previamente. Geralmente, existem dois caminhos ao provisionar um componente: pesquisar e utilizar um componente já pronto ou implementar o componente desde o início.

Para encontrar componentes já implementados, o Rigel oferece a funcionalidade de pesquisa de bens. Ao realizar a pesquisa é permitido incluir qualquer propriedade do bem nos parâmetros da pesquisa. Para atender a atividade de provisionamento, utilizamos alguns parâmetros que visam descartar bens que não atendem os objetivos desta etapa. São exemplos:

- State – estado de desenvolvimento do bem. Para provisionamento, consideramos apenas, bens no estado "published".
- Description – descrição detalhada do bem. Para este parâmetro incluído algumas palavras-chave relacionadas ao componente que pretendíamos encontrar.
- Classification.DescriptorGroup[*].DescriptorName[*] e Classification.DescriptorGroup[*].DescriptorValue[*] – estas propriedades estão relacionadas com a classificação de um bem. A pesquisa por classificação também é muito importante em um repositório. Alguns valores utilizados: DescriptorName/Value=Platform/Java; Domain/Entertainment; Type/Infrastructure, etc.
- Solution.Implementation.ElementaryComponent.Interface[*].RelatedAsset[*] – esta propriedade permite encontrar todos os componentes concretos que implementar uma determinada interface.

A outra opção para provisionar componentes é a implementação desde o início. Para estes casos, o desenvolvedor irá criar vários artefatos que compõem a implementação e posteriormente estes artefatos deverão ser adicionados ao repositório.

O bem "SistemaHotel_configuration" é atualizado para referenciar os novos componentes concretos que fazem parte da configuração arquitetural concreta do sistema.

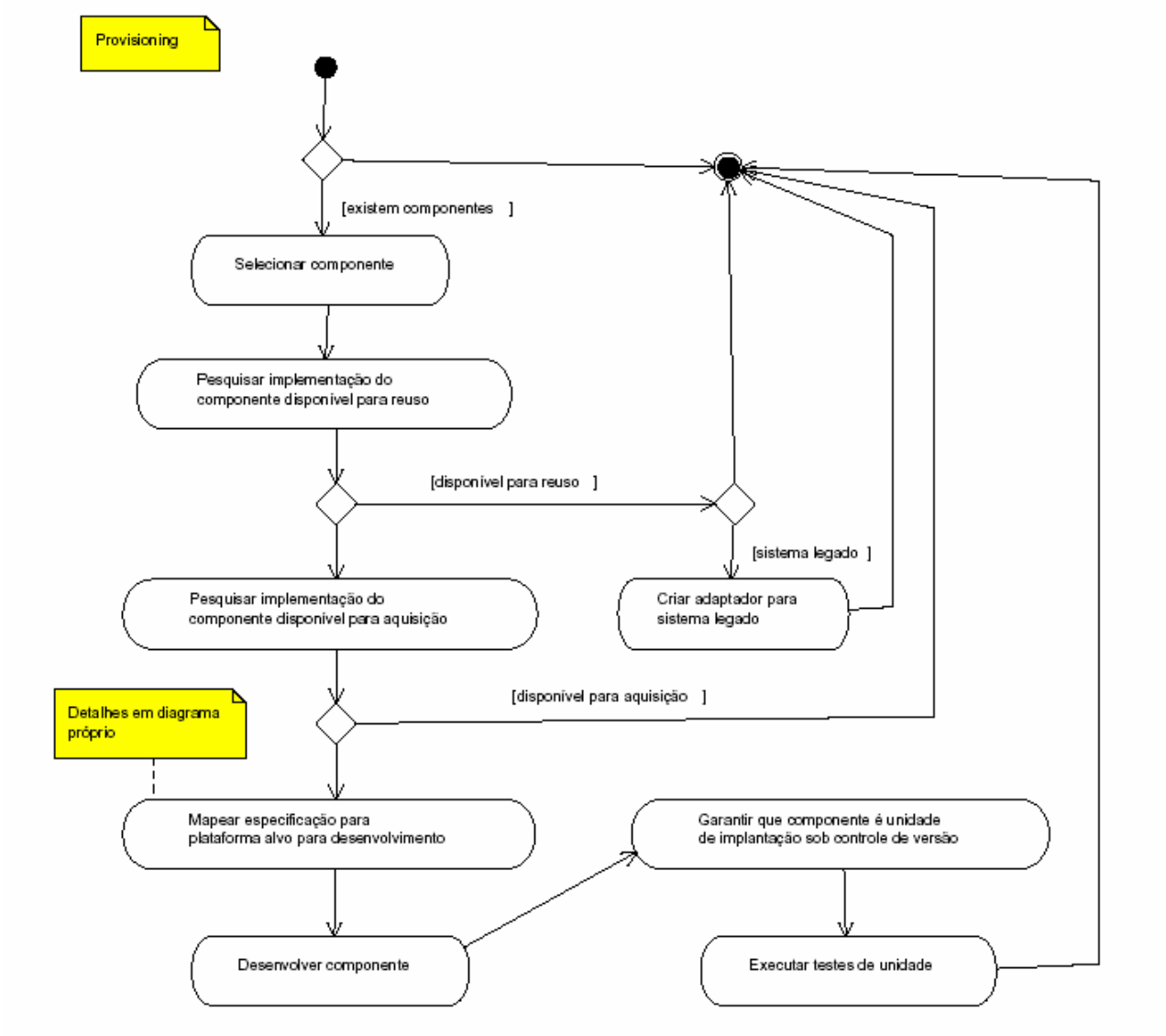

Figura 25 - Provisionamento

6.1.11 Montagem

Esta é a última fase no processo UML Components (Figura 26). Nesta fase utilizamos o Rigel para recuperar todos os componentes provisionados, para em seguida montar o sistema. O bem "SistemaHotel_configuration" é recuperado, seus metadados são utilizados para indicar quais componentes devem ser recuperados para realizar a montagem do sistema.

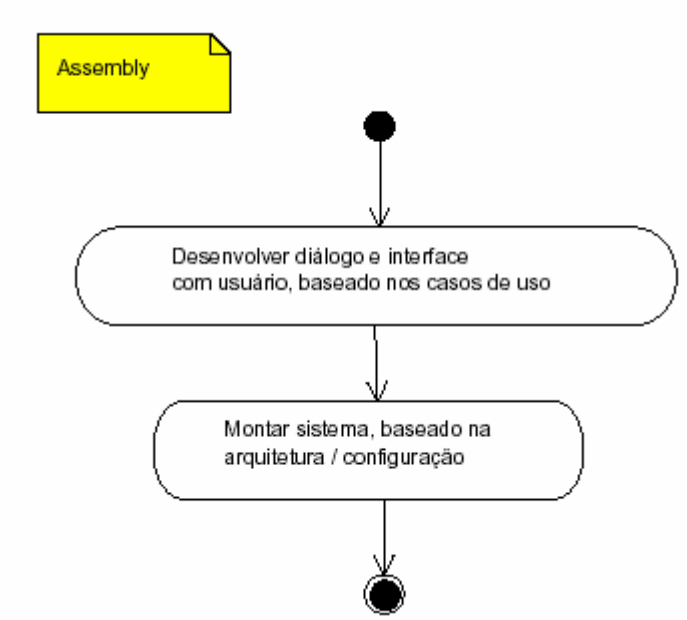

Figura 26 – Montagem

6.1.12 Resultados

Neste estudo de caso, todas as fases do processo foram exercitadas e, em todas elas, as funcionalidades do Rigel foram exercitadas. Armazenar, recuperar e pesquisar componentes são funcionalidades presentes no Rigel que são importantes para um desenvolvimento baseado em componentes eficiente.

Todos os artefatos gerados durante o desenvolvimento do sistema de reserva de hotel foram armazenados no repositório Rigel. Os metadados do repositório foram suficientes para armazenar conceitos relativos a componentes, tais como: parâmetros e operações de interfaces e separação entre especificação e implementação.

Também foi possível verificar que a pesquisa de componentes apóia o processo UML Components principalmente através da análise da fase de provisionamento. A pesquisa de componentes encontrou todos os bens criados durante o processo de desenvolvimento. As propriedades intrínsecas de componentes foram consideradas pela pesquisa. A navegação entre bens relacionados é mais uma característica importante provida pela pesquisa de bens, que permite, por exemplo, encontrar implementações para especificações de componentes.

Este estudo de caso mostra que o Rigel apóia todas as fases do processo de UML Components, possibilitando um desenvolvimento baseado em componentes mais eficiente.

6.2 Estudo de Caso 2: Armazenar Componentes Prontos

Componentes estáveis e difundidos na comunidade de desenvolvedores de sistemas baseados em componentes são boas opções para realizar testes e análises de ferramentas para DBC. No caso deste trabalho, selecionamos alguns componentes reutilizáveis do projeto Apache Jakarta para verificar o uso deste repositório.

Os componentes selecionados foram:

- Apache Jakarta Commons Logging – um componente que oferece algumas implementações de *logging* de aplicações.
- Apache Jakarta Commons IO – um componente com funcionalidades adicionais de entrada e saída de dados.
- Apache Jakarta Commons FileUpload – um componente que auxilia a implementação de operações de envio de arquivos a servidores Web a partir de clientes Web.

O processo de adição de componentes no repositório é composto de duas etapas: o primeiro passo é do cadastro apenas os metadados dos componentes utilizando a interface Web. Posteriormente seus artefatos são carregados utilizando a mesma interface Web.

Jakarta Commons Logging

Inicialmente foi cadastrado o componente Jakarta Commons Logging, devido ao fato deste componente não ter dependências de outros componentes. Este componente apresenta uma estrutura relativamente simples:

- Uma especificação composta de descrição do componente, documentação em geral sobre o componente e referências sobre as interfaces associadas a especificação.
- Um conjunto de interfaces associadas à especificação.

- Uma implementação que satisfaz a especificação.

De acordo com o modelo de metadados para componentes introduzido neste trabalho, temos os tipos de bem:

- Componente abstrato
- Componente concreto
- Interface de componente
- Configuração de componente

O componente foi divido em mais de um bem ao ser cadastrado no repositório a fim de ficar compatível com o modelo de metadados do Rigel. Assim, as informações sobre o cadastro deste componente ficaram organizadas da seguinte maneira:

- Um bem representando o componente abstrato: Jakarta.Commons.Logging_abstractComponent"
- Um bem representando a interface do componente: "Jakarta.Commons.Logging_Interface"
- Um bem representando a implementação do componente: "Jakarta.Commons.Logging_concreteComponent"

O componente abstrato é o bem principal que contém relacionamentos que permitem a navegação aos outros dois bens que representam este componente. No bem relativo ao componente abstrato adicionamos as informações relativas à especificação do componente. No caso do Jakarta Commons Loggins temos alguns artefatos contendo documentação sobre o componente tais como descrições do componente, manual de usuário e referências a documentos e páginas da Internet relativos a especificação do componente. Informações específicas da interface do componente são armazenadas em um bem separadamente do bem referente ao componente abstrato.

O bem "Jakarta.Commons.Logging_abstractComponent" contém alguns atributos que permitem a navegação entre este e o bem "Jakarta.Commons.Logging_Interface" (Figura 27). A hierarquia de parâmetros Design.AbstractComponent.ExternalProperty.Interface é responsável por conter tais relacionamentos com o bem da interface do componente. O elemento "Interface"

contém atributos que identificam o nome da interface requerida. O elemento “RelatedAsset” contém atributos que identificam o bem que contém a interface.

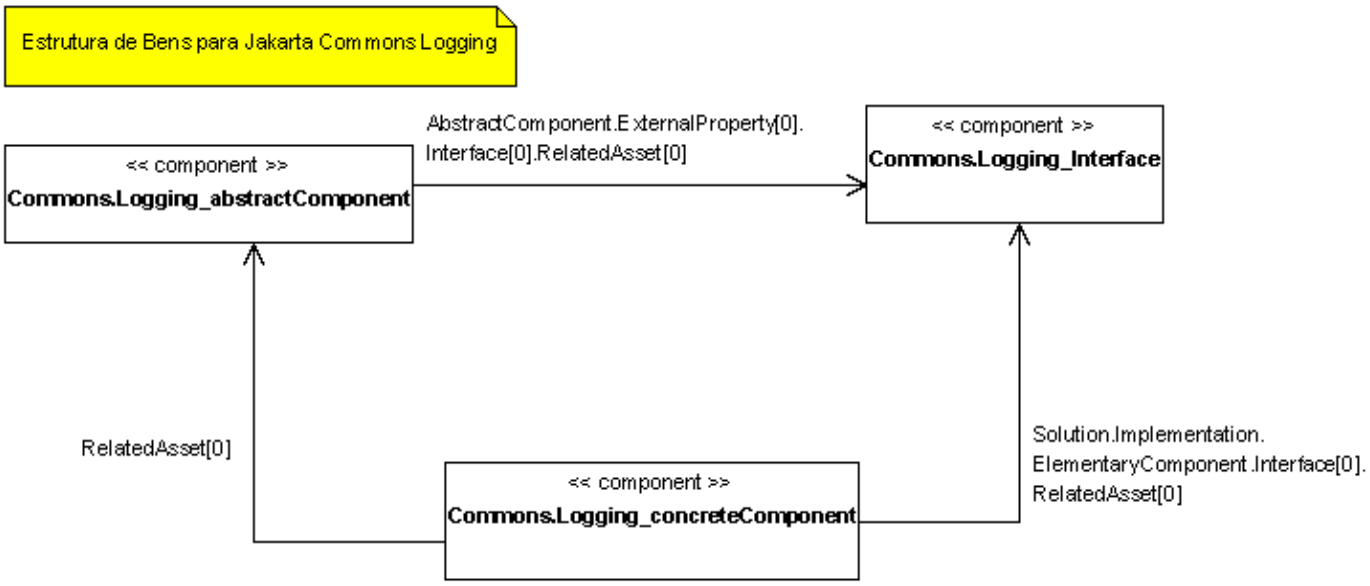

Figura 27 – Estrutura de Bens para Jakarta Commons Logging

A classificação do componente é importante para possibilitar pesquisas mais eficientes. No caso do Jakarta Commons Logging, alguns descritores foram adicionados para caracterizar o componente, tais como: Plataforma/Java; Tipo/Infra-estrutura; Funcionalidade/Logging; Domínio/Geral.

Os componentes acima são listados nos resultados de pesquisa de bens por palavras chaves, tais como pesquisas em descrição de componentes. Exemplo:

Atributo: Asset_Description Valor: “Logging”

Pesquisas por classificações de componentes também relacionam os componentes acima, são exemplos as seguintes:

Atributo: Asset.Classification.DescriptorGroup.Descriptor_Name Valor:” Platform”

Atributo: Asset.Classification.DescriptorGroup.Descriptor_Value Valor: “Java”

Jakarta Commons FileUpload e Jakarta Commons IO

Apache Jakarta Commons FileUpload é um componente que auxilia a implementação de operações de envio de arquivos a servidores Web a partir de clientes Web. Este componente apresenta uma dependência do Apache Jakarta Commons IO – um componente com funcionalidades adicionais de entrada e saída de dados.

Ao cadastrar estes componentes no repositório, novamente seguimos o modelo de meta-dados do Rigel e dividimos estes componentes em 6 bens. Assim para o FileUpload temos:

- Jakarta.Commons.FileUpload_abstractComponent – o bem que representa o componente abstrato.
- Jakarta.Commons.FileUpload_Interface – o bem que representa as interfaces de FileUpload.
- Jakarta.Commons.FileUpload_concreteComponent – o bem que contém a implementação do FileUpload.

E para o Jakarta Commons IO temos:

- commonsIO_abstractComponent – o bem que contém a especificação do componente.
- commonsIO_concreteComponent – contendo a implementação do componente.
- fileFilterInterface – o bem representa a interface oferecida por commonsIO.

Assim como o Jakarta Commons Logging, as ligações entre os bens acontecem da seguinte maneira: AbstractComponent.ExternalProperty.Interface.RelatedAsset liga o componente abstrato ao bem relativo às interfaces do componente. O componente concreto

referencia o componente abstrato através do RelatedAsset, e a interface através de Solution.Implementation.ElementaryComponent.Interface.RelatedAsset.

A ligação entre os componentes Commons FileUpload e Commons IO acontece através do atributo Solution.Implementation.CompositeComponent.ComponentBasedView.InterfaceConnection.SubComponent que interliga o bem Jakarta.Commons.FileUpload_concreteComponent ao bem commonsIO_abstractComponent, como pode ser observado na Figura 28 .

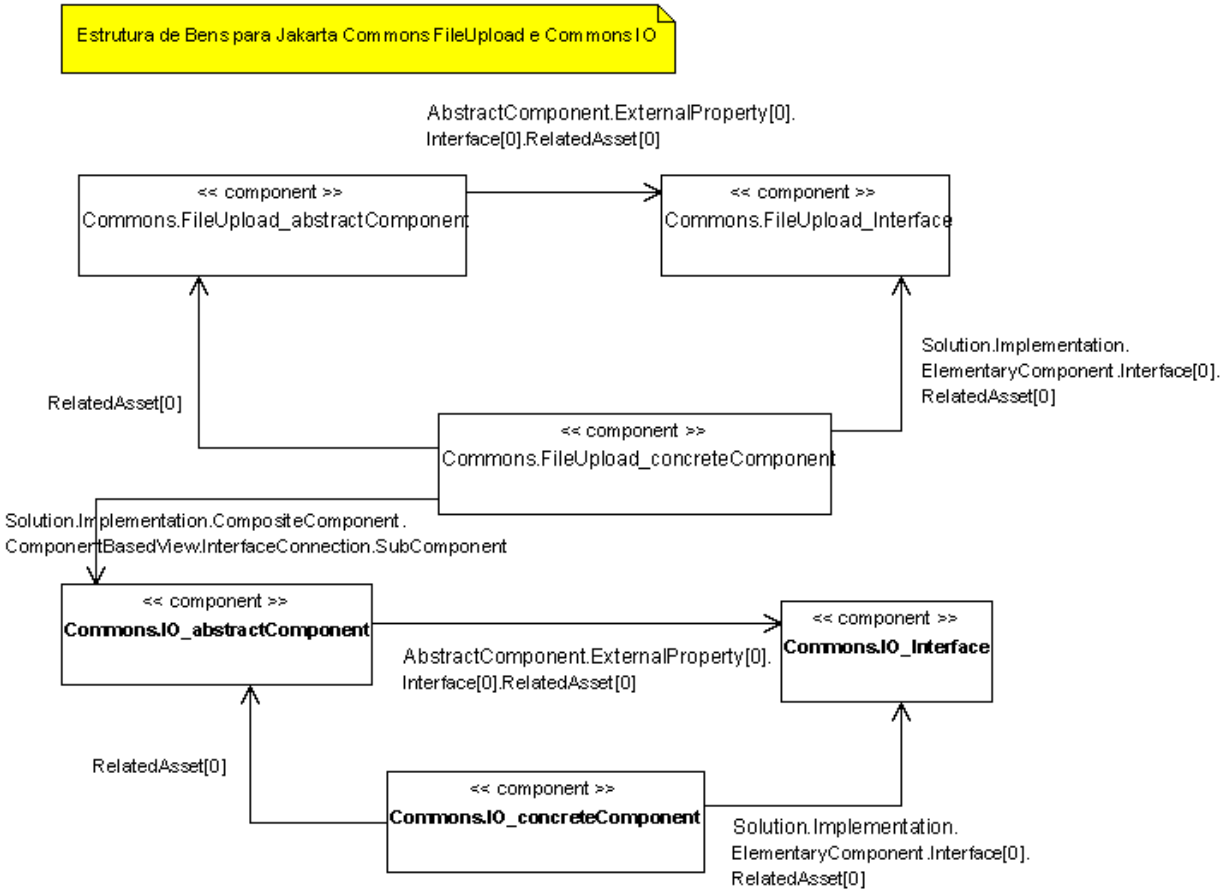

Figura 28 – Estrutura de bens para Jakarta Commons FileUpload e Commons IO

A classificação do Jakarta Commons IO utilizou os seguintes descritores:: Plataforma/Java; Tipo/Infra-estrutura; Funcionalidade/IO; Domínio/Geral. Para o Jakarta FileUpload temos Plataforma/Java; Tipo/Infra-estrutura; Funcionalidade/WebServices; Domínio/Geral.

Além das pesquisas mencionadas anteriormente quando descrevemos os bens de Commons Logging, executamos algumas pesquisas com foco no relacionamento entre componentes.

- Todos bens (componentes concretos) que utilizam o componente de Commons.FileUpload:

Atributo: Solution. Implementation. CompositeComponent. ComponentBasedView. InterfaceConnection[0]. SubComponent_Id AssetId

Valor: <assetId de Commons.FileUpload Interface>

Resultado: apenas bens que utilizam o componente CommonsFileUpload foram retornados.

- Todos os bens (componentes concretos) que implementam as interfaces de FileUpload

Atributo: Solution. Implementation. ElementaryComponent. Interface[0]. RelatedAsset[0]. AssetId

Valor: <assetId de Commons.FileUpload Interface>

Resultado: apenas bens que implementam a interface de FileUpload foram retornados.

- Todos os bens (componentes abstratos) que provêem as interfaces de FileUpload

Atributo: AbstractComponent.ExternalProperty[0].Interface[0].RelatedAsset[0].AssetId

Valor: <assetId de Commons.FileUpload Interface>

Resultado: apenas os componentes abstratos que provêem as interfaces de FileUpload foram retornados.

6.2.1 Desempenho

Realizamos testes de desempenho nas operações de pesquisa do repositório Rigel. O objetivo deste teste é verificar se o requisito de desempenho, descrito na seção 3.2.5, está sendo atendido. Este requisito especifica que o Rigel deverá ter um desempenho em pesquisas compatível com outros repositórios.

A fim de termos parâmetros para realizar esta comparação selecionamos dois repositórios: ComponentSource [CSource] e Spars [Inoue03]. O critério de seleção utilizado foi a disponibilidade para teste e a flexibilidade nas pesquisas de bens.

O quadro abaixo (Tabela 3) mostra uma comparação do tempo de resposta de pesquisas entre estes repositórios.

	Rigel Tempo (s)	ComponentSource Tempo (s)	Spars Tempo (s)
Pesquisa sem resultados	2	5	2
Pesquisa com 1 resultado	3	3	5
Pesquisa com 5 resultados	3	4	14
Pesquisa com 10 resultados	5	6	10

Tabela 3 – comparação do tempo de resposta para pesquisas (em segundos).

6.2.2 Resultados

Neste estudo de caso foi possível constatar o suporte do Rigel para componentes em geral. Os componentes selecionados foram armazenados no repositório com sucesso. As propriedades particulares de componentes, tais como: interfaces providas e requeridas; e separação entre especificação e implementação foram armazenadas nos metadados.

Os metadados do repositório também facilitam a pesquisa de componentes. Foi possível realizar diversos tipos de pesquisas com alta precisão e recuperação (*recall*), conceitos discutidos na seção 3.1.3.

O desempenho das pesquisas do Rigel mostrou-se satisfatório. Os testes realizados indicaram que o repositório Rigel obteve um desempenho similar à outras ferramentas disponíveis.

6.3 Resumo

Este capítulo apresenta dois estudos de caso do Repositório Rigel. O primeiro estudo de caso faz uma análise de como o repositório Rigel se comporta ao ser utilizado durante o desenvolvimento de sistemas utilizando o processo UML Components. Todas as etapas do processo são consideradas. Cada etapa é tratada em uma seção separada que discute o suporte do repositório Rigel.

O segundo estudo de caso explora a utilização do Rigel com componentes prontos. Neste cenário, três componentes difundidos na Web são divididos em bens e armazenados no Rigel. Exemplos de relacionamentos e valores de propriedade de bens são mostrados.

Capítulo 7 Conclusões e Trabalhos Futuros

Este trabalho apresentou o Rigel, um repositório de bens de software reutilizáveis com suporte a um processo de desenvolvimento baseado em componentes. O Rigel apresenta características que facilitam atividades executadas durante o desenvolvimento de sistemas em componentes. Em particular, um estudo de caso apresentado no Capítulo 6, seção 6.1, mostra como as funcionalidades do Rigel se encaixam nas atividades executadas em cada fase do processo UML Components, um processo de desenvolvimento baseado em componentes apresentado no Capítulo 2, seção 2.1.2.

Outro ponto importante do Rigel é o seu modelo de metadados. A arquitetura interna do repositório permite que o modelo de metadados seja estendido de acordo com as necessidades dos usuários. Inicialmente, estendemos este modelo para bens relacionados a componentes: componente abstrato, componente concreto, interface e configuração, conforme descrito na seção 4.1.7. Este modelo de metadados permite uma maior eficiência em pesquisa de bens, maior transparência na utilização do repositório e mais flexibilidade na reutilização de componentes. Um estudo de caso (seção 6.2) exercitou exemplos de utilização destes metadados, mostrando principalmente como acontecem os relacionamentos entre bens, como as principais propriedades dos componentes são armazenadas e como estes bens podem ser encontrados por operações de pesquisa.

Interoperabilidade é um importante requisito tratado neste trabalho, conforme discutido na seção 3.2.1. Neste trabalho, criamos uma arquitetura para o Rigel com suporte à interoperabilidade. A interface da camada de sistema do Rigel foi projetada para ser acoplada a diferentes sistemas. Principalmente, consideramos o cenário de integração do Rigel com outros repositórios e com ferramentas de desenvolvimento de sistemas (IDE).

O formato em que os bens são empacotados também está relacionado à interoperabilidade. Para haver a troca de bens entre ferramentas, metadados e artefatos de um bem devem possuir um formato que possibilite a sua interpretação por outras ferramentas. A utilização do padrão RAS como o formato de empacotamento de bens, é um passo importante em direção a integração do Rigel com outros sistemas.

7.1 Contribuições

As maiores contribuições deste trabalho foram o projeto e implementação do repositório Rigel. As características mais inovadoras do Rigel são:

- **Novos *profiles* RAS para componentes.** Criamos novos *profiles* RAS apoiando um modelo conceitual de componentes e arquitetura de software. Estes profiles estão baseados no modelo conceitual discutido em [Guerra05], e mostram uma maneira de estender o padrão RAS para um suporte mais sofisticado a componentes, considerando requisitos tais como maior flexibilidade ao reutilizar bens, propriedades relevantes a componentes e representação da arquitetura de sistemas.
- **Uma arquitetura de repositórios com suporte à interoperabilidade.** A arquitetura do Rigel considera interoperabilidade de duas formas: (i) a camada de sistema do Rigel foi projetada para se integrar a outros sistemas; (ii) o formato de componentes segue o padrão RAS, a fim de diminuir os esforços na conversão entre bens provenientes de diferentes sistemas.
- **Uma solução para a implementação de pesquisas de bens**. Muitas vezes os metadados de um repositório serão definidos de uma forma semi-estruturada. O padrão RAS define que os metadados deverão estar armazenados em XML. Neste trabalho implementamos uma solução para indexação e pesquisa de metadados de bens armazenados em arquivos XML.
- **Uma solução para a integração entre repositórios e o CVS.** O Rigel se integra ao CVS para armazenar artefatos. Esta integração mostra uma maneira interessante de agrupar artefatos e prover mais significado aos artefatos armazenados no CVS.

Este trabalho resultou na co-autoria da seguinte publicação:

Um Repositório de Componentes com Suporte à Evolução centrada na Arquitetura de Software

Autores: Leonardo P. Tizzei, Helder S. Pinho, Paulo A. C. Guerra, Cecília M. F. Rubira

Conferência: 5º Workshop de Desenvolvimento Baseado em Componentes – WDBC 2005, Juiz de Fora, Minas Gerais, Brazil, Novembro 2005.

7.2 Trabalhos Futuros

Podemos propor algumas linhas de pesquisa como extensões e melhorias do trabalho apresentados:

Os estudos de caso apresentados consideraram componentes prontos encontrados na Web ou a simulação do desenvolvimento de um sistema utilizando o processo UML Components, conforme descrito em [Cheesman+00]. A utlização do Rigel durante o desenvolvimento de um sistema real utilizando um processo DBC e envolvendo múltiplos desenvolvedores em uma organização real é um sugestão para um novo estudo de caso.

O suporte do Rigel para evolução de componentes ainda é primário. Atualmente a integração com o CVS provê apenas o versionamento simples de artefatos. Para um suporte completo à evolução de componentes é necessário a implementação de funcionalidades que auxiliem o desenvolvedor durante as alterações de componentes. Podemos citar como sugestões iniciais a extensão do repositório para validação de regras associadas à evolução [Lobo05], a extensão dos metadados para apoiar propriedades relativas a evolução e a melhoria da integração Rigel-CVS a fim de apoiar operações com ramos (seção 2.2.2).

Uma outra necessidade é a implementação da integração do Rigel com IDEs e outros repositórios. No estado atual do trabalho, o Rigel possui uma arquitetura que provê facilidades na integração com outros sistemas, mas nenhuma integração concreta foi executada.

Apesar das vantagens do protocolo Java RMI – utilizado nas interfaces do Rigel, este tem a desvantagem de ser específico para plataformas Java. Uma estudo comparativo detalhado de possíveis protocolos para integração do Rigel como web-services, CORBA, seguido de posterior implementação no Rigel são sugestões para melhorar o suporte a interoperabilidade.

A pesquisa de bens possibilita a especificação de critérios de pesquisa para quaisquer propriedades do bem. Todas as propriedades dos metadados estão disponíveis para pesquisa. Entretanto, não é possível realizar a pesquisa por conteúdo de artefatos. Uma oportunidade de melhorar a pesquisa seria através da adaptação da máquina de busca para indexar e pesquisar conteúdo de artefatos como documentos, código fonte e diagramas.

Segurança é um requisito importante para repositórios, mas não foi considerado neste trabalho. Um controle de acesso ao repositório Rigel é mais uma sugestão para uma evolução do Rigel.

Referências Bibliográficas

[Ambriola97] Vincenzo AMBRIOLA, Reidar CONRADI, Alfonso FUGGETA, Assessing Process-Centered Software Engineering Environments. In: ACM Transactions on Software Engineering and Methodology, v.6, n.3, pp.283-328, Julho 1997.

[Bass03] L. Bass, P. Clements, and R. Kazman. (2003). Software Architecture in Practice. SEI Series in Software Engineering. Addison-Wesley, 2nd edition.

[Bishop96] J. Bishop and R. Faria, "Connectors in Configuration Programming Languages: are They Necessary?", IEEE Third International Conference on Configurable Distributed Systems, Annapolis, Maio, 1996

[Boldyreff02] C. Boldyreff; D. Nutter; S. Rank. Open-Source Artefact Management. In: Workshop Open Source Software Engineering, Orlando, Florida, USA, 2002.

[Boldyreff03] C. Boldyreff; D. Nutter; S. Rank. An Artefact Repository to Support Distributed Software Engineering. In: Workshop On Cooperative Supports For Distributed Software Engineering Processes (CSSE'2003). Benevento, Italy. 2003.

[Bronsard+97] Francois Bronsard, Douglas Bryan, W. (Voytek) Kozaczynski, Edy S. Liongosari, Jim Q. Ning, Asgeir Olafsson, and John W. Wetterstrand. Toward software plug-and-play. In ACM Symposium on Software Reusability (SSR'97), pages 19 to 29, 1997.

[Brian01] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, Moshe Shadmon. A Fast Index for Semistructured Data. Proceedings of the 27th VLDB Conference, 2001

[Cheesman+00] John Cheesman and John Daniels. UML Components. Addison-Wesley, 2000.

[Chen02] Feng Chen, Qianxiang Wang, Hong Mei, and Fuqing Yang. An architecture-based approach for component-oriented development. 26th Annual International Computer Software and Applications Conference,August 2002.

[Christensen99] Henrik Bærbak Christensen. The Ragnarok Architectural Software Configuration Management Model. In Proceedings of the 32nd Hawaii International Conference on System Science (HICSS) , ACM, Page: 8067, 1999.

[Clements01] P. Clements, L. M. Northrop, 2001, Software Product Lines: Practices and Patterns. 1 ed., Boston, MA, Addison-Wesley.

[Clements03] Paul Clements and Rick Kazman. Software Architecture in Practices. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[CSource] ComponentSource. www.componentsource.com em março, 2005.

[CVS] Concurrent Versions System. https://www.cvshome.org/, em novembro/2004

[D'Souza99] D. D'Souza, A. C. Willis. Objects, Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, 1999.

[Eclipse] Eclipse Consortium. Eclipse.org main page. http://www.eclipse.org/, 2003.

[Estublier94] J. Estublier and R. Casallas. Configuration Management - The Adele Software Configuration Manager, pages 99–139. Trends in Software. J. Wiley and Sons, Baffins Lane, Chichester West Sussex, PO19 1UD, England, 1994.

[Estublier00] Jacky Estublier. Software configuration management: a roadmap. In Proceedings of the Conference on The Future of Software Engineering, pages 279-289, 2000.

[Flashline] Flashline 4, http://www.flashline.com/fcm/fcm.jsp, em fevereiro, 2005.

[Garlan00] Garlan Garlan, Robert T. MONROE, , David WILE. Acme: Architectural Description of Component-Based Systems. In: LEAVENS, G.T., SITARAMAN, M. (Eds.) Foundations of Component Based Systems. Cambridge University Press, Cambridge, UK. 2000. Cap. 3, pp.47-67.

[Gibson00] David S. Gibson, Bruce W. Weide, Scott M. Pike, Stephen H. Edwards. Toward a normative theory for component-based system design and analysis. In: LEAVENS, G.T., SITARAMAN, M. (Eds.) Foundations of Component Based Systems. Cambridge University Press, Cambridge, UK. 2000. Cap. 10, pp.211-230.

[Guerra05] Paulo A. C. Guerra, Tiago C. Moronte, Rodrigo T. Tomita, Cecília M. F. Rubira. Um Modelo Conceitual e uma Terminologia para o Desenvolvimento Baseado em Componentes e Centrado na Arquitetura de Software. Em 5o. Workshop de Desenvolvimento Baseado em Componentes, Brazil, 2005.

[Hatcher04] E. Hatcher and O. Gospodnetic. Lucene in Action (In Action series). Manning Publications, 2004.

[Henninger94] S. Henninger. Using Iterative Refinement to Find Reusable Software, IEEE Software, 11(5), 1994. Henninger, S. Supporting the construction and evolution of component repositories.

[HMR+01] André van der Hoek, Marija Mikic-Rakic, Roshanak Roshandel, and Neno Medvidovic. Taming Architectural Evolution, Proceedings of the Sixth European Software Engineering Conference (ESEC) and the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), Vienna, Austria, September 2001.

[Inoue03] K. Inoue, R. Yokomori, H. Fujiwara; T. Yamamoto; M. Matsushita; S. Kusumoto. Component Rank: Relative Significance Rank for Software Component Search. In ICSE, pages 14–24, Portland, OR, 2003.

[ISO2005] ISO 9126. [http://www.iso.org] em Julho, 2005.

[Jacobson+99] I. Jacobson, J. Rumbaugh e G. Booch Unified Software Development Process – Addison Wesley, Reading – MA, 1999.

[JAXB] Sun Java XML Binding (JAXB). http://java.sun.com/webservices/jaxb/index.jsp acessado em janeiro/2006.

[JCVS] jCVS home page. http://www.jcvs.org/, em novembro/2004

[JSP] Java Server Pages, http://java.sun.com/products/jsp/, acessado em janeiro/2006.

[KENT55] A. Kent, M. Berry, F.U. Leuhrs, and J.W. Perry. Machine literature learning VIII. Operationalcriteria for designing information retrieval systems.American Documentation, USA, v.6, n.2, p.93-101,1955.

[Larsson99] M. Larsson. The Different Aspects of Component Based Systems. ABB Automation Products, Sweden, 1999

[Leon00] A. Leon, 2000, A Guide to Software Configuration Management, Norwood, MA, Artech House Publishers.

[Lobo05] Lobo, A., Guerra, P., Rubira, C., and Castor, F. (2005). A systematic approach for the evolution of reusable software components. Workshop on Architecture-Centric Evolution.

[Logidex] LogicLibrary Logidex, http://www.logiclibrary.com/logidex.htm, em fevereiro, 2005.

[Lucene] Lucene. http://lucene.apache.org, acessado em janeiro, 2006.

[Luer01] Chris Lüer, David S Rosenblum. WREN – An Environment for Component-Based Development. Em proceedings of the Joint 8th European Software Engineering Conference (ESEC), published as softeare. Austria. 2001.

[Luckham00] D. C. Luckham, J. Vera, and S. Meldal (2000). Key concepts in architecture definition languages. In Leavens, G. T. and Sitaraman, M., editors, Foundations of Component-Based Systems, pages 23–45. Cambridge University Press, Cambridge, UK.

[Lucredio04] D. Lucredio, A. F. Prado, E. S. de Almeida. A survey on software components search and retrieval. Em Euromicro Conference, 2004. Proceedings. 30th . On page(s): 152- 159, Sept 2004.

[Luqi99] Luqi, Jiang Guo, "Toward Automated Retrieval for a Software Component Repository", IEEE Conference and Workshop on Engineering of Computer-Based Systems, 1999.

[Mcllroy69] M. D. Mcllroy. Mass-produced software components. In P. Naur and B. Randall, editors, Software Engineering: Report on a conference by the NATO Science Committee, pages 138 150 NATO Scientific Affair Division, 1968.

[Medvidovic+00] N. Medvidovic, R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Transaction on Software Engineering, Vol. 26, No. 1, January, 2000

[Mili98] A. Mili , R. Mili , R. T. Mittermeir, A survey of software reuse libraries, Annals of Software Engineering, 5, p.349-414, 1998.

[Monroe97] Robert T. Monroe, Andrew Kompanek, Ralph Melton, and David Garlan. Architectural styles, design patterns, and objects. IEEE Softw., 14(1):43–52, 1997.

[Moore91] J. M. MOORE, S. C. BAILIN, 1991, "Domain Analysis: Framework for reuse", IEEE Computer Society Press Tutorial, pp. 179-202.

[Murta04] L. G. P. MURTA, Odyssey-SCM: Uma Abordagem de Gerência de Configuração de Software para o Desenvolvimento Baseado em Componentes. Exame de Qualificação, COPPE/UFRJ, Rio de Janeiro, RJ, Brazil, 2004

[NetBeans] Projeto NetBeans. http://www.netbeans.org, acessado em janeiro, 2006.

[Omondo] Omondo. EclipseUML. http://www.omondo.com, 2003.

[PrietoDiaz87] R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability" IEEE Software, vol.4, no. 1, pp 6-16, 1987.

[Rational] IBM Rational. Rational rose. http://www.rational.com/products/rose/index.jsp, 2003.

[RAS04] Reusable Asset Specification. OMG Final Adopted Specification pct/04-06-06. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#RAS. acessado em março, 2005.

[RMI] Java Rmi. http://java.sun.com/products/jdk/rmi/ . Acessado em Janeiro, 2006.

[Roshandel04] R. Roshandel et al. (2004). Mae - a system model and environment for managing architectural evolution. ACM TOSEM, 11(2):240 – 276.

[Sametinger97] J. Sametinger, Software Engineering with Reusable Components, Springer, 1997.

[Schuenck05] Michael Schuenck, Carla Tanure, José Jorge L. Dias Jr.,Sindolfo Miranda, Yuri Negócio, Glêdson Elias. Análise de Repositórios de Componentes para a Identificação das Soluções mais Adotadas. Em 5o. Workshop de Desenvolvimento Baseado em Componentes, Brazil, 2005.

[Seacord98] R. Seacord; S. Hissam, C. Wallnau. Agora: A Search Engine for Software Components, CMU/SEI-98-TR-011, 1998.Seacord, Robert C. Software engineering component repositories. Technical Report,

[Seacord99] Robert C. Searcord, "Software Engineering Component Repository", Proceedings of 1999 International Workshop on CBSE, Los Angeles, at URL: http://www.sei.cmu.edu/cbs/icse99/cbsewkshp.htm, acessado em 1999

[Shaw+96] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline – Prentice Hall, 1996.

[Silva+2003] Moacir Silva Jr, Paulo Asterio de C. Guerra, and Cecília Mary F. Rubira. A java component model for evolving software systems. In Proceedings of IEEE International Conference on Automated Software Engineering (ASE'2003), October 2003.

[Silva2003] Moacir Silva Jr. COSMOS - Um Modelo ed Estruturação de Componentes para Sistemas Orientados a Objetos. Dissertação de Mestrado. IC-UNICAMP. 2003

[Sommerville01] Ian Sommerville. Software Engineering. Addison-Wesley, 6th edition, 2001.

[Szyperski97] C. Szyperski. Component Software – Beyond Object Oriented Programming, Addison-Wesley, 1997

[Tomita+04] Rodrigo T. Tomita, Fernando Castor Filho, Paulo A. C. Guerra, Cecília M. F. Rubira. "Bellatrix: Um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes", In 4o. Workshop de Desenvolvimento Baseado em Componentes, Brazil, 2004.

[Tomita+04a] Rodrigo T. Tomita, Fernando Castor Filho, Paulo A. C. Guerra, Cecília M. F. Rubira. "Bellatrix: Um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes", In 4o. Workshop de Desenvolvimento Baseado em Componentes, Brazil, 2004.

[Vitharana03] Padmal Vitharana, Fatemeh “Mariam” Zahedi, and Hemant Jain. Knowledge-Based Repository Scheme for Storing and Retrieving Business Components: A Theoretical Design and an Empirical Analysis. Em IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 29, NO. 7, JULY 2003

[Voth04] Dana. Voth. Packaging Reusable Software Assets. Em IEEE Software, Volume 21, Issue 3, May-Jun 2004 Page(s):107 - 108, 110.

[WCOP96] WCOP’96. International workshop on component-oriented programming. http://sky.fit.qut.edu.au/~szypersk/WCOP96/. Acessado em Janeiro, 2006.

[Wikipedia] Definição para metadados. http://pt.wikipedia.org/wiki/Metadados. acessado em janeiro/2006.

[Westfechtel+01] B. Westfechtel, R. Conradi : Software architecture and software configuration management. In A. van der Hoek, ed.: 10th International Workshop on Software Configuration Management: New Practices, New Challenges, and New Boundaries (SCM 10), Toronto, Canada (2001) 19—26

[W3XML] Especificação XML em W3. http://www.w3.org/TR/REC-xml/ . Acessado em janeiro, 2006.

[W3Schema] Especificação XML Schema em W3. http://www.w3.org/TR/xmlschema-1/ . Acessado em Janeiro, 2006.

[Ye01] Yunwen Ye. Supporting Component-Based Software Development with Active Component Repository Systems. Tese de doutorado, Universidade do Colorado, EUA, 2001.

Apêndice A Casos de Uso do Repositório Rigel

Neste apêndice apresentamos os casos de uso considerados no projeto do repositório Rigel. A Figura 29 contém o diagrama de casos de uso. Estes casos de uso estão agrupados em três grupos:

1. Editar bens e armazenar artefatos
2. Recuperar bens e artefatos
3. Pesquisar bens

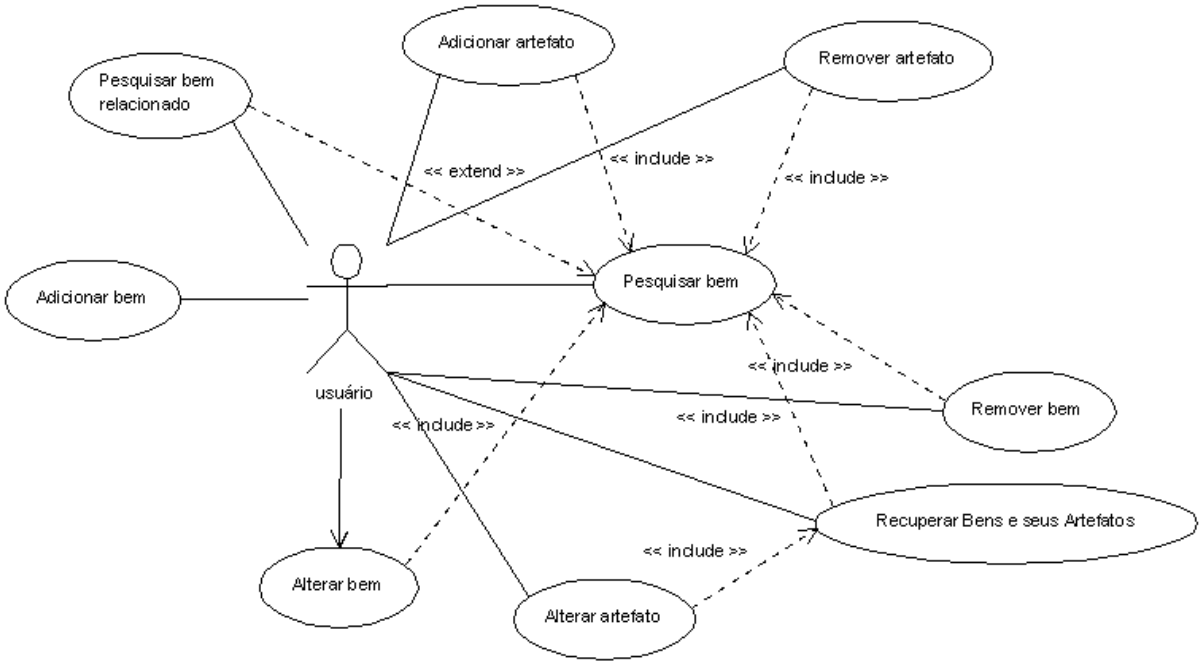

Figura 29 – Diagrama de casos de uso para o repositório Rigel

A.1 Editar bens e armazenar artefatos

A seguir temos os casos de uso relacionados ao requisito de editar bens e armazenar artefatos.

1. Adicionar Bem

Objetivo: adicionar um bem ao repositório.

Pré-condição: o bem não existe no repositório.

Pós-condição: o bem está armazenado no repositório

1) O usuário seleciona a opção de Adicionar bem
2) O repositório abre um tela de cadastro do bem
3) O usuário cadastra os dados do bem
4) O usuário confirma os dados
5) O repositório armazena o bem.

2. Remover Bem

Objetivo: remover um bem existente no repositório.

Pré-condição: o bem existe no repositório

Pós-condição: o bem é removido do repositório

1) O usuário seleciona a opção de remover bem
2) O usuário seleciona um bem. Incluir caso de uso: pesquisar bem
3) O usuário clica na opção excluir
4) O repositório solicita a confirmação pelo usuário
5) O usuário confirma
6) O repositório exclui o bem.

3. Alterar Bem

Objetivo: alterar dados do bem existente no reposiótio

Pré-condição: o bem existe no repositório

Pós-condição: os dados do bem são alterados.

1) O usuário seleciona a opção de Alterar bem

2) O usuário seleciona um bem. Incluir caso de uso: pesquisar bem

3) O usuário clica na opção alterar

4) O repositório abre a tela de cadastro do bem, exibindo os dados do bem.

5) O usuário modifica os dados do bem

4) O usuário confirma os dados

5) O repositório armazena os novos dados do bem.

4. Adicionar artefato no repositório

Objetivo: Adicionar um artefato no repositório.
Pré-condição: O usuário ter criado o artefato e o bem.
Pós-condição: O artefato estar armazenado no repositório.

1) O usuário seleciona a opção Adicionar artefato.
2) O sistema abre uma tela onde o usuário pode selecionar o artefato que deseja adicionar (escolher o path).
3) O usuário seleciona o artefato que deseja adicionar e confirma.
4) O sistema solicita o bem para o usuário. Inclui caso de uso Pesquisar bem;
5) O sistema solicita meta-informações sobre o artefato.
6) O usuário preenche as meta-informações sobre o artefato e confirma.
7) O sistema armazena o artefato no repositório.

5. Alterar artefato

Objetivo: Alterar um artefato no repositório.
Pré-condição: O artefato já deve estar contido no repositório.
Pós-condição: O artefato estar contido no repositório como uma nova versão.

1) O usuário recupera o artefato do repositório. Inclui caso de uso Recuperar artefato;
2) O usuário altera o artefato que foi recuperado utilizando o editor apropriado;
3) O usuário seleciona a opção Alterar artefato;
4) O usuário seleciona o bem que contém o artefato. Inclui caso de uso Pesquisar bem;

5) O sistema abre uma tela onde o usuário pode selecionar path do artefato;

6) O usuário seleciona o path do artefato que deseja adicionar e confirma;

7) O sistema solicita o artefato para o usuário. Inclui caso de uso Pesquisar bem;

8) O usuário seleciona o artefato a ser alterado;

9) O sistema exibe as meta-informações sobre o artefato;

10) O usuário não altera as meta-informações sobre o artefato e confirma;

11) O sistema armazena o artefato no repositório.

Extensão:

10) O usuário altera as meta-informações sobre o artefato e confirma;

11) O sistema armazena o artefato e as meta-informações alteradas pelo usuário no repositório.

6. Remover artefato

Objetivo: Remover um artefato no repositório.

Pré-condição: O artefato já deve estar contido no repositório.

Pós-condição: O artefato não estar contido no repositório.

1) O usuário seleciona a opção Remover artefato;

2) O sistema abre uma tela onde o usuário possa selecionar o artefato. Inclui caso de uso Pesquisar bem;

3) O usuário seleciona o artefato que deseja remover do repositório e confirma;

4) O sistema pergunta se o usuário realmente deseja remover o artefato;

5) O usuário confirma;

6) O sistema remove o artefato do repositório.

Extensões:

5) O usuário cancela.

A.2 Recuperar bens e artefatos

A seguir, apresentamos o caso de uso relacionado com a funcionalidade de recuperação de bens.

1. **Recuperar Bem e seus Artefatos**

Objetivo: recuperar artefatos que compõem um bem

Pré-condição: o bem existe no repositório

Pós-condição: os artefatos selecionados são copiados para o caminho (*path*) informado pelo usuário.

1) Incluir caso de Uso Pesquisar Bem.

2) O usuário seleciona o bem desejado

3) O repositório mostra os detalhes do bem

4) O usuário seleciona a opção de recuperar bem

5) O repositório requer o caminho (*path*)

6) O usuário informa o caminho (*path*) onde o bem será copiado

8) O repositório copia os artefatos para o caminho (*path*) informado

A.3 Pesquisar bens

A seguir apresentamos os casos de uso relacionados com a funcionalidade de pesquisa de bens.

1. **Pesquisar bem**

Objetivo: encontrar um bem.

Pre-condição: -

Pós-condição: -

1) O usuário seleciona a opção pesquisar bem

2) O repositório mostra a tela de pesquisa de bens

3) O usuário preenche os dados da pesquisa (palavras-chave) e clica em pesquisar

4) O repositório encontra bens.

5) O repositório exibe os bens encontrados.

Fluxo Alternativo:

4) O repositório não encontra nenhum bem.

5) O repositório mostra uma tela avisando que a pesquisa não encontrou nenhum bem.

2. **Pesquisar bem Relacionado**

Objetivo: encontrar um bem que está relacionado a outro bem.

Pre-condição: o usuário seleciona um bem

Pós-condição: -

1) Extends Pesquisar bem

2) O usuário seleciona o bem desejado

3) O repositório mostra os detalhes do bem

4) O usuário seleciona a opção de pesquisar bem relacionado

5) O repositório mostra os bens relacionados

Apêndice B Modelos de Profiles RAS

Neste apêndice, apresentamos todos os *profiles* RAS propostos neste trabalho.

O padrão RAS prevê o suporte a diferentes tipos de bens, através do conceito de "*Profiles*", especificações XML Schema que adicionam propriedades ao Core RAS. Os profiles definem a estrutura e semântica dos arquivos XML que contém os metadados do repositório. A identificação, o significado, o uso e o formato dos elementos XML são definidos pelos *profiles*.

Baseados na definição do Core RAS, criamos quatro Profiles: AbstractComponent, ConcreteComponent, InterfaceDefinition e Configuration Profile que descrevem respectivamente uma definição de interface, um componente abstrato, um componente concreto e uma configuração. Estas extensões do padrão RAS estão baseadas no modelo conceitual para DBC e arquitetura do software proposto por Guerra [Guerra05], a seção 2.3.1, contém mais detalhes sobre este modelo conceitual.

A seguir descrevemos em detalhes os modelos de metadados criados para o Rigel: *AbstractComponent, ConcreteComponent, InterfaceDefinition e Configuration*.

AbstractComponent

O *AbstractComponent* é modelo para componentes abstratos (Figura 30). Os artefatos com especificações do componente podem ser armazenados em *Asset.Solution.Artifact*. As interfaces referenciadas devem ser armazenadas em bens do tipo InterfaceDefinition. A ligação entre o componente abstrato e suas interfaces é dada por: *Solution.Design.AbstractComponent*. O elemento *AbstractComponent* contém algumas propriedades relativas a especificação do componente:

- *Name* – nome do componente.
- *QualityOfServiceContract* – contém informações sobre a qualidade do serviço provido pelo componente.

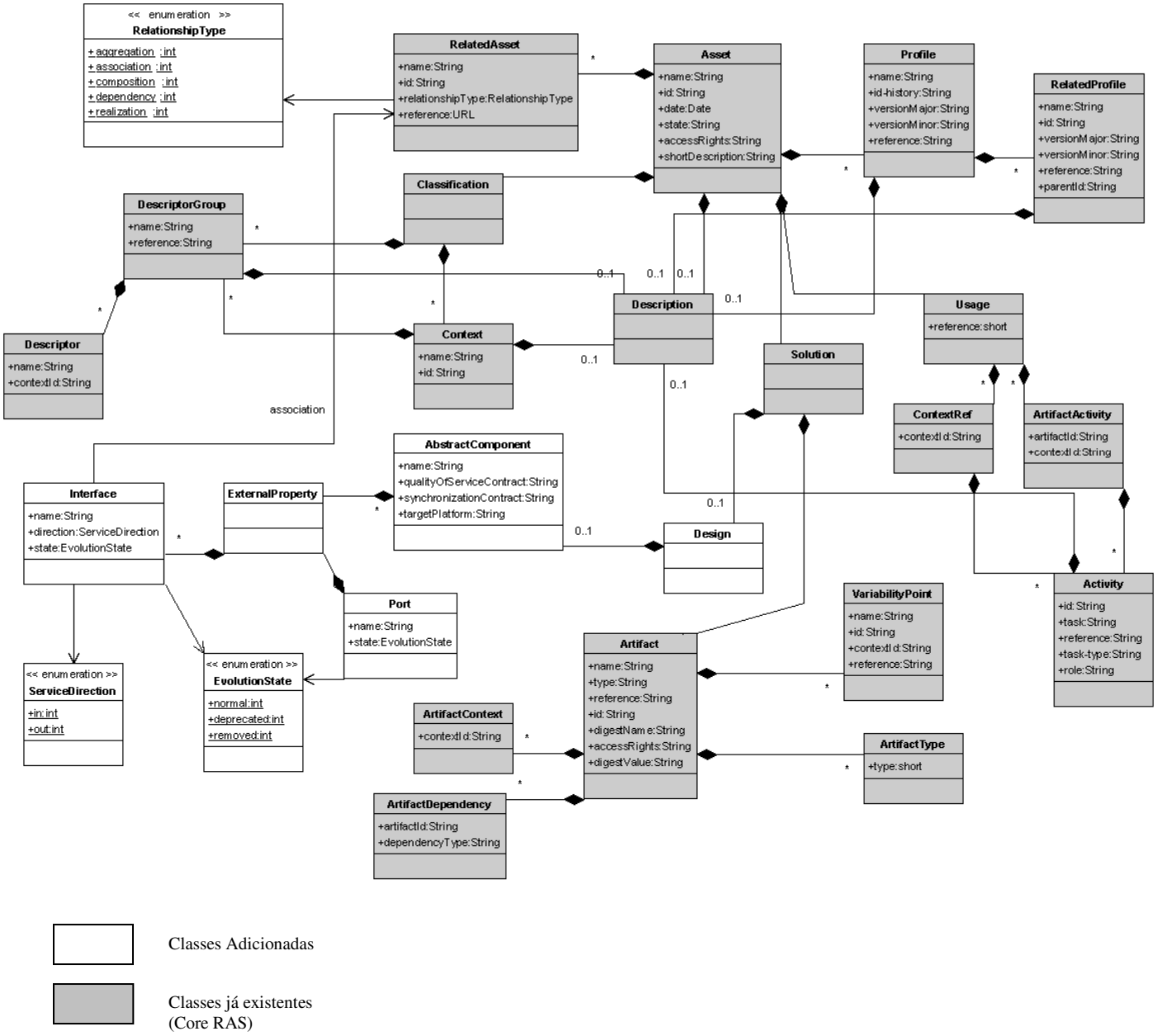

Figura 30 – modelo RAS para componentes abstratos

- *SynchronizationContract* – contém informações sobre a sincronização das operações de um componente.
- *TargetPlatform* – indica a plataforma alvo do component.

AbstractComponent contém elementos *ExternalProperty*, que por sua vez contém interfaces.

AbstractComponent contém elementos *Interface.* Elementos *Interface* se ligam com bens de interface através de elementos *RelatedAsset*. Os elementos *Interfaces* contém as seguintes propriedades:

- *Name* – nome da interface.
- *Direction*, com os valores *in* para interfaces requeridas e *out* para intefaces providas..
- *State*, com os valores: *normal, deprecated, removed*

ExternalProperty é um elemento intermediário entre *AbstractComponent* e *Interface* que agrupa interfaces em portas, através da utilização do elemento *Port*.

ConcreteComponent

O modelo para componente concreto pode representar um componente sem dependências, denominado componente elementar, ou um componente dependente de outros componentes, denominado componente composto (Figura 31).

Componentes elementares são representados através da propriedade *Asset.Solution.Implementation.ElementaryComponent*.

A representação de componentes compostos deve conter referências para os componentes dependentes. Estas referências estão armazenadas em *Asset.Solution.Implementation.CompositeComponent.ComponentBasedView*.

ComponentBasedView pode referenciar uma interface ou uma porta, para isso ele possui dois elementos: *InterfaceConnection*, que referencia uma interface e ServiceConnection que referencia uma porta. *InterfaceConnection* e *ServiceConnection* referenciam os bens que contêm interface e portas respectivamente através dos sub-elementos *SubComponent* e *RelatedAsset*.

Os artefatos que contêm a implementação do componente são armazenados em *Asset.Solution.Artifact*.

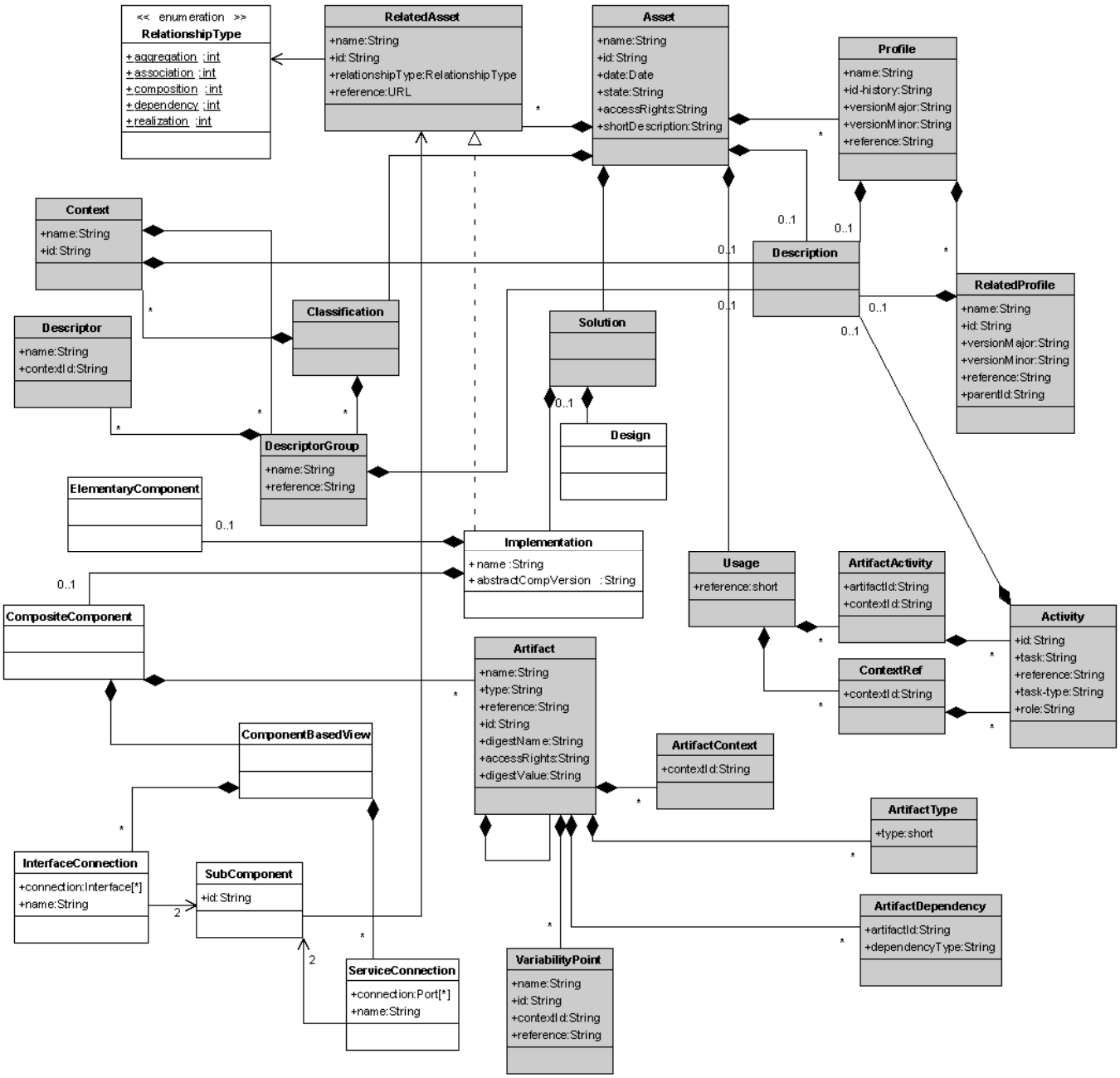

Figura 31 – modelo RAS para componentes concretos

InterfaceDefinition

O modelo *InterfaceDefinition* armazena bens contendo interfaces (Figura 32). As definições de interface estão em *Asset.Solution.Design.InterfaceDefinition*. Cada elemento *InterfaceDefinition* tem definições para uma interface. São propriedades de *InterfaceDefinition*:

- *Name* – nome da inteface.
- *Description* – descrição textual da inteface.
- *DevelopmentState*, que pode conter os valores:
  - *Development*: para interfaces que ainda estão sendo desenvolvidas.
  - *Published*: para interfaces estáveis, já implementadas e publicadas para serem reutilizadas.
  - *Discarded*: para interfaces que não funcionam mais, e portanto, não deverão ser utilizadas novamente.

InterfaceDefinition contém elementos *Operation*, que descrevem detalhes de operações da interface. São atributos de *Operation*:

- *Name* – nome da operação.
- *Description* – descrição textual da operação.

Cada operação pode conter parâmetros, modelados em elementos *Parameter* com as seguintes propriedades:

- *Name* – nome do parâmetro.
- *Type* – tipo do parâmetro.
- *Direction* – direção dos parâmetros. Pode ter os valores: *in* para parâmetros de entrada e *out* para parâmetros de saída.
- *Position* – índice que indica a posição do parâmetro.

Cada operação pode conter condições, modeladas em elementos Condition com as seguintes propriedades:

- *Type*: com os possíveis valores: *preCondition* para pré-condições, *posCondition* para pós-condições e *invariant* para invariantes..
- *Description* – descrição textual da condição.
- *Expression* – expressão que modela a condição.

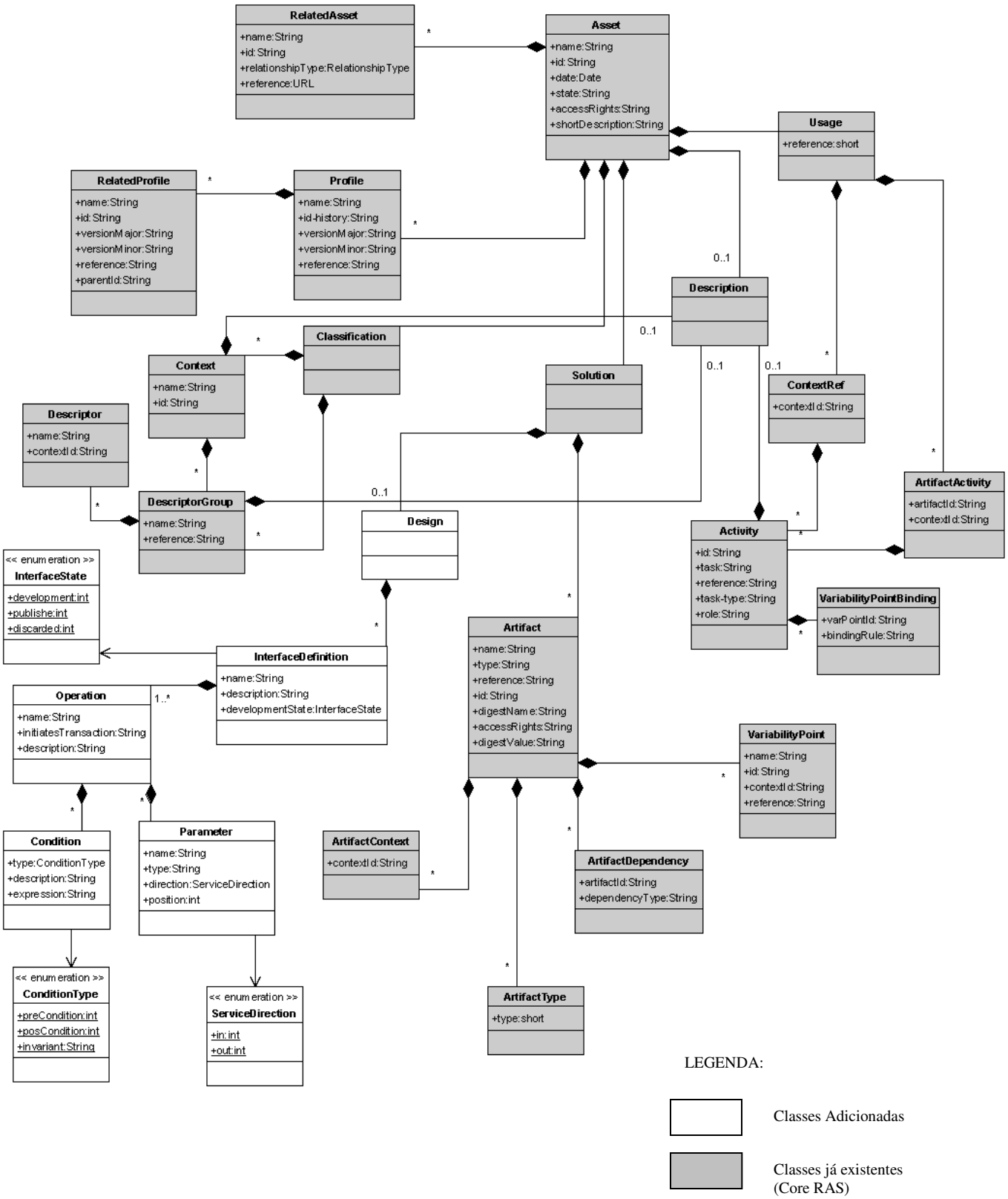

Figura 32 – modelo RAS para interfaces.

Configuration

O modelo *Configuration* armazena uma configuração de uma arquitetura concreta (Figura 33). Para isto, cada instância do modelo referencia todos os bens de componentes concretos que compõem uma arquitetura. Essas referências para arquiteturas são feitas através de *Asset.Solution.Configuration.ConcreteInstance*. *ConcreteInstance* contém referências a outros bens através de elementos *RelatedAsset*.

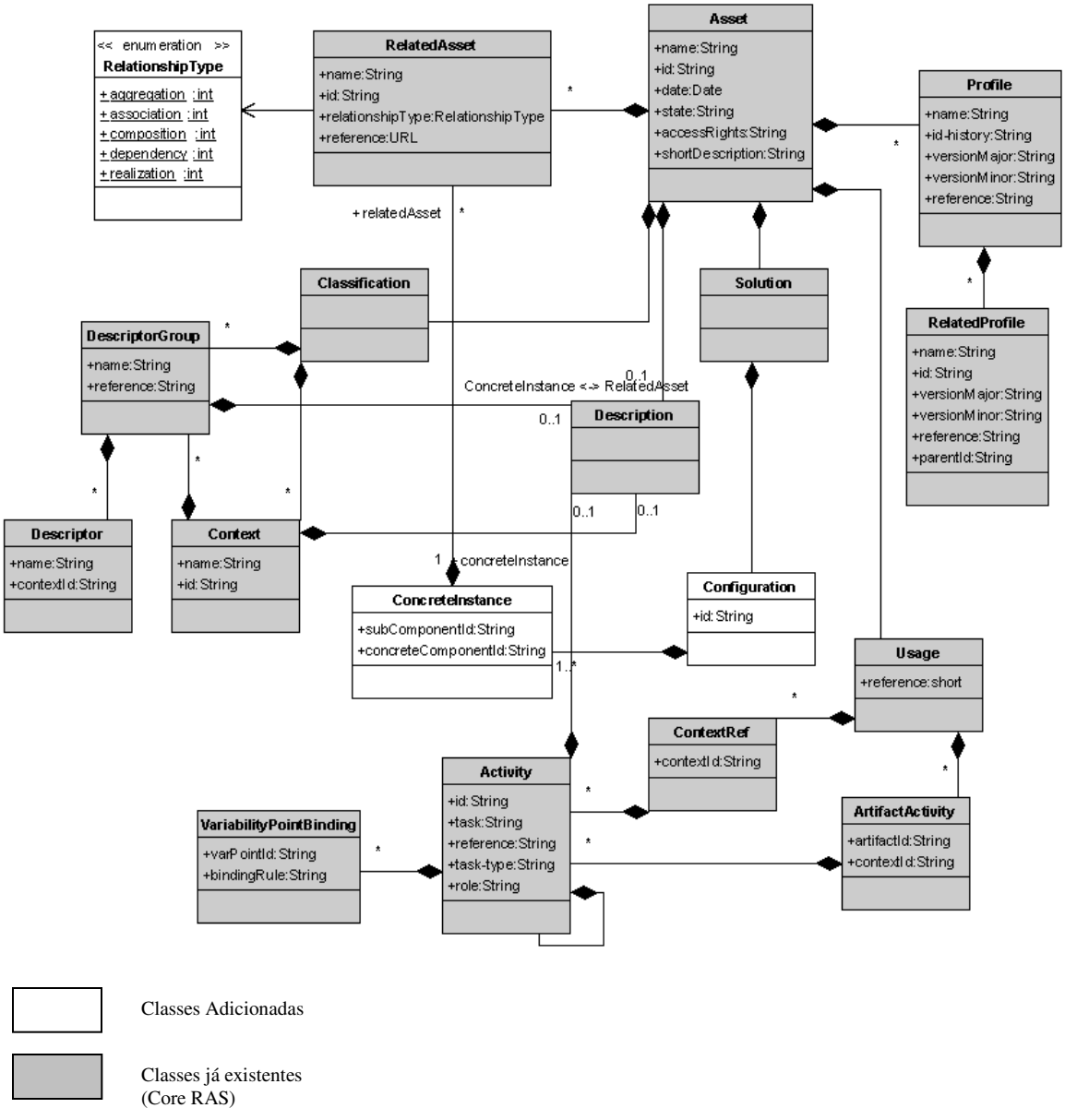

Figura 33 – modelo RAS para representação de arquiteturas concretas.